PHD DISSERTATION

"FEDERICO II" UNIVERSITY OF NAPOLI

DEPARTMENT OF INFORMATION TECHNOLOGY
AND ELECTRICAL ENGINEERING

DOCTOR OF PHYLOSOPHY IN
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

AUTOMATED OFFENSIVE SECURITY: INTELLIGENCE IS ALL YOU NEED

FRANCESCO CATURANO

PhD coordinator
Prof. Daniele RICCIO

Supervisor
Prof. Simon Pietro ROMANO

Cycle XXXIV

Acknowledgments

First and foremost, I would like to thank my PhD supervisor, Prof. Simon Pietro Romano. When I was a student I used to visualize myself as one of his collaborators and having the chance to become one was a wonderful experience, both workwise as well as personally.

Then, I would like to thank my partner in crime, fellow PhD student Gaetano Perrone, who teached me a lot and shared the doctorate path all along. I am sure that your talent and passion will bring you success and happiness. I would also like to thank all the people met within the research field during the last three years, starting from the guys at ARCLAB. Each one of them has represented a unique experience, allowing me to grow professionally and as a human being.

My thanks also go to all other ITEE PhD students. Only them can truly understand what it means to go through tough times and how to overcome them.

My sincerest gratitude goes to professors, researchers, and postdocs at Department of Information Technology and Electrical Engineering. They have teached me so much and prepared me for the many more challenges to come in the nearest future.

Thanks especially to ITEE PhD coordinator Daniele Riccio, who advised PhD students throughout the doctorate.

Thanks to all the students that I got to meet, help and teach during the past years. I enjoyed my time with you as if I still was one of you.

My biggest thanks to the guys at SecSI. We part ways at the end of this PhD, with the promise of keeping in touch and still work together in the years to come. I am sure that your passion and dedication will lead to great things for cybersecurity in Naples.

Finally, huge thanks to my family and my friends. The best part of achieving something is that you get to share the results with the ones you love.

As usual, thanks to music and movies: the first one for keeping me grounded, the second ones for making me dream.

Contents

List of Figures

Abstract

Offensive security is the practice of testing security measures from the adversary's perspective. Though it is constantly growing from a set of disorganized hacking practices to a mature and separate engineering discipline, most of it still relies on personal experience and skills. Tools that automate security testing, perform well when they have to provide hints on what is the most promising attack plan to conduct. However, they heavily rely on inefficient business logic models, such as brute force. These are far away from the way human testers would work, who try to be as precise and efficient as possible. This Thesis deals with offensive security, by exploring a few approaches to its automation that are inspired by the way security experts would act.
First, a Reinforcement Learning-based intelligent agent that performs discovery of Cross-Site scripting vulnerabilities, is presented. In particular, the design and implementation of an interactive Reinforcement Learning environment are discussed. Such a framework allows the agent to learn autonomously, through interactions with the environment, the policy that an expert penetration tester applies to look for such vulnerabilities in a web application. The final platform is evaluated with respect to other popular automated frameworks, in order to show the improvements in terms of accuracy and efficiency.
Then, an approach to create an ontology for web application penetration testing, representing the knowledge of such a context in the form of a knowledge graph, is showed. The purpose of this work is to create an expert system that recommends the best actions to perform during a penetration test, by making inferences that output the most promising attack paths.
Finally, a toolset for collecting actions performed during a web application penetration test, such as browser interactions as well as generated network traffic, is presented. Such a platform is capable of creating hacking sessions datasets, in order to promote research in the field of machine learning applied to cybersecurity.

Chapter 1

Introduction

1.1 Offensive Security

Over the years, the practice of exploitation has improved the understanding of what it means for computers to be secure. It has grown from a set of disorganized hacker conventions into a distinct engineering discipline that supports an entire industry. Nowadays, it is an area that seeks for its own definition, in order to be understood beyond its own confines, by makers of law and policy. For instance, when the Computer Fraud and Abuse Act (CFAA, year 1986) was written, each computer had its relatively small and well-defined set of authorized users. The concept of "unauthorized" access had no point of existence, since no servers were meant for random members of the public. Then the World Wide Web happened, and connecting to computers without any kind of prior authorization became not just the norm but also the foundation of all related business. Therefore, research in the field of Offensive Security is encouraged in order to find not only its definition, but also its place within the industry, law regulation as well as schools and universities. This will avoid policy-makers designing their own language, which would be disastrous to the future state of computer security.
Exploitation is another kind of programming and computer security depends on the understanding that we have of such a technique. This means that all models of unintended execution need to be understood if one wishes to eliminate them.
However, exploitation is only the core of an extended set of practices that expert hackers employ to perform attacks to high profile organizations. Therefore, Offensive Security can be intended as the set of practices for testing com-

puter security measures from an adversary's perspective. Many of these practices are discussed in the subsequent sections.

1.2 Ethical disclaimer

The New Hacker's Dictionary [1] defines a hacker in one of the following ways:

> **HACKER** noun 1. A person who enjoys learning the details of computer systems and how to stretch their capabilities—as opposed to most users of computers, who prefer to learn only the minimum amount necessary. 2. One who programs enthusiastically or who enjoys programming rather than just theorizing about programming.

Such definition was popular in the computer science community when intrusions did not represent actual threats, but were rather fairly benign. Over time, though, such intrusions became progressively noticeable to the point of being responsible for damages to the well being of entire organizations. Instead of using the more accurate term of "computer criminal," the media began using the term "hacker" to describe individuals who break into computers for fun, revenge, or profit. Since calling someone a "hacker" was originally meant as a compliment, computer professionals prefer to use the term "ethical hacker" to distinguish security enthusiasts from those hackers who turn to the dark side of hacking. Organizations in search for a solution to ever increasing intrusions, realized that one of the best ways to evaluate their security measures in response to intruder threats would be to have independent computer security professionals attempt to break into their computer systems, the same way a company would hire independent auditors to verify its records. "Ethical hackers" reproduce the behavior of criminal hackers using their tools and techniques, but being careful in neither damaging the systems under test nor steal information. Instead, they would evaluate the target systems' security and report back to the owners with the vulnerabilities they found as well as remediation instructions [2]. The author of the work discussed in this Thesis is fully aware of the sensitivity of the covered topics and strongly adheres to the principles of ethical hacking, believing in the improvements it can bring to the security of our computers.

1.3 Penetration Testing

The very first occurrence in the literature of the term Penetration referred to software testing is in a paper by R.R. Linde [3] (year 1975), who says "penetration tests are used to examine an implementation and from these analyses infer areas of possible design weakness". Nowadays, it is considered as the set of computer security practices to uncover vulnerabilities, emulating real attacks. It is also a process that consciously avoids the risks the exploitation of the found vulnerabilities. Penetration testing is most powerful when fully integrated into the development life cycle, so that findings can help improve design, implementation, and deployment practices [4]. There are several possible approaches to perform penetration test, depending on the amount of information provided to the security professionals, the defense mechanisms employed as well as the position, relative to the target infrastructure, occupied by the ethical hackers.

- **White box**: The attacker has detailed prior knowledge of the target. All known hacking techniques are adopted and it is important to know how to classify vulnerabilities according to their level of risk.
- **Black box**: The attacker has no prior knowledge of the target system.
- **Overt**: Employees are aware of ongoing tests. Therefore, some techniques for gaining access to the system such as social engineering, which tend to compromise users, instead of machines, are not performed.
- **Covert**: Employees are not aware of ongoing tests. The target system .
- **External**: The attacker is outside the target's network and attempts to access via the Internet.
- **Internal**: The attacker is inside the target's network, therefore from a position of advantage to test the target infrastructure.

Several are the phases that compose a penetration test. It is hard to find a standardized methodology, since it is a process that still relies a lot on personal experience and skills. However, many authors seem to agree on the following lifecycle [5]:

- **Planning phase**: the objective of the assignment is defined. Agreements like NDA (Non Disclosure Agreements) are signed by the parties in order to avoid the leak of sensitive information. After the management

consent, the penetration testing team receives information according to the type of activity that needs to be put into place. Such information can include operational procedures, security policies as well as infrastructural details, in order to define the scope for the test.

- **Discovery phase**: also known as the information gathering phase. Such phase is characterized by the penetration testing team scanning and enumerating the target system in order to capture as much information as possible about the health status of the network, as well as the exposed services. It can be carried out in both a stealthy way, looking only at public information (e.g., public repositories, documents, mailing lists, web profiles etc.) as well as a bit more intrusive (e.g., by performing port scanning, discovering firewall rules, matching OS fingerprints, etc.) The information captured during this phase constitutes a knowledge base from which the team can derive an attack plan.

- **Exploitation phase**: It is a crucial phase that allows penetration testers to accurately verify whether the suspected issues discovered in previous phases are actually vulnerabilities that represent a danger for the organization. This is done by providing a proof-of-concept of the exploitation process. Such proof has to be effective enough in order to convey the damage that might be done after an actual exploitation. However, it should not compromise the target system in the same way a real attack would do.

- **Reporting phase**: The report writing usually begins in parallel with the previous phases, although it is completed after the exploitation phase is over. A successful report details all the findings and their impacts to the organization. It takes into account both the technical and management aspects, in order to let the target organization quickly prioritize the remedy interventions. The report should be edited in a way that allows people with poor security background to understand and even reproduce the presented issues.

1.4 Web Application Penetration Testing

Nowadays, web applications are among the main targets of attacks and so this has justified the creation of a dedicated branch of Penetration Testing, typically referred to as Web Application Penetration Testing (WAPT). There are several

well established and famous methodologies that may drive a web application penetration tester during his analysis. Regardless of the particular methodology, the following are a set of security industry guidelines on how the testing should be conducted.

1.4.1 Web Application Hacker's Methodology

According to the famous Web Application Hacker's Handbook [6], the process is divided into a sequence of tasks organized according to the dependencies among them.

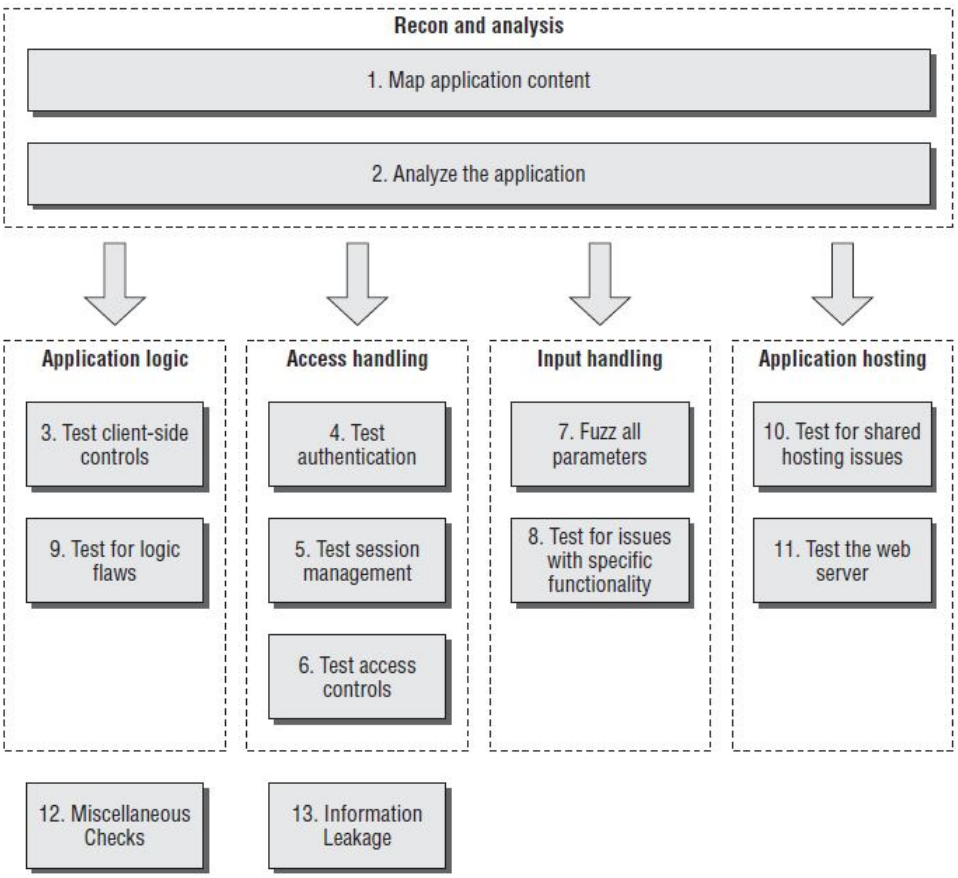

Figure 1.1: Web Application Hacker's Handbook PT Methodology

In figure 1.1 the first block includes two of the most critical phases because they affect the subsequent ones: it defines a series of rules to map the application structure such as setting a proxy that intercepts HTTP requests discovered through the page browsing, heuristic probes for discovering hidden contents, google dorks to find out public resources. It is also suggested to use automatic web spidering tools such as *DirBuster* to increase the coverage factor of the site map process. Once the application tree structure is discovered, the next step is to: (i) analyze the pages to classify functionalities such as authentication and session management; (ii) identify the entry points that may represent possible injection flaws (i.e., HTTP parameters that interact with the server); (iii) discover underlying technologies (JavaScript, PHP, ActiveX, Python). According to the clues retrieved during these two phases, one or more blocks in the second layer can be explored. This is the layer in which any vulnerabilities emerge. This process must not be considered "waterfall": each phase may discover important footprints that involve returning to the previous phases to be successfully investigated.

1.4.2 OWASP Testing Guide

The OWASP testing guide is a de-facto standard, published by OWASP, used to evaluate the security of a web application by methodically validating and verifying the effectiveness of application security controls through the use of tests. A web application security test is an action to demonstrate that an application meets the security requirements. This process is split into 11 testing subcategories and each one includes a set of specific tests identified by a unique name (Figure 1.2). As mentioned, the most common web application security weakness is the failure to properly validate input coming from the client or from the environment before using it. This weakness leads to almost all of the major vulnerabilities in web applications, such as XSS, SQL injection, Local File Inclusion, Path traversal and Buffer Overflows.

OTG-INPVAL-001 - Testing for Reflected Cross Site Scripting

The Cross-Site scripting attack is a perfect example of how a very common web vulnerability can become dangerous for users of a website. Its danger and popularity justify, in this Thesis, the use of Cross Site scripting as an example of the benefits we could obtain from an automated approach to its discovery. OWASP suggests a methodology to test for Cross-Site scripting vulnerabilities that consists of three phases:

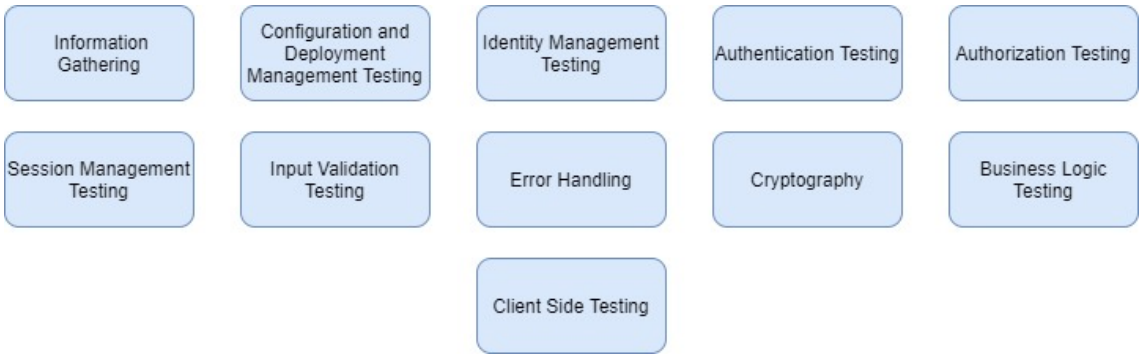

Figure 1.2: OTG v4.0 Testing Categories

1. **Detect input vectors**: the tester must determine all of the web application's user-defined variables and how to input them. This includes hidden or non-obvious inputs such as HTTP parameters, POST data, hidden form field values, and predefined radio or selection values.

2. **Analyze each input vector**: to detect potential vulnerabilities, the tester uses specially crafted input data with each input vector. Non-exhaustive examples of such input data are:

   - <script>alert(1)</script>
   - "><script>alert(document.cookie)</script>
   - " onfocus="alert(document.cookie)

3. **Analyze the results**: the tester analyzes the results of each input vector attempted in the previous phase and determines if it represents a vulnerability. Once found, the tester identifies any special character that was not properly sanitized.

1.5 Reinforcement Learning

Reinforcement Learning is a machine learning model in which an agent is trained to perform a task by interacting with an environment and receiving a feedback signal for each action performed. Such signal, called reward, is used by a Reinforcement Learning algorithm that allows the agent to improve through trial and error. The more positive are the rewards the agent gets during

training, the closer it gets to the final goal.
From an implementation point of view, in order to interact with such an environment, agents have to be equipped with sensors that provide information about the state of environment, actuators that allow them to perform actions and a mechanism that decides which action to perform next.
The basic assumption behind a Reinforcement Learning model, is that the environment can be formally described as a Markov Decision Process, composed by:

- **S**: is a set of states called the state space;
- **A**: is a set of actions called the action space (alternatively, A_s is the set of actions available from state s);
- $\mathbf{P_a}(\mathbf{s}, \mathbf{s}')$: is the probability that action a in state s at time t will lead to state s' at time t+1:

$$P_a(s, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a) \tag{1.1}$$

- $\mathbf{R_a}(\mathbf{s}, \mathbf{s}')$: is the immediate reward received after transitioning from state s to state s', due to action a.

The goal of the agent is to maximize the total amount of reward he receives from the environment in the long term:

$$E\left[\sum_{k=0}^{\infty} r_{t+k+1}\right] \tag{1.2}$$

The problem with equation 1.2 is the potentially infinite temporal horizon; since the common problems refer to a limited time, to solve this issue, one assumes to perform maximization over a fixed and limited time. So equation 1.2 becomes:

$$E\left[\sum_{k=0}^{T} r_{t+k+1}\right] \tag{1.3}$$

We can use equation 1.3, but usually in the practice a new factor is added:

$$E\left[\sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}\right] \tag{1.4}$$

Equation 1.4 is commonly referred to as Expected discounted future sum of rewards. Here γ is the discount factor satisfying $0 \leq \gamma \geq 1$ and it represents

the interest of the agent about that reward in the future. If γ is close to 1 the agent cares about this reward and it will be considered in the future; otherwise if γ is close to 0 the agent will not consider that reward for next decisions.
After describing the environment, the goal for the agent is to find a policy $\pi(s|a)$ that maps a state s to a probability distribution over actions a in order to maximize the long-run expected reward.
An interesting Reinforcement Learning algorithm is the so called Q-learning. It allows to learn estimates for the optimal value of each action, as the expected sum of future discounted rewards. Given a policy π, an estimate of the optimal value of an action a in a state s is:

$$Q_\pi = E[R_1 + \gamma R_2 + ...|S_0 = s, A_0 = a, \pi] \tag{1.5}$$

where $\gamma \in [0, 1]$ is the discount factor. The optimal value is

$$Q_*(s, a) = \max_\pi Q_\pi(s, a) \tag{1.6}$$

Selecting the highest valued action in each state, determines an optimal policy.

1.6 Contributions

This Thesis deals with Offensive Security by proposing different approaches to its automation. Such approaches take inspiration from the behavior of security experts and attempt to transfer it to intelligent agents using technologies related to the field of Artificial Intelligence. In particular, the contributions of this work are the following:

1. Design and implementation of an intelligent agent able to learn a methodology to discover Cross-Site Scripting vulnerabilities, using a model-free Reinforcement Learning algorithm.

2. Implementation of an automated and interactive Reinforcement Learning environment suitable for the training of the XSS agent.

3. Comparison with other state-of-the-art tools that deal with the automation of Web Penetration Testing, showing the improvements made in terms of accuracy and efficiency. Such improvements are attributable to the use of Reinforcement Learning algorithms.

4. Realization of an Ontology for Web Penetration Testing.

5. Representation of such an Ontology in the form of a Knowledge Graph;
6. Design and early implementation of a Recommender System for expert penetration testers based on the above-mentioned ontology.
7. Design and implementation of a toolset for collecting interactions with the browser during a Web Penetration Test, as well as generated network traffic, in order to create datasets of real-world hacking sessions.

The Thesis is structured as follows: chapter 2 reports a detailed analysis of the current state of the art for attempts to automate offensive security practices. Chapter 3 discusses improvements with respect to existing benchmarks. Chapter 4 presents a platform, based on Reinforcement Learning, able to suggest the best actions to a security tester during the process of discovering Cross-Site scripting vulnerabilities. In chapter 5, the design and implementation of a module that automates the interaction with a web application during the analysis of Cross-Site scripting vulnerabilities is presented. In chapter 6, the design and implementation of a fully automated intelligent agent for Cross-Site scripting vulnerabilities discovery is described, showing the integration of the previously presented modules as well as the realization of an interactive Reinforcement Learning environment based on GymAI. Chapter 7 evaluates the presented intelligent agent, by showing results in terms of accuracy and efficiency. Chapter 8 presents other approaches to offensive security automation through the application of Artificial Intelligence techniques. [7] Finally, in chapter 9 an architecture that combines infrastructure as code with different virtualization techniques to leverage a lightweight cyber range instantiation platform is proposed. Moreover, the design and implemetation of a security tomography that allows to assess the exposure of an ICT infrastructure to cyber security attacks is showed. Both approaches are results of real world applications.

Chapter 2

Related Work

In this chapter, works in the fields of Reinforcement Learning, XSS vulnerabilities discovery and penetration testing automation, are analysed.

2.1 Reinforcement learning environments

The fundamental source of inspiration for this work comes from the literature of Reinforcement Learning applied to Web Navigation tasks. Shi et al. in [8] introduce the World of Bits (WoB), a platform in which agents complete web navigation tasks by performing keyboard and mouse actions. After developing a methodology to log demonstrations, the authors show that agents trained via behavioral cloning and reinforcement learning can complete a subset of the web-based tasks. As the authors did with the WoB, we also collect a comprehensive list of test cases to support our penetration testing methodology. To the purpose, we leverage WAVSEP[1], the Web Application Vulnerability Scanner Evaluation Project, which includes a number of common Web Application vulnerabilities. With regards to Cross-Site Scripting, we upgraded WAVSEP by introducing new test cases, in order to cover all the possible ways to discover such a vulnerability.
Liu et al. in [9] induce high-level "workflows" that constrain the allowable actions at each time step to be similar to those performed during demonstrations. This allows the agent to easily identify successful workflows and avoids the stagnation of learning, that happens when the action space is big and the reward function sparse. In our work the agent has to learn both the best action

[1] https://github.com/sectooladdict/wavsep

to perform at each time step and the overall sequence of actions that allow to eventually discover the vulnerability. The sequence of actions is inferred by looking at expert demonstrations during a web penetration test and forms a workflow allowing to dynamically adjust the objective to be pursued in the environment, as well as apply separate techniques of discovery.
Of course, our work differs from [8] and [9] for the context, which is web application security testing. In fact, in both of the cited papers, the web site represents the environment upon which the agent takes actions and retrieves observations. In our case, the environment is made up by attack strings that can exist in different states of their formation, while the web application is used as a means to collect observations that allow the agent to move to the next step in the testing workflow.
Liu et al. in [10] propose the idea of Multi Objective Reinforcement Learning (MORL), as a way to solve the scaling problem for sequential decision-making frameworks. Compared to RL, MORL problems require an agent to be able to learn a policy that optimises multiple objectives at the same time. A difference can be made between related and unrelated objectives. In the former case, a single objective can be obtained by combining multiple objectives together. In the latter case, unrelated objectives can be separately optimised and a combined policy can subsequently optimise all of them. In our environment, we identify each attack string to be produced as a different objective. Since each attack string is constructed in a different way, depending on some parameters defined in section 4.1, we consider the objectives to be unrelated. Thus, we optimise them separately and then use a policy to decide, at an operational level, whether to pursue one objective or the other.
We deal with a parameterized action space, as defined by Masson et al. in [11], in the sense that the agent must select, at each step, both the action and the parameters to use with that action. However, our work differs from the cited one because we consider a discrete parameter space, rather than a continuous one.
Moreover, we do not use a single algorithm to learn a policy. We rather choose to optimise the parameters separately, using two hierarchically activated policies. This approach provides a decomposition of the reinforcement learning problem into sub-problems, as introduced in [12] by Dietterich.

2.2 XSS vulnerabilities discovery

As for the testing methodology leveraged to discover reflected XSS vulnerabilities, few attempts in the literature recall our approach. In [13], Lekies et al. propose an automated approach to discover DOM-based XSS vulnerabilities that allows for context-sensitive exploit generation. Depending on the context of reflection of the input, the attack string can be constructed in different ways. Our approach, though, identifies more contexts of reflection and therefore a higher number of possible attack strings. Moreover, the deconstruction of the attack string into several sub-strings, each with a specific semantics, is something that we leverage as well in order to perform the definition of the state. This allows to incrementally focus on specific parts of the attack string and modify them according to the hints provided, at each time step, by the application under test. However, we refine such deconstruction of the attack payload, by identifying other parts of the string that need to be taken into account throughout the testing. Also, the purpose of the cited work is to provide a taint-aware JavaScript engine, rather than a support tool for penetration testers. The idea of constructing attack strings based on the feedback provided by the web application, besides being employed by penetration testers for many years [6], has also been explored, with regards to reflected XSS vulnerabilities, by D'Amore et al. in [14]. The authors perform a categorisation of reflection contexts, accompanied by the respective attack payloads that trigger the XSS and propose *snuck*, a tool implementing the described methodology. Our approach, though, does not rely exclusively on the reflection context, but also on other parameters that influence the construction of the final attack string. Therefore, while the cited work employs a two-step testing (1. check the reflection context, 2. inject payload), our approach leverages a multiple step methodology allowing to refine an attack string at a fine grain in each of its constituent parts. Also, many more reflection contexts are taken into consideration. Neither Lekies et al. in [13], nor D'Amore et al. in [14] consider the use of reinforcement learning as an engine taking decisions in a specific environment.

Fang et al. in [15] propose an XSS adversarial attack model based on reinforcement learning, called RLXSS. The purpose is to optimize the detection of XSS attacks based on adversarial attack models. To do so, a reinforcement learning approach aims at identifying the most appropriate escaping technique. The authors show how this approach improves the detection capabilities against XSS attacks. We also leverage escaping techniques in our work, as they represent a common attack pattern deployed by hackers in order to craft a working attack

string. However, the purpose of the authors in [15] is to improve detection models at recognising XSS adversarial attacks. Our purpose is to leverage reinforcement learning to reproduce the attacker's behaviour and build a tool that supports penetration testers in the discovery of XSS vulnerabilities in web applications. Moreover, though escaping represents a core technique to build attack strings, we define a methodology that relies on other significant patterns as well.

2.3 Penetration testing automation

Interpreting penetration testing as a decision making process is a path explored in the past. In the last decade, reinforcement learning was the framework chosen to model such a process. Sarraute et al. in [16] and in [17] treat penetration testing as an attack planning model, namely in terms of a Partially Observable Markov Decision Process (POMDP). They do so in order to intelligently narrow down the list of scans to be performed in a network. Although being a thorough examination of how to face an incomplete knowledge of a network, the approach does not scale. It also needs a decomposition algorithm to identify the best direction to take after performing scans and exploits. We tackle this issue by reversing the perspective and starting directly from the end of a penetration test, which is the application of a specific exploit model. By doing so, we can simplify the actions and the states in order to make sure the approach scales better.
The same issue affects the work done by Ghanem et al. in [18] and [19]. They propose the "Intelligent Automated Penetration Testing System (IAPTS)" as a module that integrates with industrial PT frameworks to perform an autonomous assessment of network security. However, the action and state space is still too large and hence prevents the approach from scaling to real world scenarios. In fact, the platform strongly relies on experts to perform penetration testing and adjust the learning in the early phases, as opposed to creating a comprehensive model with clear actions that can be performed autonomously by an agent, as it is in our case.
The work done by Schwartz et al. in [20] frames penetration testing as a Markov Decision Process (MDP), includes the network configuration in the state space and uses scans and exploits as actions. The authors prove that Q-Learning can be a useful tool for finding the best attack paths. They also propose the Network Attack Simulator (NAS) as a benchmark for intelligent agents that perform scans and exploits on the network. It is a step forward in

the definition of a common ground against which different intelligent agents can be tested. Yet, the lack of a thorough definition of actions and states limits its applicability to real world scenarios.
Following the framework of a POMDP, Schwartz et al. in [21], take into account the defender's behaviour as a source of uncertainty that can affect decisions made by an autonomous penetration test agent. The authors show that even a simple defender's model can improve the capability of autonomous pen-testing. We do not take into account techniques that web applications might employ to defend against Cross-Site scripting attacks, such as character filtering, input validation or Web Application Firewalls. Such improvements represent one of the directions of our future work.
All of the cited approaches apply reinforcement learning to network penetration testing, in order to provide a means to simplify the attack plan in large-scale environments. Our work is instead an attempt at creating an intelligent agent that mimics the behavior of a penetration tester when applying a well known exploit model to test a web application vulnerability.

Authors F.M. Zennaro et al. in [22], [23], [24] and [25], built the most recent state of the art in terms of modeling penetration testing through reinforcement learning.

In particular, in [22] the authors observe these capture-the-flags type of challenges experimentally across a set of varied simulations and study how different reinforcement learning techniques may help addressing them. In this way they show the feasibility of tackling penetration testing using reinforcement learning, as well as highlight the challenges that must be taken into consideration and possible directions to solve them.

In [23] the authors focus their attention on simplified penetration testing problems expressed in the form of capture the flag hacking challenges and analyze how model-free reinforcement learning algorithms may help to solve them. They highlight how the biggest challenge for an agent is to discover the structure of the problem at hand and show how this problem may be solved relying on different forms of prior knowledge provided to the agent. By using techniques to inject a priori knowledge, the authors show that it is possible to better direct the agent and restrict the space of its exploration problem, thus achieving solutions more efficiently.

In [24], the authors propose a formalization of the process of exploitation of SQL injection vulnerabilities. They consider a simplification of the dynamics of SQL injection attacks by casting this problem as a security capture-the-flag challenge. The problem is modeled as a Markov decision process and

implemented as a reinforcement learning problem. We then deploy reinforcement learning agents tasked with learning an effective policy to perform SQL injection. we design our training in such a way that the agent learns not just a specific strategy to solve an individual challenge but a more generic policy that may be applied to perform SQL injection attacks against any system instantiated randomly by a problem generator. The results are analyzed in terms of the quality of the learned policy and in terms of convergence time as a function of the complexity of the challenge and the learning agent's complexity.

In [25] the same authors present the Agent Web Model, that considers web hacking as a capture-the-flag style challenge, and defines reinforcement learning problems at seven different levels of abstraction. The complexity of these problems in terms of actions and states an agent has to deal with, is discussed. They also provide an implementation for the first three abstraction layers, in the hope that the community would consider these challenges in order to develop intelligent web hacking agents.

In [26], Chowdhary et al. propose an autonomous security analysis and penetration testing framework (ASAP) that creates a map of security threats and possible attack paths in the network using attack graphs. The framework utilizes: (i) a state of the art reinforcement learning algorithm based on Deep-Q network (DQN) to identify optimal policy for performing penetration testing, and (ii) incorporates a domain specific transition matrix and reward modeling to capture the importance of security vulnerabilities, as well as the difficulty inherent in exploiting them. The ASAP framework generates autonomous attack plans and validates them against real-world networks. The attack plans are generalizable to complex enterprise network scenarios, and the framework scales well on a large network.

In [27], Castiglione et al. define a modular semi-automatic approach, which allows to collect and integrate data from various exploit repositories. These data are then used to provide the penetration tester (i.e., the pentester) with information on the best available tools (i.e., exploits) to conduct the exploitation phase effectively. Also, the proposed approach has been implemented through a proof of concept based on the Nmap Scripting Engine (NSE), which integrates the features provided by the Nmap Vulscan vulnerability scanner, and allows, for each vulnerability detected, to find the most suitable exploits associated with it. The proposed approach is focused on the automatic finding of the exploits that can be used to take advantage of the results achieved by the vulnerability scanning phase.

In [28], authors Maeda et al. propose a method of automating post-

exploitation by combining deep reinforcement learning and the PowerShell Empire, which is a renowned post-exploitation framework. The presented reinforcement learning agents select one of the PowerShell Empire modules as an action, whereas the state of the agents is defined by 10 parameters such as the type of account that was compromised by the agents. This work is similar to the one presented in this thesis, in fact it builds a training environment in which an agent can be autonomously trained using exploration, in this case to perform post-exploitation tasks. It uses also a similar definition of the state as the one presented in the next chapters, because it represents the levels of increasing privilege that an attacker gains by undergoing a sequential decision making process. However, the usage of deep reinforcement learning might seem appealing because it can ensure fast times of convergence, but it does not make much sense in this case where the definition of the environment does not encompass a feature representation of the states. In fact, the inputs are completely categorical, therefore it would be advisable in this context to use a more traditional approach, by leveraging tabular algorithms. This aspect will be underlined in the thesis.

In [29], Bland et al. model cyberattacks using an extension of the well-known Petri net formalism. The formalism models the attacker and defender as competing players who may observe the marking of a subset of the net and based on the observed marking act by changing the stochastic firing rates of a subset of the transitions in order to achieve their competing goals. Then, a reinforcement learning algorithm using an ϵ-Greedy policy was implemented and set to the task of learning which actions to take, i.e., which transition rates to change for the different observable markings, so as to accomplish the goals of the attacker or defender. In terms of design, the choice of using a formalism that captures the dynamics of the system, upon which implement a reinforcement learning algorithm, is probably better than the one of using tabular algorithms, or neural networks. In fact, the computational efforts of storing a table and perform read/write operations are avoided as well as the risks of having a neural network that approximates a tabular function.

The use of Petri Nets to model cyberattacks is well described in [30], which shows that they can be useful to provide additional knowledge on the planning stages of defense systems. The study introduces the formalism required to compose individual Petri Nets with Players, Strategies, and Cost models from a single system attack to a full system, which may include different methods of attacks being attempted, modeling a more realistic situation. The model composition described includes the sequential and parallel possibilities of multiple

attacks. An example of a possible attack scenario is described in order to demonstrate the practical application of the results.

The same authors also define, in [31] and [32], ways to build on the previously generated petry nets to construct composite attack scenarios. They also specify a way to start from the representation of a cyberattack via a petri net and perform decomposition methods that identify fine-grained as well as coarse-grained subnets, that can be reused to construct bigger cyber-attack nets.

Elderman et al. in [33] focus on cyber-security simulations in networks modeled as a Markov game with incomplete information and stochastic elements. A cyber attack is modeled as a game, in the form of an adversarial sequential decision making problem played with two agents, the attacker and defender. The two agents pit one reinforcement learning technique, like neural networks, Monte Carlo learning and Q-learning, against each other and examine their effectiveness against learning opponents. The purpose of the work is to create a defender agent able to dynamically adapt to the changes in the network caused by the attacks and therefore efficiently pick the right actions. Again, giving the categorical nature of the inputs to the problem, tabular-based algorithms seem to perform better than those based on neural networks.

Kujanpää et al. in [34] present an agent that uses a state-of-the-art reinforcement learning algorithm to perform local privilege escalation. Our results show that the autonomous agent can escalate privileges in a Windows 7 environment using a wide variety of different techniques depending on the environment configuration it encounters. The authors claim the agent is usable for generating realistic attack sensor data for training and evaluating intrusion detection systems. Because in cybersecurity it is easier to practically reproduce the attacks than finding a representation of the inputs by means of features, as done in other fields of artificial intelligence, reinforcement learning becomes a good choice.

Microsoft [35] has developed a research toolkit called CyberBattleSim[2], which enables modeling the behavior of autonomous agents in a high-level abstraction of a computer network . Reinforcement learning agents that operate in the abstracted network can be trained using the framework. The objective of the platform is to create an understanding of how malicious reinforcement learning agents could behave in a network and how reinforcement learning can be used for threat detection. Deep reinforcement learning can also be applied to improving feature selection for malware detection.

[2]https://github.com/microsoft/CyberBattleSim

Walter et al. in [36] incorporated deceptive elements, including honeypots and decoys, into the Microsoft CyberBattleSim experimentation and research platform. The defensive capabilities of the deceptive elements were tested using reinforcement learning based attackers in a capture the flag environment. The attacker's progress was found to be dependent on the number and location of the deceptive elements. It is a promising step towards reproducible testing attack and defense algorithms in a simulated enterprise network with deceptive defensive elements.

Standen et al. in [37] introduce CybORG, a work-in-progress gym for ACO (Autonomous Cyber Operation) research. CybORG features a simulation and emulation environment with a common interface to facilitate the rapid training of autonomous agents that can then be tested on real-world systems. They provide some promising test results that show the feasibility of the approach. Their work should support the application of machine learning algorithms to develop blue and read team decision-making agents.

Li et al. in [38] present CyGIL: an experimental testbed of an emulated RL training environment for network cyber operations. CyGIL uses a stateless environment architecture and incorporates the MITRE ATT&CK framework to establish a high fidelity training environment, while presenting a sufficiently abstracted interface to enable RL training. Its action space and game design allow agent training to focus on particular advanced persistent threat (APT) profiles and to incorporate a broad range of potential threats and vulnerabilities. It aims to leverage state of the art RL algorithms for application to real-world cyber defence.

Nguyen et al. in [39] present a survey of Deep Reinforcement Learning approaches developed for cyber-security. It shows how, among the several applications, the modern human-on-the-loop model would be a solution for a future human-machine teaming cyber security system. This model allows agents to autonomously perform the task whilst humans can monitor and intervene operations of agents only when necessary. How to integrate human knowledge into DRL algorithms [182] under the humanon-the-loop model for cyber defense is an interesting research question.

Mern et al. in [40] present techniques to scale deep reinforcement learning to solve the cyber security orchestration problem for large industrial control networks. The authors propose a novel attention-based neural architecture with size complexity that is invariant to the size of the network under protection. A pre-training curriculum is presented to overcome early exploration difficulty. Experiments show that the proposed approaches improve both the learning

sample complexity and converged policy performance over baseline methods in simulation.

Niculae in [41] formalizes penetration testing as a security game between an attacker who tries to compromise a network and a defending adversary actively protecting it. The thesis paper compares multiple algorithms for finding the attacker's strategy, from fixed-strategy to Reinforcement Learning, namely Q-Learning (QL), Extended Classifier Systems (XCS) and Deep Q-Networks (DQN). The attacker's strength is measured in terms of speed and stealthiness, in the specific environment used in our simulations. The results show that QL surpasses human performance, XCS yields worse than human performance but is more stable, and the slow convergence of DQN keeps it from achieving exceptional performance. In addition, all the Machine Learning approaches outperform fixed-strategy attackers.

In [42], Basori et al. use Reinforcement Learning to let a defense agent to adapt and learn the attacker's behavior or pattern.

Gangupantulu et al. in [43] present a novel method for crown jewel analysis[3] named CJA-RL that uses reinforcement learning to identify key terrain and avenues of approach for exploiting crown jewels. In their experiment, CJA-RL identified ideal entry points, choke points, and pivots for exploiting a network with multiple crown jewels, exemplifying how CJA-RL and reinforcement learning for penetration testing generally can benefit computer network operations workflows.

Also Gangupantulu et al. in [44] present methods for constructing attack graphs using notions from IPB on cyber terrain analysis of obstacles, avenues of approach, key terrain, observation and fields of fire, and cover and concealment. Authors demonstrate their methods on an example where firewalls are treated as obstacles and represented in (1) the reward space and (2) the state dynamics. They show that terrain analysis can be used to bring realism to attack graphs for RL.

2.4 Network Security Ontologies and knowledge graphs

Christian et al. in [45] present MalONT2.0 – an ontology for malware threat intelligence. It allows researchers to extensively capture all requisite classes

[3]Crown Jewels Analysis (CJA) is a process for identifying those cyber assets that are most critical to the accomplishment of an organization's mission.

and relations that gather semantic and syntactic characteristics of an android malware attack. Such an ontology forms the basis for the malware threat intelligence knowledge graph, MalKG. Here knowledge graphs are used to both provide captivating visualization of the threat intelligence domain, as well as to take advantage of graph theory in order to perform inferences that draw associations among entities that would otherwise not be recognized.

Wang et al. in [46] propose a network attack path prediction method based on knowledge graph and attack graph model, which uses CVSS's quantitative indicators for a single vulnerability, and combines the network security evaluation method to calculate the possible path. Experimental results show that this method can evaluate the security risk value of networks and nodes, which can point out the possible attack path of the attacker and calculate the risk value of the corresponding path. It can also rank the network node on the path and give repair suggestions.

Wang et al. in [47] propose a method to construct cyber-attack knowledge graphs based on CAPEC and CWE, using Neo4j as implementation platform. The authors claim that the knowledge graph proposed in their paper can be used as an intelligence source for multi-stage attack models known as Kill Chains.

Cermak et al. in [48] present an application of graph-based network forensics, a new approach to analyzing network traffic data utilizing modern database technologies capable of storing large amounts of information based on their associations. They introduce the GRANEF toolkit utilizing Dgraph database that stores transformed information from network traffic captures extracted by the Zeek network security monitor. The stored data are presented to the user via a web-based user interface that provides an abstraction layer above the database query language and allows the user to efficiently query data, visualize results in the form of a relationship diagram, and perform exploratory analysis.

Sarhan et al. in [49] present Open-CyKG: an Open Cyber Threat Intelligence (CTI) Knowledge Graph (KG) framework that is constructed using an attention-based neural Open Information Extraction (OIE) model to extract valuable cyber threat information from unstructured Advanced Persistent Threat (APT) reports. Security professionals can execute queries to retrieve valuable information from the Open-CyKG framework.

Qian et al. in [50] propose a novel ontology based BDI-agent RL automatic PT framework. By combining SWRL penetration testing knowledge base and RL in a BDI (belief-desire-intention) agent, the proposed model can

make use of the ontology based knowledge base (prior knowledge) to optimize the planning problem in the uncertain and dynamic environment. Finally, the simulation on ASL simulation platform Jason proved the new BDI-agent auto-PT model can improve the accuracy and speed up performance.

Hermanowski et al. in [51] proposed a method based upon the MulVAL reasoning engine that identifies possible attack paths leading from an attacker to pointed assets of an IT network. These paths create an attack graph used for attack probability calculation. The method takes advantage of information from vulnerability scanners and topology snapshot. In the paper, an enterprise network was examined and an attack graph based security evaluation presented. The case study probability calculations were provided, including possible remediation actions. Benefits and limitations of the proposed method are also discussed.

Kurniawan et al. in [52] developed a vocabulary to extend a cybersecurity knowledge graph with adversary tactics and techniques. Using this vocabulary, we represent rich threat intelligence instance data from MITRE Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) in a knowledge graph. This knowledge can be used to contextualize indicators of compromise from log messages, identify potential attack steps, and link them to cybersecurity knowledge. To demonstrate the benefits of the approach, we link low-level threat alerts produced by community rules to the cybersecurity knowledge graph.

Eliztur et al. in [53] presented the Attack Hypothesis Generator (AHG) which takes advantage of a knowledge graph derived from threat intelligence in order to generate hypotheses regarding attacks that may be present in an organizational network. Based on five recommendation algorithms developed by the authors and preliminary analysis provided by a security analyst, AHG provides an attack hypothesis comprised of yet unobserved attack patterns and tools presumed to have been used by the attacker. The proposed algorithms can help security analysts by improving attack reconstruction and proposing new directions for investigation. Experiments show that when implemented with the MITRE ATT&CK knowledge graph, our algorithms can significantly increase the accuracy of the analyst's preliminary analysis.

Hemberg et al. in [54] link MITRE's ATT&CK MATRIX of Tactics and Techniques, NIST's Common Weakness Enumerations (CWE), Common Vulnerabilities and Exposures (CVE), and Common Attack Pattern Enumeration and Classification list (CAPEC), to gain further insight from alerts, threats and vulnerabilities. Authors preserve all entries and relations of the sources, while

enabling bi-directional, relational path tracing within an aggregate data graph called BRON. For example, BRON can be used to enhance the information derived from a list of the top 10 most frequently exploited CVEs. Authors identify attack patterns, tactics, and techniques that exploit these CVEs and also uncover a disparity in how much linked information exists for each of these CVEs.

Kriaa et al. in [55] propose a novel approach that leverages a combination of both knowledge graphs and machine learning techniques to detect and predict attacks. Using Cyber Threat Intelligence (CTI), authors build a knowledge graph that processes event logs in order to not only detect attack techniques, but also learn how to predict them.

Kaloroumakis et al. in [56] created D3FEND, a framework in which they encode a countermeasure knowledge base, but more specifically, a knowledge graph. The graph contains semantically rigorous types and relations that define both the key concepts in the cybersecurity countermeasure domain and the relations necessary to link those concepts between each other. To demonstrate the value of this approach in practice, authors describe how the graph supports queries that can inferentially map cybersecurity countermeasures to offensive TTPs (Tactics, Techniques and Procedures). Authors also outline future D3FEND work to leverage the linked open data available in the research literature and apply machine learning, in particular semi-supervised methods, to assist in maintaining the D3FEND knowledge graph over time.

The purpose of the work in [57] is to propose an ontology-based automated penetration testing approach. The authors build their ontology basing on the following taxonomy for penetration testing:

- Information Gathering attack: these are attacks aiming to collect information about the target (IP address, opened ports, application information, OS information, human or organization information, network information, defense mechanisms and so on);

- Configuration attack: attacks based on system administrator's misconfiguration of the system under test; e.g. exposing the "robots.txt" file or the private ftp area;

- Buffer Overflow attack: typically, a buffer overflow occurs due to a mistake in a software's coding phase. Essentially, this vulnerability enables the attacker to overwrite the memory of the system by injecting arbitrary executable code;

- Password attack: it is a brute-force attack that aims to guess the password of a user or of the administrator;
- Web attack: it is an attack against a web application; they include: XSS, CSRF, SQLi and so on. The most dangerous web application attacks are listed in the OWASP top 10;
- Sniffer attack: when the attacker gains access to a private network, they can attempt to sniff traffic trying to catch some useful data;
- Social Engineering attack: they include Spear-Phishing Attacks by email or link, Website forge Attacks or Spoofing Attacks;
- Denial of Service attack (DOS): it is an attack which aims to exhaust a system's resources, preventing users from access it.

The authors build an ontology using Protégé. The built ontology has three main classes: attackers, target and attack methods. The target's data properties are: OS, IP address, port, configuration, application, vulnerability and current permission. The attack methods class consists of multiple levels of attack methods, based on the above taxonomy. The specific attack actions are defined as attack method's instances with the following data properties: action, precondition and post-condition. The authors define five object properties in order to describe the relations between instances:

- *hasPermission*: represents that the attacker has a specific permission on the target;
- *isConnected*: represents that the attacker instance can be connected to the target;
- *isNotConnected*: represents that the attacker instance can not be connected to the target;
- *isSameSubnet*: represents that the target instances which are in the same subnet.
- *exploitBy*: represents that attacker can perform a specific attack.

Authors added SWRL (Semantic Web Rule Language) rules to the built ontology. SWRL rules play a very important role in the reasoning process that leads to the discovery of a vulnerability. To automate the attacks stored

(together with the knowledge) in the ontology, authors added a BDI agent to the system. The latter, in fact, is performing actions such as load, query, update classes and properties, as well as perform reasoning. Python (BDIPython library) has been used to implement the BDI mechanisms. In this way, the SWRL rules are used to determine the attack actions (basing on the target information stored in the ontology), while the BDI mechanism is used to perform the attack. At the end of the attack, if a vulnerability is discovered, the ontology will be updated. This tool performs everything automatically: it starts with the information gathering phase and finishes with the exploit of the vulnerability.
The touching points with this paper are clearly the ontology implementation, the use of rules to perform reasoning and the creation of a knowledge base. The dissonance here is in the automation reached by the authors' tool: the scope of this thesis is to support the penetration tester's work, rather than automatically launch pre-existing tools and use them "as they are".

In [58], a large scale system, attempting to automate the penetration testing (intended as black-box testing) of large-scale heterogeneous systems, is presented. The paper's purpose is to narrow the gap between information that security experts obtain through their tools, and the knowledge they need in order to make security assessments on the system. The used penetration testing methodology is the following:

1. Collecting publicly available information: information is obtained through search engines, web-sites that contain statistics and registration information (e.g., whois, domaintools), social networks and forums, DNS name finders;

2. Automated testing: using automated tools, testing is performed on the collected data;

3. Pruning false positives: the problem of automated tools is the generation of false positives, which must be pruned off;

4. Unfolding found vulnerabilities and entry points: they look for entry points and possible vulnerabilities in the system;

5. Business logic testing: trying to exploit the vulnerabilities to penetrate the system.

The following steps have been followed to create the penetration testing ontology:

1. Form the dictionary of research domain and tasks: authors used a lot of different standard taxonomies (ARF, CAPEC, CCE, CCSS, CEE, CPE, CRF, CVE, CVRF, CVSS, CWE, CWSS, CybOX, IODEF, MAEC, MMDEF, OCIL, OVAL, SWID, WS-Agreement, XACML, XCCDF) and several general purpose ontologies;

2. Identify the concepts: The set of concepts could be divided into three subsets: the subset representing the process of security analysis; the subset representing the technical description of the system under test; the subset representing the description of the system's business logic;

3. Build hierarchy of concepts;

4. Identify non-hierarchical relations;

5. Develop rules;

6. Populate ontology by discovering new instances of concepts and relations;

7. Extend hierarchy of concepts.

Authors used OWL to create the penetration testing ontology and Protégé to edit it. Their ontology contains 519 concepts. Ontology concepts are interconnected through different relations: one to many, one to one, has subclass. Each relationship has its own name, to better explain it. In order to better explain the multi-dimensional ontology, authors give different views (intended as a "smaller ontology derived from, but independent of, the source ontology") of it. In order to explain the multi-dimensional ontology, authors give different views (intended as a "smaller ontology derived from, but independent of, the source ontology") of it. The given views are: host-centric, virtual identity-centric, vulnerability-centric and organization-centric. Thus, they build a tool for the automated penetration testing. The tool's basic workflow is the following: the semantic net is built using the previously proposed ontology with data from passive and active security scans, along with external data sources, associated with the information system under test. Inference rules and mappings, among retrieved data and ontological database, are manually formed by security experts. The built semantic net contains meaningful entities and relationships extracted from data sources. At this point, the inference engine automatically performs some penetration testing actions using manually built attack vectors. The whole acquired knowledge is finally presented to the user.

The similarities with this paper are: the construction of an ontology, in order to map security elements and relations among them; the ability to make inference through rules written by experts; the possibility to extend the tool to a wide range of systems. The differences are: the use of OWL, the automated use of security tools and the automatic data extraction.

2.5 Penetration testing datasets

Vsvabensky et al. in [59] present a dataset of 13446 shell commands from 175 participants who attended cybersecurity trainings and solved assignments in the Linux terminal. Each acquired data record contains a command with its arguments and metadata, such as a timestamp, working directory, and host identification in the emulated training infrastructure. The commands were captured in Bash, ZSH, and Metasploit shells. The data are stored as JSON records, enabling vast possibilities for their further use in research and development. These include educational data mining, learning analytics, student modeling, and evaluating machine learning models for intrusion detection. The data were collected from 27 cybersecurity training sessions using an open-source logging toolset and two open-source interactive learning environments. Researchers and developers may use the dataset or deploy the learning environments with the logging toolset to generate their own data in the same format. Moreover, authors provide a set of common analytical queries to facilitate the exploratory analysis of the dataset.

Vsvabensky et al. in [60] presented a modular toolset for logging commands from Bash and Metasploit shell. It enables a data-driven understanding of students' approaches to practicing cybersecurity, system administration, and networking. Authors deployed the toolset in two learning environments and showed the value of the gained data by analyzing student approaches. Since the toolset can be applied in many other contexts, it can foster further research and development.

Chapter 3

Improvement of existing benchmarks

The first activity conducted to tackle the automation of tools for web penetration testing is understanding how to evaluate their effectiveness when executed against real web applications. In the ethical hacking community, beginner students that are eager to become familiar with penetration testing activities, tend to perfect their skills using vulnerable applications, designed with the purpose of creating virtual environments where hacking techniques can be practiced safely. Nowadays, such environments have evolved into complex cyber-ranges, which are used by companies to train security experts in both defensive and offensive scenarios. The open source community has developed a few of such platforms over the years[1], however they usually lack the necessary requirements to be employed as fully-fledged web application security benchmarks in order to test Dynamic Application Security Testing (DAST) tools. Such requirements, like having diversified test suites, support for automated benchmark test's execution and so on [61], seemed to have been fulfilled in the past by an open source project called WAVSEP (Web Application Vulnerability Scanner Evaluation Project), which was used as a benchmark platform to test DAST tools. However, the project is no longer maintained. This chapter describes the efforts made to take the old version of WAVSEP and bring it up to date to the current state of the art of known web application vulnerabilities.

[1] https://owasp.org/www-project-vulnerable-web-applications-directory/

3.1 WAVSEP: Web Application Vulnerability Scanner Evaluation Project

Web Application Vulnerability Scanner Evaluation Project (WAVSEP) is an open-source project released by the researcher Shay Chen. It is an intentionally vulnerable web application used mainly as a benchmark to evaluate the quality, the functionality and effectiveness of DAST tools. Such a testing activity has been conducted in the past on WAVSEP, evaluating 63 different DAST tools in 2012, 2014 and 2016. Results of which are published in an understandable format at `http://sectooladdict.blogspot.com/`, in order to allow both researchers and users to use data for their own purposes. The project uses the JavaServer Pages (JSP) technology to implement vulnerable web pages that can be used as test cases. It also uses a MySQL database used to support SQL-related attacks.

In 2014, version 1.5 of WAVSEP was released, which is also the latest version available. It contains a total of 1191 vulnerable test cases as well as 40 test cases made up by intentionally non vulnerable web pages, in order to test tools under test's tendency to produce false positives. The classes of vulnerabilities present in WAVSEP 1.5 include SQL injection, Reflected Cross-site Scripting (RXSS), DOM cross-site scripting, Local File Inclusion (LFI), Remote File Inclusion (RFI) and information disclosure.

3.2 WAVSEP 2.0

Since the last update to WAVSEP 1.5 dates back to 2014, new test cases have been added in order to update it to the current state of the art. In fact, over the years new attack vectors for web applications have been discovered as well as brand new classes of vulnerabilities. Therefore, an analysis of the vulnerable training applications publicly available has been conducted, in order to identify test cases that are representative of the current state of web application vulnerabilities as well as their respective exploitation techniques. The list of vulnerable web applications published by OWASP [62] as well as the PortSwigger Academy interactive labs [63] were taken into account to enumerate the most popular exploitation techniques. Other accredited sources, such as "OWASP Top 10 2017" , "MITRE ATT&CK" , "AcunetixWeb Application Vulnerability Report 2020", "2020 CWE Top 25 Most Dangerous SoftwareWeaknesses", were considered to make sure that the classes of vulnerabilities considered were up to date. WAVSEP already offered some test cases for most of the vul-

Vulnerability	Added test cases
Reflected XSS (RXSS)[2]	39
SQL Injection (SQLI)	23
Blind SQLI	6
Local File Inclusion (LFI)	1
OS Command Injection	8
XML External Entity (XXE)	12

Table 3.1: Amount of newly added vulnerability test cases

nerabilities, but numerous classes of vulnerabilities were found absent, such as the case for SQL Injection exploited using obfuscation techniques as well as blind SQL injection test cases. Cross-Site Scripting is the most covered category in terms of amount of test cases. However, many attack vectors made popular in recent years were not included. At last, entire classes such as OS command injection and XML External Entities were absent. After a careful study phase, vulnerability classes were selected and test cases to be added, all of which have been deemed necessary in order to make WAVSEP an up to date benchmark.

Many of the works examined have provided numerous ideas in order to make this work of enlargement both robust and efficient. Among the most recent ones, Munoz et al. in [64] provide a thorough analysis of vulnerable-by-design web applications and list the desired requirements for a vulnerable application that serves as a benchmarking platform:

1. High extensibility. Adding new vulnerabilities should be easy. The source code of the application should be available, and it should be easy to add new features without making drastic changes to the source code, database and technologies used.

2. The application should use frameworks, databases and current languages, therefore be developed using current technologies and functionality.

3. The application should be of general use, having different types of vulnerabilities and not be focused on just a few selected ones.

4. Vulnerabilities should be well documented, with examples of exploitation, as well as a description of their application range.

To fulfill the above mentioned requirements, a work of technologies refinement has been performed as well. In fact, an update to the current version of Java Server Pages (v11) as well as MySQL database (v8) was made. Moreover, the entire project has been made available in the form of container-based microservices using Docker and docker-compose. A distributed testbed made up by WAVSEP as well as many well known DAST tools in the form of Docker containers has been provided[3]. In this way, it becomes easier for researchers to conduct testing sessions for their own purposes. Container-based technologies, in fact, provide good portabilty among several operating systems as well as high extensibility (it is easy to add or remove a container in order to enlarge or modify the testbed).

[3] https://github.com/NS-unina/Reinforced-Wavsep

Chapter 4

A semi-automated platform for penetration testing based on reinforcement learning

Nowadays, web applications are among the main targets of attacks. This is due to the ever-increasing level of dynamicity (and interactivity) they entail, through a mix of approaches, leveraging both client-side and server-side programming techniques. This has justified the creation of a dedicated branch of Penetration Testing, typically referred to as WAPT (Web Application Penetration Testing).

Cross Site Scripting (XSS) is a specific attack to web applications. It leverages so-called injection techniques, whereby malicious scripts are injected into otherwise benign and trusted websites. Typically, attackers use a web application to send malicious code, generally in the form of a client-side script, to a different end user. The typical flow of execution of such attacks envisages that some data are fed as input to a web application through an untrusted source (e.g., a request carrying some externally-provided data) and subsequently included in a dynamically built response that is sent to a web user without being validated for malicious content.

Various types of XSS attacks do exist. Among them, Reflected attacks are those where the injected script is reflected back by the web server, either in an error message, or as a search result, or in any other form that includes at least a portion of the input sent to the server as part of the request. Reflected attacks are typically designed in such a way as to be delivered to victims via an alternative means, like, e.g., an e-mail message, or even within a website other than

the reflecting one. When a victim is tricked into clicking on a malicious link, they typically unintentionally submit an ad hoc crafted form, thus allowing the injected code to reach the vulnerable web site and get reflected back towards the victim's browser, which in turn executes the code because it came from what is deemed to be a trusted server.

Much of the work presented in this thesis deals with Reflected XSS, by proposing an automated approach to its discovery in web applications.

The main contributions are the following:

- refine a methodology to discover XSS vulnerabilities based on constructing an attack string by small increments. With the proposed step-by-step approach, each new increment is justified by the presence of specific parameters in the response of the web application. One such parameter is the context of reflection of the string;
- train an intelligent agent to produce attack strings in an *ad hoc* reinforcement learning environment;
- develop a human-in-the-loop tool, called Suggester, that sends attack strings to the web application and collects observations, following the recommendations of the trained agent;
- enrich the test coverage of the Web Application Vulnerability Scanner Evaluation Project (WAVSEP), with reference to reflected XSS.

4.1 Reflected XSS discovery

The developed testing methodology revolves around the idea that, in order to prove that the web application is affected by an XSS vulnerability, the goal is to make some reflected javascript code be executed by the browser. Penetration testers often attempt to get a Proof of Concept of the attack, by making the browser pop up an alert message, using the string "alert(1)".
After sending this string within an HTTP request, a vulnerable web application provides observations that penetration testers use to their advantage in order to build a proper attack payload, such as the popular *<script>alert(1);</script>* or *onerror="alert(1);"*.
Such a process is sequential in nature and has the purpose of unveiling, through a series of requests, parameters that guide the penetration tester towards the discovery of the vulnerability. Such parameters are the following:

- the so called “context of reflection", i.e., the location inside the HTML page where the injected parameter ended up;
- the fact that the current context of reflection does not allow for code execution;
- the requirement that the injected string must keep some well-formedness with respect to the context of reflection;
- the consideration that the application might implement filtering policies.

These parameters are leveraged by performing an injection and observing the related application’s response.
In the next sections, we will better describe the aforementioned parameters and the effect they have on the discovery methodology.

4.1.1 Reflection Context

Web applications often insert, inside the HTTP response, HTML code crafted based on user provided inputs that get inserted at specific positions within the Document Object Model (DOM). If such inputs are not sanitised correctly, an attacker is able to inject strings that enable javascript code execution in the client’s browser. The position within the DOM influences the way the attack string is crafted.

HTML context

user input flows inside the body of an existing HTML tag.

```
1        <HTML_TAG> user_input </HTML_TAG>
```

In this case the javascript code must be surrounded by a valid HTML tag, in order for it to be rendered by the browser. A popular choice among penetration testers is to surround the “alert(1);” code with the <script> tag, but there are other options.

HTML attribute field context

user input flows inside the opening of an HTML tag.

```
1        <HTML_TAG user_input tag_attribute>
```

In this context of reflection, it is quite common to have HTML events, that are "things" that happen to HTML elements. Javascript code can react whenever such events are triggered. Having the "alert(1);" code be surrounded by an HTML event such as "onclick", can make the browser render the injected code.

HTML attribute value context

user input flows inside the value field of an HTML tag attribute.

```
1        <HTML_TAG tag_attribute="user_input">
```

Depending on the type of attribute in which the input appears, the penetration tester can act in different ways:

- **HTML event attributes**: as stated before, such attributes are already contexts of javascript code execution, so the attack string does not need to be modified;
- **URL attributes**: HTML tag attributes that accept a URL as value. Using the keyword "javascript:" might lead to code execution;
- **Special URL attributes**: HTML tag attributes that accept URL values leading to security issues other than XSS are not taken into consideration;
- **Other attributes**: these are not execution contexts. Any other HTML tag attribute can be escaped in order to let the input code flow into a different reflection context. For specific information on escaping the current context we refer the reader to section 4.1.2.

HTML comment context

the user input flows inside an HTML comment area.

```
1    <!-- COMMENT_AREA  user_input COMMENT_AREA -->
```

Neither is this a code execution context, nor a new context can be injected to let it become one. In such cases the current context of reflection needs to be escaped.

Javascript context

user input flows inside a javascript code area. This is already an execution context, meaning that the "alert(1);" code might be enough to let the browser render it. Apart from the input falling directly inside a <**script**> tag, escaping actions are usually needed when the actual context of reflection is one of the following:

- *javascript function input parameter*;
- *javascript variable*;
- *javascript single line comment area*;
- *javascript multiple line comment area*;

CSS context

user input flows inside a CSS code portion of the HTML page, especially inside an expression property:

```
1  css_selector
2  {
3      property_name: expression(user_input);
4  }
```

or inside CSS properties that take a URL as input value:

```
1  css_selector
2  {
3      property_name: javascript:user_input
4  }
```

Code execution within CSS areas is disabled in modern browsers [65]. Still, such context is taken into account, considering that a target for penetration testers might also be a system not up to date in terms of security patches [65].

4.1.2 Context Escape

As stated in Section 2, as opposed to early approaches that focused exclusively on the analysis of the reflection context, we base our penetration testing methodology on more parameters, like, e.g., how to switch from one context to the other. Some contexts of reflection, in fact, do not allow for javascript code execution. In these cases, the penetration tester, after figuring out in

which context of reflection the input flowed, might need to escape it. This means switching to another context of execution where the injected code can be rendered by the browser. Techniques that allow to do so, involve the usage of sequences of characters that close the current context and open a new one. These are characters that are already inserted, of course, by the application.The penetration tester injects inputs that prove what happens if those characters are not correctly handled or sanitised on the server's side.
Every context of reflection that needs to be escaped is indeed characterised by some sequences of characters that identify the beginning and end of that context. Hence, the characters injected by the penetration tester aim to close the current context.
In the following sections, a categorisation of these characters, with respect to the various contexts of reflection, will be provided.

HTML context escape

whenever the input flows inside an HTML tag, in most cases there is no need for escaping. A new context of execution, by injecting the <**script**> tag, is enough to trigger the vulnerability. The only exceptions concern the reflection inside tags that do not allow for rendering of javascript code. An example is the <**textarea**> tag: strings inside this tag would be ignored by the javascript engine of the browser. The solution is to escape this context by simply closing the current tag: <**/textarea**>. This would allow the user input to flow inside a plain HTML context.

HTML attribute value context escape

for those attributes that are not execution contexts, escaping techniques must be employed. Usually, a penetration tester would inject characters that allow to close the current context and land into the closest context of reflection. However, if the application allows it, nothing prevents the tester from directly closing the entire HTML tag and ending up in a plain HTML context.
Under this category fall all those characters that allow to close the context of the value of a tag attribute:

- *double quote character*;
- *single quote character*.

Numerical attribute values are not surrounded by any character.

HTML comment context escape

to escape this context, a penetration tester can simply close the HTML comment, using the sequence of characters '**- ->**'. This will result in having the input string flow into a plain HTML context.

Javascript context escape

escaping a Javascript context becomes necessary when specific sinks prevent the injected code from being rendered by the browser.
In order:

- variable closing character (double quote or single quote) followed by a semicolon to escape a javascript variable context;
- a semicolon to escape a numerical javascript variable context;
- function closing characters (double quote or single quote followed by a closing parenthesis) followed by a semicolon to escape a javascript function input parameter context;
- single line or multiple line javascript comment to escape a javascript comment area context.

The non-mentioned contexts in this section are considered not necessary to be escaped.

4.1.3 Attack string well-formedness

After checking the reflection context and, where necessary, escaping it, the injected code might still not be executed by the browser. This might happen because the string injected is not properly rendered by the HTML engine of the browser. In some cases, in fact, some "fine tuning" on the attack string becomes necessary. This can be done by injecting other characters that fix the unveiled errors. Such characters depend both on the context of reflection and on the escaping strings.
When a new tag is injected in the HTML code (for example, a <**script**> tag needed to keep syntactic consistency), the tester typically injects the tag closure, after the "alert(1);" string. However, there are other cases where the characters to inject are not as straightforward:

- when the tag attribute value context is escaped and a new context is injected by inserting an HTML event, a new attribute opening character (double quote or single quote) needs to be inserted before the javascript code. This is because these characters are already in the HTML and, when we escaped the context of reflection, they have been pushed further. This causes a syntax error that we need to fix by performing a balancing of the attack string;

- when the javascript code context is escaped, the injected input is surrounded by javascript code already used by the application. In order to let the code be executed, a javascript comment character "**//**" needs to be used at the end of the attack string, to comment out whatever comes after;

- when the input is inside a javascript single line comment area, the only way to have the injected code be executed is to insert a couple of characters, carriage return and line feed. This will make our javascript code jump to the next line;

- when the javascript comment area is spread across multiple lines, the context is escaped. In these cases it is necessary to make the code jump to the next line by inserting carriage return and line feed characters. Moreover, escaping such a context automatically uncomments the code that was already present in the HTML page. In order to restore everything to the state before the injection, a new opening javascript comment (**/***) character can be inserted after the "alert(1);" code.

4.1.4 Filtering Policies

At each step the application might implement filtering techniques that strip or sanitise some of the characters described so far. This is not the best course of action, because such policies can be eventually defeated. The only recommendation which has proven to be effective against XSS attacks is to escape the inputs provided by the user. However, the penetration tester can notice the presence of such policies after sending the attack string to the application and realising it did not have the expected outcome.
A thorough characterisation of filtering techniques is out of the scope of this thesis. However, we do take into account some cases where the characters described are treated differently by the application. For example, when the penetration tester injects a double quote character but the attribute tag value

context is not escaped. In this case the next attempt would be to inject a single quote character.

4.2 Environment setup

In this section we will describe the design of an *ad hoc* reinforcement learning environment that enables training of an intelligent agent to produce attack strings, according to the guidelines outlined in section 4.1.

4.2.1 Background

In the standard reinforcement learning model, an agent interacts with an environment by performing actions. At discrete time steps, the agent receives as input the current state of the environment and chooses an action. The action changes the state of the environment: a value associated with the state transition is transferred back to the agent using a reinforcement signal, called "reward". The goal for the agent is to find a policy $\pi(s \mid a)$ that maps a state s to a probability distribution over actions a in order to maximise the long-run expected reward. Reinforcement learning agents, especially in the field of computer science, learn through trial-and-error, adjusting the policy based on the results of so called episodes of training, which are sequences of state-action pairs [66].
An environment is represented in the form of a Markov decision process and algorithms to learn optimal behaviours come from the field of dynamic programming [67].
Q-Learning is a type of model-free reinforcement learning, in the sense that it does not rely on a representation of the environment in the form of a mathematical model [68]. It allows to learn estimates for the optimal value of each action, as the expected sum of future discounted rewards. Given a policy π, an estimate of the optimal value of an action a in a state s is

$$Q_\pi \equiv E[R_1 + \gamma R_2 + ... \mid S_0 = s, A_0 = a, \pi]$$

where $\gamma \in [0, 1]$ is a discount factor that takes into account the trade off between immediate and future rewards. The optimal value is $Q_*(s, a) = \max_\pi Q_\pi(s, a)$. Selecting the highest valued action in each state, determines an optimal policy [69].
The purpose of the testing methodology defined in section 4.1, was to cut down to pieces the way a penetration tester behaves and simplify actions enough in

order to let an intelligent agent learn them. Training an intelligent agent to complete vulnerability detection tasks, requires modeling a novel state and action space.

4.2.2 State space

The state consists of an attack string. An attack string is subdivided into five sections, each of them having a direct reference to one of the steps in the detection methodology. Sorted from the beginning of the string on the left, to the end on the far right, the identified sections are listed below:

- **PreContext**: the section where the agent places the characters used to perform context escaping;
- **Context**: the section where the agent places keywords responsible for injecting a new context of reflection;
- **PreExploit**: a section that precedes the javascript code, in order to maintain well-formedness across the characters of the string;
- **ExploitCode**: the section where the agent places the javascript code to be executed (e.g., “alert(1);”);
- **PostContext**: a section that comes after the javascript code, in order to keep well-formedness across the characters of the string.

Fig. 4.1 is an example of the deconstruction of an attack string into the aforementioned five sections. This preprocessing operation is performed before the training for all the attack strings taken into consideration as part of the environment.

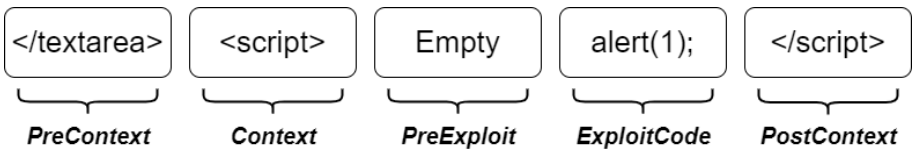

Figure 4.1: State structure

4.2.3 Action Space

An action consists in the act of modifying one and only one of the five sections. It can be considered as a function that accepts two input parameters: the *substring* to insert and the *section* where to insert it. To identify the substrings that fill each section of the state, a deconstruction of attack strings was performed. The "empty string" is considered among the list of substrings that modify the state. An action that places an empty string in a section of the state, signifies that the attack string did not need to be modified.
Thanks to this definition of the action space, we allow the agent to modify only one portion of the string at a time. In this way, we can correlate the feedback provided by the application with the action performed on the string. The number of substrings found during the preprocessing directly influences the state space dimensionality, that can be computed as

$$\prod_{n=1}^{5} S_n$$

where S is the number of substrings and n is the number of sections. This definition of the state and action space is prone to explosion. We tackle this issue in the evaluation section and show how it can be taken under control with the appropriate precautions.

4.2.4 WAVSEP

As thoroughly explained in Chapter 3, the Web Application Vulnerability Scanner Evaluation Project is a vulnerable web application designed to help assess the accuracy of automated vulnerability scanners. It covers multiple types of injection vulnerabilities. For each one of them, it encompasses several exploit techniques, providing also a technical explanation for the applied solution. Regarding Reflected XSS vulnerabilities, the different exploit methods are separated basing on the context of reflection. This approach adheres well with the environment definition proposed in this work, reason why WAVSEP's test cases were chosen as training data for the reinforcement learning agent. Although being an important open-source platform for the assessment of vulnerability scanners, the project is outdated. New test cases were introduced in order to keep the training data up to date with the current state of the art.
The newly introduced test cases cover the following contexts of reflection:

- HTML Attribute Name (1 test case);

- HTML Attribute Value (7 test cases);
- Javascript code (1 test case);
- Javascript function (2 test cases);
- CSS (1 test case);
- CSS comment (1 test case);

A number of 13 new test cases is added to the previous 24, for a final amount of 37.

4.2.5 Training

The first thing the agent has to do, in order to complete an XSS-reflected detection task, is to learn how to produce an attack string starting from the substrings and the sections identified. To fulfil this requirement, we noticed that a penetration tester performs different sequences of actions depending on the context of reflection of the string. This happens because the causes for XSS vulnerabilities are different and so are the attack strings needed to prove their existence. We decided to identify each attack string as the goal to reach for an agent during each training session. This results in a multiobjective environment: an agent is trained to reach different objectives, on the same environment. Table 4.1 shows some of the goals of the training sessions.
In section 4.3, we will discuss how the learned actions can be translated - thanks to the support of a human - into sequences of actions against the web application.

We apply standard Q-Learning to train the agent: the agent transitions states, performing actions. Executing an action in a state provides the agent with a reward. An ε-greedy exploration strategy was selected. The discount factor γ is set to 0.98, to allow the agent to strive for long-term high rewards. The learning rate α is set to 0.05.
Reward function. At this stage, we want the agent to:

- learn how to place a substring in the right section;
- learn how to perform a sequence of actions that lead to the construction of an attack string.

The described good actions will belong to subset A_1. In order to let the agent learn the best actions autonomously, we need to provide:

Objective	Final State
HTML	{"PreContext": "", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"}
HTML tag	{"PreContext": "</textarea>", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"}
HTML comment	{"PreContext": "–>", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"}
URL attribute	{"PreContext": "", "Context": "javascript:", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": ""}
tag attribute	{"PreContext": "¨", "Context": "onerror=", "PreExploit": "¨", "ExploitCode": "alert(1);", "PostContext": ""}
attribute filtering	{"PreContext": "'", "Context": "onerror=", "PreExploit": "'", "ExploitCode": "alert(1);", "PostContext": ""}
javascript variable single quote	{"PreContext": "';", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"}
javascript variable double quote	{"PreContext": "";", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"}
javascript numerical variable	{"PreContext": ";", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"}
javascript single line comment	{"PreContext": "", "Context": "", "PreExploit": "%0A%0D", "ExploitCode": "alert(1);", "PostContext": ""}
javascript multi line comment	{"PreContext": "*/", "Context": "", "PreExploit": "%0A%0D", "ExploitCode": "alert(1);", "PostContext": "/*"}

Table 4.1: Some objectives of training

- a constant, positive reward when the agent places a substring in the right section;

- a variable, positive reward, that increases as the string gets closer to the objective.

Because there are five sections of the attack string to fill, the ideal scenario is that the agent performs a sequence of five actions. Depending on the objective to reach, the attack string might find itself in a range from zero to five steps distant from the final one, with zero being the complete attack string and five being a completely wrong attack string.
In addition, we need to:

- discourage the agent from performing an action that places the wrong substring in a section, according to the attack string currently under construction. These actions will belong to the subset A_2.

- discourage the agent from distancing the string from the final one, by performing an action that invalidates a good action performed in one of the previous steps. These actions will belong to the subset A_3.

This results in providing an agent with:

- a constant, negative reward when a bad action is performed, but does not modify those sections that were already in their optimal state;

- a constant, negative reward when an action that makes the string regress one step back from the final one, is performed.

Given an action a, the piecewise reward function (y) is represented as follows:

$$\begin{cases} a \in A_1 \implies y = 10 + 2x & 1 \leq x \leq 5 \\ a \in A_2 \implies y = -\mu_1 & \mu_1 \gg 1 \\ a \in A_3 \implies y = -\mu_2 & \mu_2 < \mu_1 \end{cases}$$

where x is a variable that represents the number of sections already in their desired state.

4.2.6 Simulated Gym environment

The RL environment described in this section was implemented in a simulated form, using the OpenAI Gym[1] toolkit. Expert demonstrations were conducted on WAVSEP test cases, in order to record observations. For each objective of training, such observations were implemented in a simulated environment. In particular, each section of the attack string has an associated observation:

- **isEscaped**, associated with the PreContext section. If true tells the agent the escaping was correctly performed;
- **isContextOfExecution**, associated with the Context section. If true tells the agent the current context allows execution of javascript code;
- **isPreExploitFixed**, associated with the PreExploit section. If true tells the agent the attack string is well formed;
- **isExploitCodePresent**, associated with the ExploitCode section. If true tells the agent the javascript code is present in the string;
- **isPostContextFixed**, associated with the PostContext section. If true tells the agent the attack string is well formed;

In this environment, an agent modifies one section of the attack string at a time. At first, the agent selects substrings more randomly and accumulates rewards. Then, as ε decreases, the agent chooses correct actions that allow it to get closer and closer to the objective.

4.3 Agent orchestration through human interactions

In this section, we discuss the development of a tool called *Suggester*, that supports penetration testers in the discovery of Reflected XSS vulnerabilities.
Woolridge in [70] makes a difference between Intelligent Agents and Expert Systems on the basis of their autonomy: Intelligent Agents interact directly with an environment and get their information via sensors, whereas Expert Systems provide advice to users who act as middle men. However, the author states that “despite these differences, some expert systems, (particularly those that perform real-time control tasks), look very much like agents" [70]. Basing on this definition, our approach, at first sight, seems to fall under the category

[1] https://gym.openai.com/

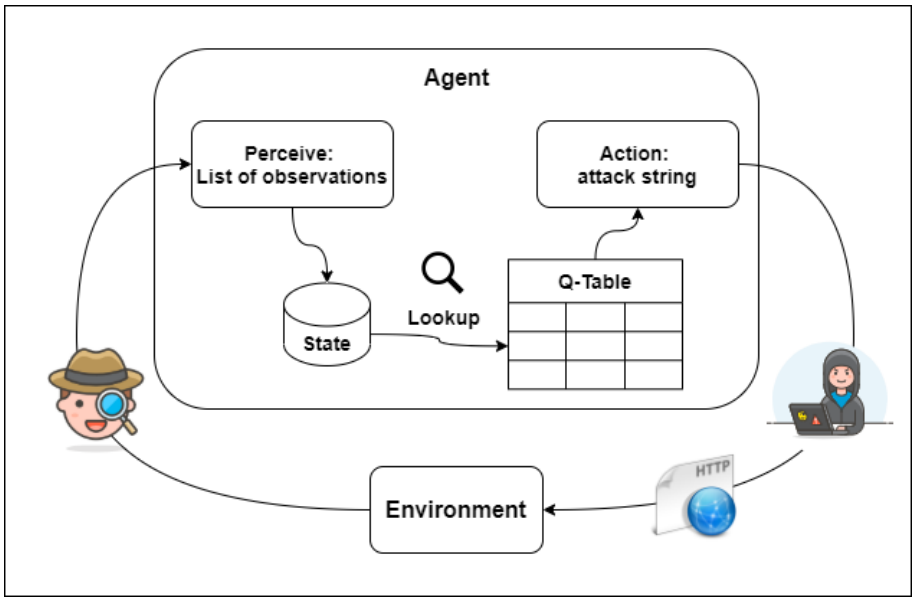

Figure 4.2: Intelligent Suggester architecture

of Expert Systems, having to rely on human interactions to collect observations from the real environment. However, the Suggester has been designed with the rationale of solving a live task, with suggestions put into practice by users who collect information that make the agent change its internal state. This is opposed to an Expert System, which is inherently disembodied and does not directly interact with an environment. Moreover, the architecture of a so called Intelligent Agent, well adheres to the Reinforcement Learning framework, where an agent explores an environment by directly interacting with it.

4.3.1 Architecture

Figure 4.2 shows the overall architecture of the Suggester. The agent recommends the first action, in the form of an attack string. Users translate the suggested action in an HTTP request sent to the vulnerable web application. Then, the agent provides a list of possible observations. Users inspect the web application's response and select the observation that matches the results seen in the response. The selected observation allows the agent to set the right attack objective. As discussed in section 4.2.5, the output of the training is

represented by Q-tables, one for each attack objective. Each Q-table encodes the agent's desired behaviour for the respective objective. The next action to suggest is selected by performing a lookup in the current Q-table.
In order to perform a lookup, the agent needs to know the following state information:

- the current attack string, which allows to select a row of the Q-table;
- the section of the attack string to be modified next.

The process that allows the agent to modify the right section of the string is described in Figure 4.3. At each time step, the agent keeps track of the section that is currently being modified. In fact, the name of each state corresponds to a section of the attack string, with exception of the initial state, where the string is still empty. The specific sequence of states is based on the methodology described in section 4.1. In fact, starting from an empty attack string, penetration testers would

1. inject an exploit code, to check if the input flows in a reflection context where javascript code is executed;
2. escape the current context of reflection, to let the input flow in a new reflection context;
3. inject a parameter that represents a new context of javascript execution;
4. complete the injection of a new context of javascript execution;
5. fix any syntax error that compromises the well-formedness of the overall attack string.

If filtering is detected, in any of the states, the Suggester remains in the current state and provides users with observations that allow to set a new objective. The described procedure is shared among the separate objectives.

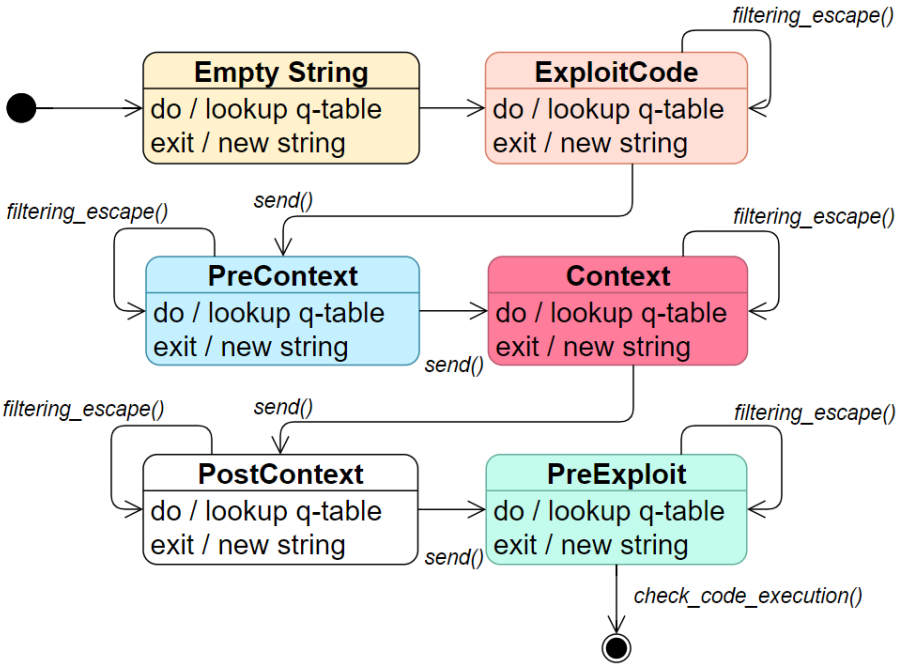

Figure 4.3: Agent orchestration policy state machine diagram

4.3.2 Bringing it all together

Figure 4.4 shows how the combined action between the Suggester and the user allows to build an attack payload that proves the existence of an XSS vulnerability. In this particular example, the reflection happens inside a "textarea" tag.

1. The string is empty and has not been sent to the application yet. The user tells the suggester there are no observations available;

2. The agent suggests to send the string "alert(1);". The user tells the suggester the reflection happened inside a <textarea> tag. In this reflection context, javascript code is ignored. Therefore, an escape action is required, in order to let the injected code flow into one that allows execution of malicious code;

3. The tool suggests to send the string "</textarea>alert(1);". In fact, closing the tag where the reflection happens allows to escape the current

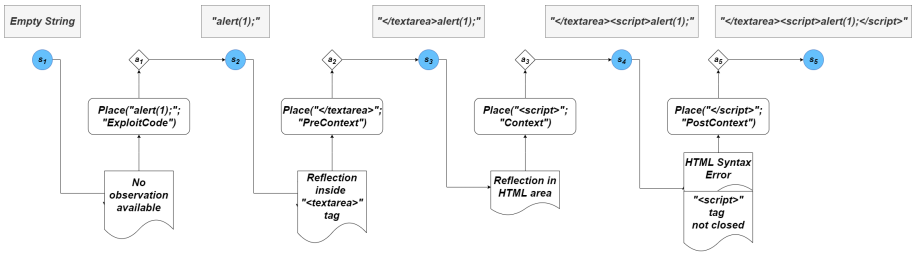

Figure 4.4: Discovery of reflected XSS with reflection inside an HTML tag.

context. After the string has been sent by the user to the application, the tool is notified that the context has been escaped and the current reflection happens in the HTML area;

4. Inside the HTML area, the "alert(1);" code is not executed, unless it is reflected in a tag where the javascript is not ignored, for example a "<script>" tag. In fact the tool suggests to send the string "</textarea><script>alert(1);" . The user inspects the response and realises now the injected code is inside a context of execution. However, the code is not executed yet, because the script tag needs to be closed. So, the user informs the suggester that a well-formedness error occurred;

5. The tool correctly suggests to close the script tag, by recommending the following string: "</textarea><script>alert(1);</script>". After sending such a string to the application, the user checks whether an alert popped up in the browser. This represents the final proof that the attack string worked and the vulnerability has been discovered.

4.4 Limitations and future research

The proposed agent was designed with a hybrid approach that relies on human intervention just to translate the recommendations of the agent into practical actions and observations.
However, future research could translate the human interactions into a fully automated intelligent agent. In particular, a module that sends the suggested attack strings inside HTTP requests and inspects the related HTTP responses, looking for the observations defined in Section 4.1, could provide a meaningful feedback and remove the human from the loop. The use of a headless browser would allow for the simulation of those javascript events triggered when the exploit code is executed, therefore ensuring to keep a low rate of false positives.
Moreover, the agent could also be trained to perform different sequences of actions, as opposed to the current implementation requiring 5 actions, one per each section of the attack string.
Since each action is associated with a corresponding HTTP request, such an improvement would affect the amount of requests performed by the agent, especially for those attack strings that require minimum modification to reach the objective. In order to accomplish this task, a way to optimize both parameters of an action (substring and subsection) needs to be taken into account during training.

Chapter 5

Towards a fully automated intelligent agent: the Observer module

This chapter describes the design and implementation of a module called "Observer", which is responsible for sending HTTP requests and analyze the response looking for observations. It is a fundamental feature for a modern reinforcement learning environment, because it allows agents to interact with it using a standard interface. It represents the first improvement to the architecture described in chapter 4, effectively enabling the realization of a fully automated intelligent agent.

5.1 General Architecture

Once the agent is trained, it is able to provide the best action to be performed in any given state. Since filtering policies and partial input sanitization could affect the choice of the best action to perform (i.e., sub-string to insert and the section where to insert it), it is necessary to have an observation module that recognizes the effect of the payload inside the reflection context. It is possible to see the Observation module as a sensor that provides the agent with real world information in order to perform an action. The observation module can return a positive feedback that means the payload properly worked, and a negative one that means the payload did not work as it was supposed to do. For example, after filling in the Pre-Context section in order to escape

the reflection context, the observation module checks if the Pre-Context string properly worked and the remaining part of the payload flows in a new reflection context. Based on the Observation feedback, the agent changes the state and selects the next action.
The 5.1 summarizes this concept.

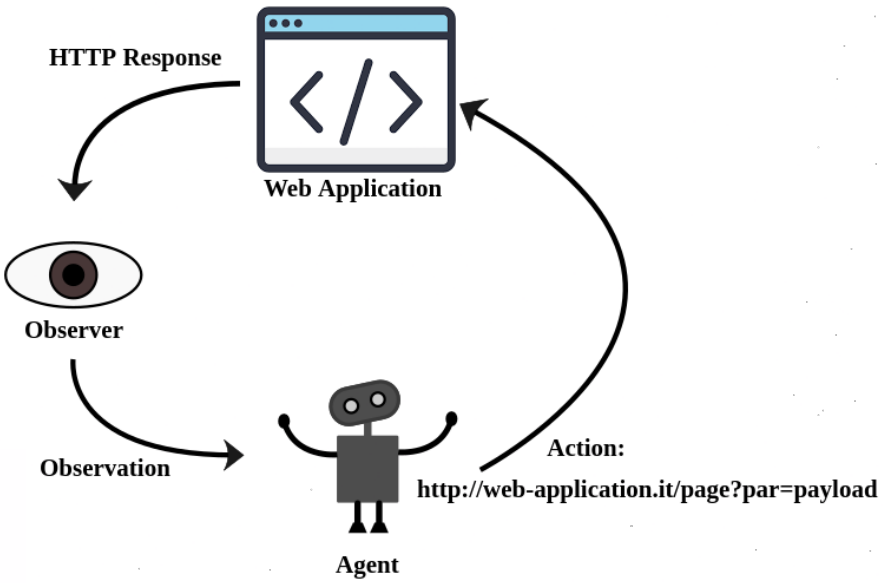

Figure 5.1: User input reflection

In order to understand whether the performed action was successful or not, the observer module provides one out of the three following feedbacks:

- **Green**: positive feedback, action successful (e.g., original reflection context closed, new JavaScript sink opened, no syntax errors hinder the JavaScript execution, JavaScript code properly executed or JavaScript sink properly closed).
- **Orange**: mixed feedback. The corresponding section of the state is in the right state but the remaining sections are not correct as of yet.
- **Red**: negative feedback, action not successful. The corresponding section is wrong and would never work in the current reflection context.

Because a state id made up by 5 separate sections, each section requires its own feedback signal. This structure represents an Observation. It is what the agent expects to receive from the observer after every HTTP request is sent to the web server under test.
The next paragraphs describe the Observer module design in detail.

5.2 Observer Interface

The main goal of the Observer module is to provide a feedback to the agent. That feedback reports to the agent if its action had an impact on the web application and if this impact can lead to the discovery of an XSS vulnerability. In order to produce the observation, the Observer module offers the "observe" function. The function needs two parameters from the environment:

- **HTTP response**: The HTTP response generated from the web application to the agent's request. This is provided to the Observer as a string;
- **Payload**: This is the state; that is the payload sent from the agent to the web application. This must be provided to the Observer as a JSON formatted string. The string is a JSON object with five fields, one for each sub-section. An example is shown in fig. 5.2.

```
payload = '{"PreContext": "", "Context": "onerror=", "PreExploit": "",
           "ExploitCode": "alert(1)", "PostContext": ""}'
```

Figure 5.2: JSON payload string

The Observer interface will provide the following function:

ReflectionFeedback[ ] observe(String response, String payload);

Calling this function, the agent is able to check whether the changed section is in a good state or not. If it worked, the agent can save the sub-section string in the state and can move forward to the next section. Otherwise, if the sub-section string did not work, the agent remains in the current state and performs a new action until a positive feedback is returned. After all the sections are in a good state, the Observer looks for a proof-of-work for the discovery of the vulnerability. This is possible, thanks to the "check_code_execution" function. The function needs two parameters from the environment:

- **URL**: The URL of the vulnerable page;

- **Payload**: The attack vector. This must be provided to the Observer as a JSON formatted string. The string is a JSON object with five fields, one per each section. An example is shown in fig. 5.2.

The Observer interface will provide the following function:

Boolean[] check_code_execution(String url, String payload);

After calling this function, the agent receives an array of two Boolean values:

- The first Boolean value is set to '1' if the payload worked on a Firefox based browser; '0' otherwise;

- The second Boolean value is set to 1 if the payload worked on a Chromium based browser; '0' otherwise.

Figures 5.3 and 5.4 show the complete Observer interface.

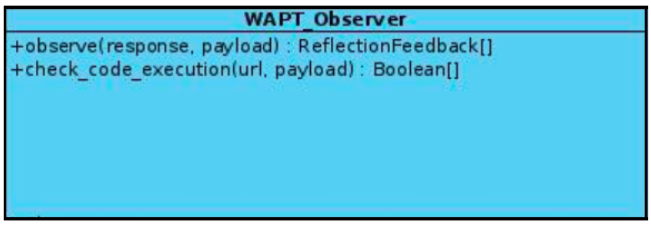

Figure 5.3: Observer class

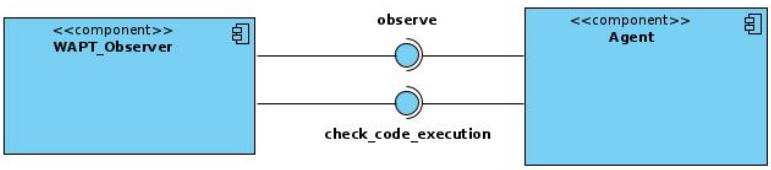

Figure 5.4: Observer interface

5.3 Observer Domain Model

According to the Software engineering principle of loose coupling and high cohesion, the Observer module is devided in sub-modules. Since the Observation is composed of five feedbacks, it comes natural to divide the Observer module into five sub-modules, each one responsible for the corresponding payload section. Such five modules rely on a separate component that is able to capture the reflection context of a string inside the HTML response. At last, a dedicated component is responsible for the code execution checking. The Observer Domain Model is shown in fig. 5.5.

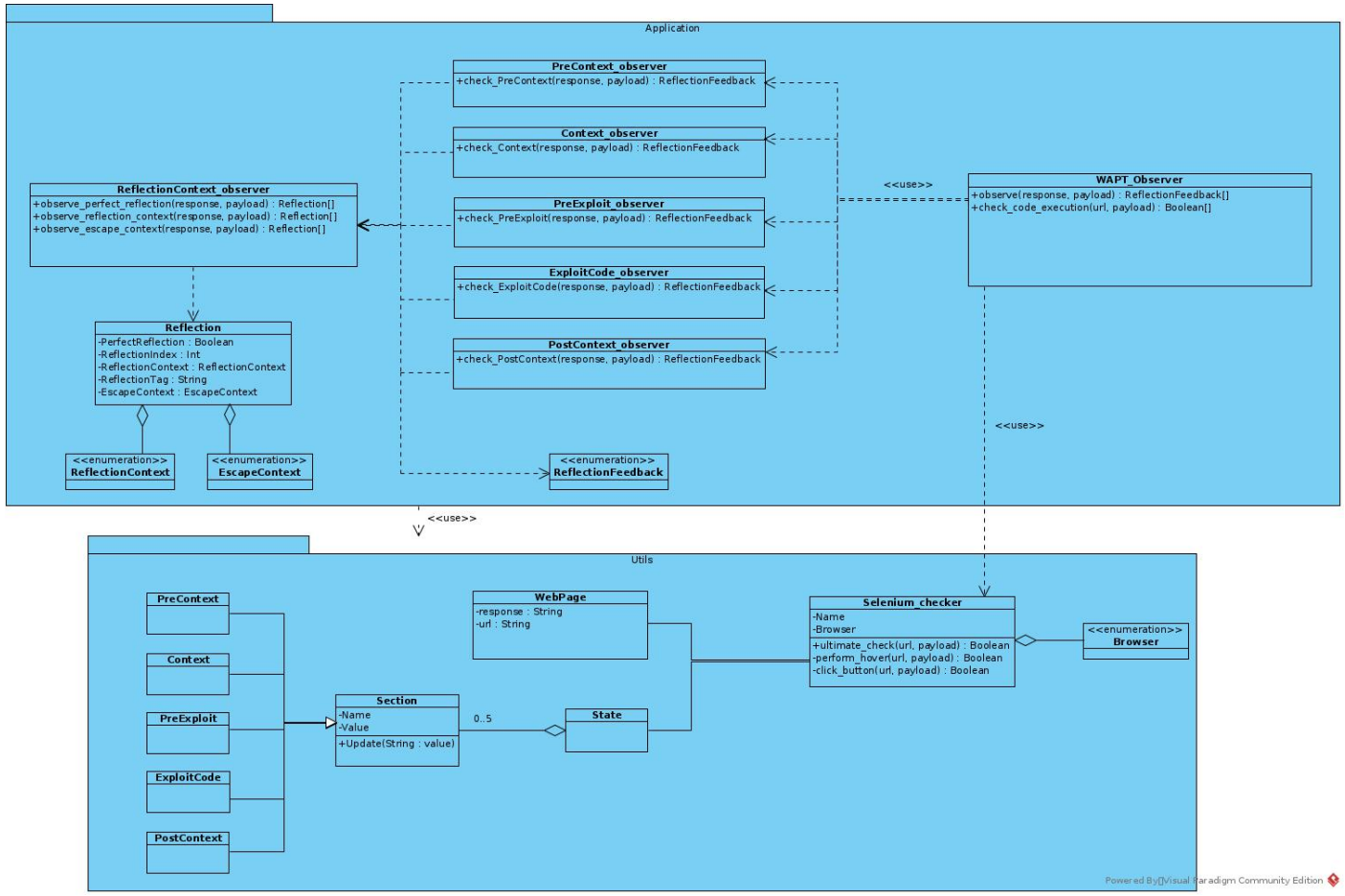

Figure 5.5: Observer Domain Model

Below, in table 5.1 there is a brief description of the classes:

Class	Description
WAPT_Observer	Class that exposes the methods to allow the Agent to obtain a feedback from the environment.
PreContext_observer	Class that checks if the PreContext string closed the current reflection context, if needed.
Context_observer	Class that checks if the Context string opened a new reflection context.
PreExploit_observer	Class that checks if the reflected string caused a syntax error in the reflection context.
ExploitCode_observer	Class that checks if the ExploitCode string ended in a JavaScript sink context.
PostContext_observer	Class that checks if the JavaScript sink used is properly closed.
ReflectionContext_observer	Class that looks for string reflection in the HTTP response and returns a list of Reflection objects.
Reflection	Class that models a single Reflection.
Selenium_checker	Class that looks for a proof-of-work. It checks if the payload actually opened a pop-up window in the browser.

Table 5.1: Observer Domain Model classes

5.4 Observer Execution Flow

Once we divided our Observer module in sub-modules, it is useful to explain how sub-modules cooperate in order to achieve the goal that underlies the Observer module interface. In this section, the following functionality is further explained:

- create an observation according to the guidelines described in the General Architecture section;
- test the final payload string and create an XSS proof-of-work.

Two design level sequence diagrams are provied, one for each main feature exposed in the Observer module's interface.

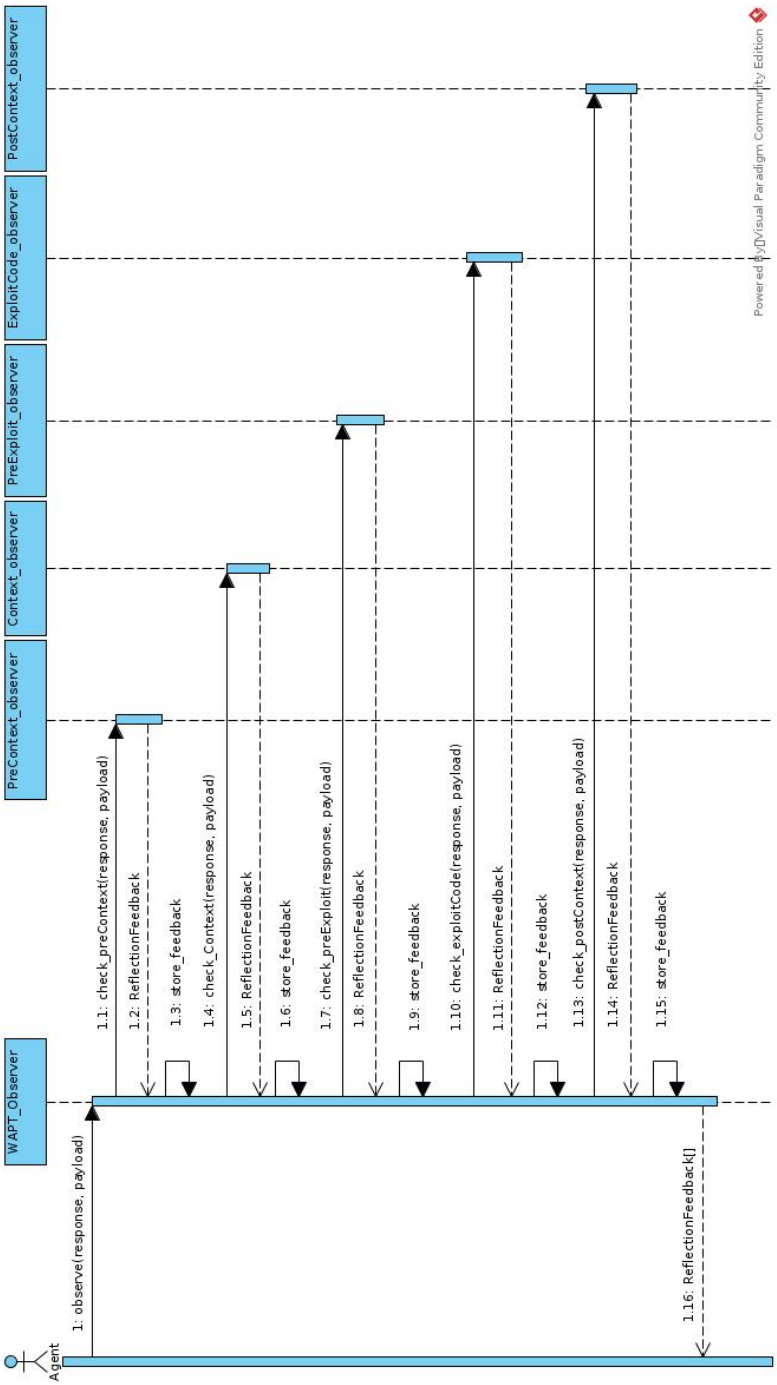

Figure 5.6: observe design sequence

Fig. 5.6 shows the high level sequence diagram of the “observe” function. As we already said, there is a dedicated sub-module for each payload section to check. The ‘observ” function stores a data structure containing the feedback provided by the five sub-modules. This data structure is the final observation returned to the agent.

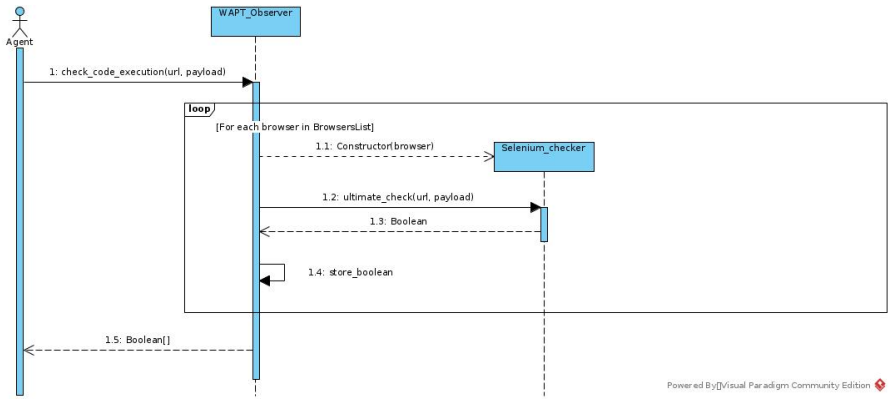

Figure 5.7: check_code_execution design sequence

Fig. 5.7 shows the high level sequence diagram of the “check_code_execution” function. This function instantiates a “Selenium_checker” object for each supported browser. The “ultimate_check” function of the “Selenium_checker” object is called in order to check whether the browser actually opened a pop-up window.

5.5 Design Decisions

In this section the decisions undertaken during the design of the Observation module as well as the motivation behind them are summarized.

- *Test Driven Development*: the module is developed using the Test Driven Development approach.

- *Training Set*: a custom version of WAVSEP benchmark as training set for the intelligent agent is used. The goal is to cover all of the reflection contexts.

- *Number of reflections*: One user input can be reflected in multiple points of the web page and so in more than one reflection context. The module takes into account all reflections of the user input. Therefore, it returns a positive feedback if the payload worked for at least one reflection;
- *The use of a headless browser*: The reflection of user inputs is a necessary yet not sufficient condition for the XSS vulnerability. Instead of looking only for the reflected user input in a JavaScript sink, the observer is able to supply a real proof-of-concept to the Agent by using a headless browser. This is handled by the "check_code_execution" function by instantiating an object that, using Chrome or Firefox drivers, looks for a pop-up window opened by the attack payload.

5.6 Implementation

The Observer module is implemented using python 3.6. The use of this programming language allowed to adopt methodologies like Test Driven Development and Continuous Integration. It is not mandatory to use python for Test Driven Development and Continuous Integration; however, Python offers an already mature development platform that allows to:

1. create virtual environments to manage all module's dependencies. We used Python venv.
2. run tests locally before committing to the mainline. For automated tests we used python pytest.

5.7 Class diagram

Figure 5.8 shows the implementation level class diagram. In such a diagram, the relations among classes remain the same as those of the design level. The same is true for the interfaces. In particular, new private functions are added to "ReflectionContext_observer", one responsible for each supported reflection context. The supported reflection contexts are explicitly indicated in their enumeration class, as well as the relative escape contexts techniques implemented.

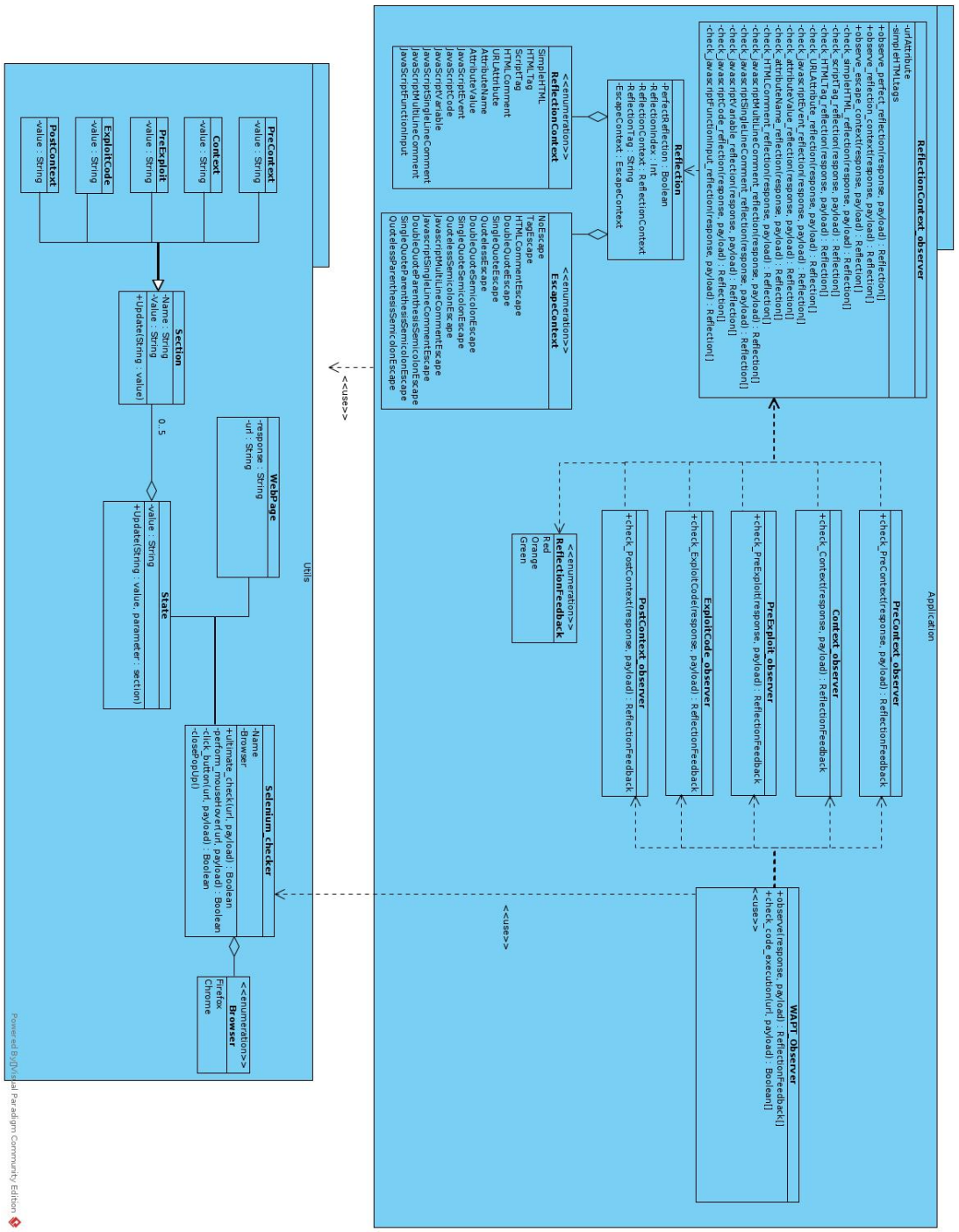

Figure 5.8: Observer Implementation Model

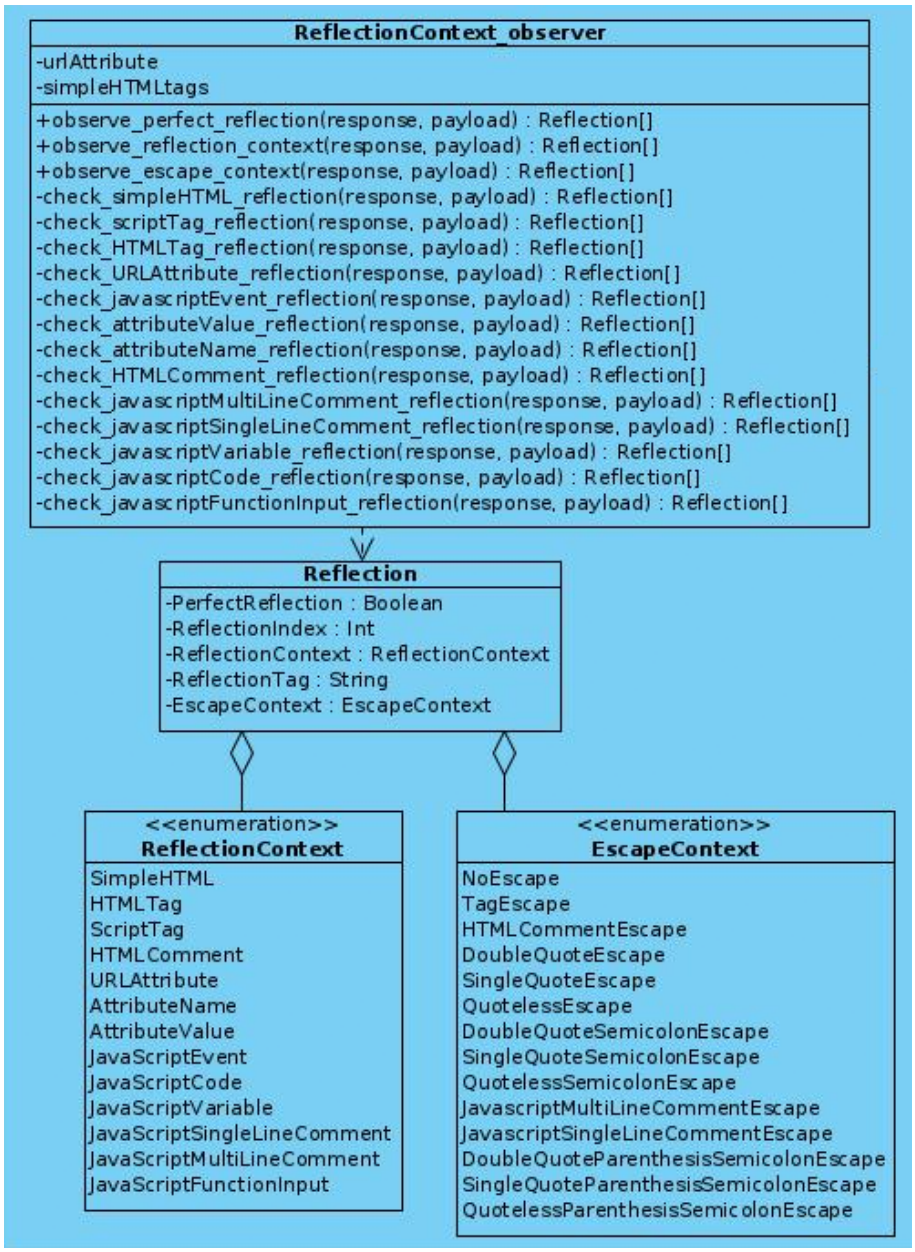

Figure 5.9: Reflection contexts in detail

5.8 ReflectionContext_observer class

The main class, upon which the entire module depends, is the ReflectionContext_observer class. This class is used by other classes to detect if a section in the payload was correctly filled. The class exposes three public functions:

- *observe_perfect_reflection(response, payload)*;
- *observe_reflection_context(response, payload)*;
- *observe_escape_context(response, payload)*;

They all return a list of Reflection objects, but they carry different responses.
The "observe_perfect_reflection" function fills out a Reflection object for each user input reflection found inside the web page response. Such objects signal the presence or absence of reflection and the reflection index. The "observe_reflection_context" function performs the same actions as the previous function and it also reports the reflection context of the reflected user input.
The "observe_escape_context" function performs the same actions as the previous function and it also reports the escape context technique to use.
The main feature of the ReflectionContext_observer class is the ability to find the user inputs reflected in the web page response and also to figure out the reflection contexts.
In order to find reflection contexts, we used the XPath query language. This language gives us the ability to find out not only the HTML node in which the user input is reflected, but also its position inside the node. Table 5.2 shows the XPath query used for basic reflection contexts:

Reflection Context	**XPath query**
Simple HTML	*xpath="//*[contains(text(), '"+input+"')]"*
HTML tag	*xpath="//*[contains(text(), '"+input+"')]"*
HTML Attribute Name	*xpath="//*"*
HTML Attribute Value	*xpath="//*[@*[contains(.,'"+input+"')]]"*

HTML Comments	*xpath="//*[comment()[contains(.,'"+input+"')]]"*
JavaScript	Combination of XPaths
CSS Context	Combination of XPaths

Table 5.2: XPath queries

Such XPath queries are used to retrieve the reflection context. Further analysis is needed to retrieve the specific reflection context. Next sections will explain this aspect.
In order to use the XPath queries, the “lxm” library was used. This library gives us the ability to construct an HTML tree data structure from the web page response. Once we have the tree data structure of the HTML page, we can perform the XPath query on the HTML tree. The use of an XPath query is shown in the following code snippet.

```
from lxml import etree
from lxml import html

def check_generic_reflection(self, response, payload):
  answer = []
  tree = html.fromstring(response)
  xpath = "XPath_query_from_table"
  search = tree.xpath(xpath)
  for element in search:
    if condition:
      r = Reflection()
      r.set_perfect_reflected(True)
      r.set_reflection_tag(element.tag)
      r.set_reflection_context(ReflectionContext.Generic)
      answer.append(r)
  return answer
```

Listing 5.1: Detection of generic reflection

In the snippet 5.1 the “ReflectionContext.Generic” and the “if condition” are just examples since they are different for every reflection context. Let us now take a look at the sequence diagrams of the private "ReflectionContext_observer" functions.
In Fig. 5.10, Fig. 5.11, Fig. 5.12 and Fig. 5.13 we can see that the logic to

retrieve, respectively, a SimpleHTML, HTMLTag, ScriptTag and a CSS reflection context is quite similar:

1. First use the lxml library to obtain a tree-like representation of the HTML page;

2. Use the respective XPath query according to the context we are looking for;

3. For each element returned by the lxml library, check if the html tag where the user input is reflected is respectively a “body” tag, a normal tag, a “script” tag or a “style" tag;

4. If the element meets the condition, then a new Reflection object is created and added to the answer list.

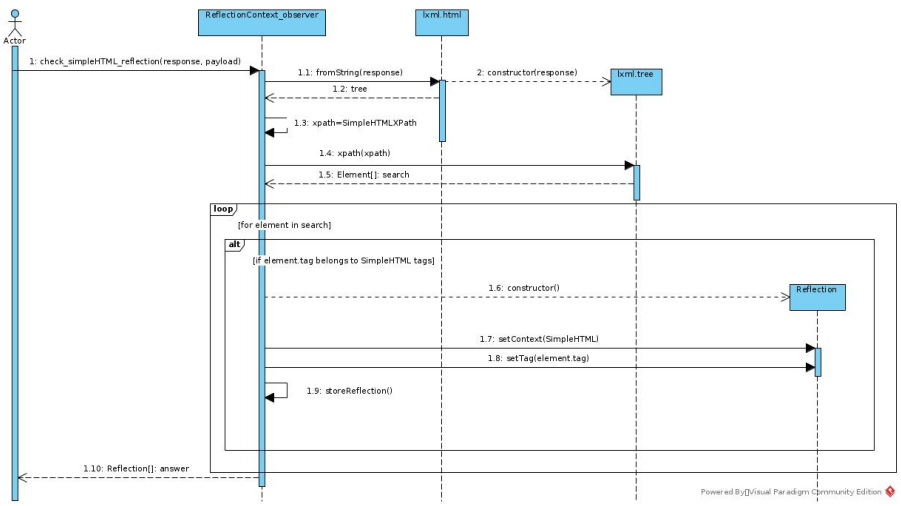

Figure 5.10: check_simpleHTML sequence diagram

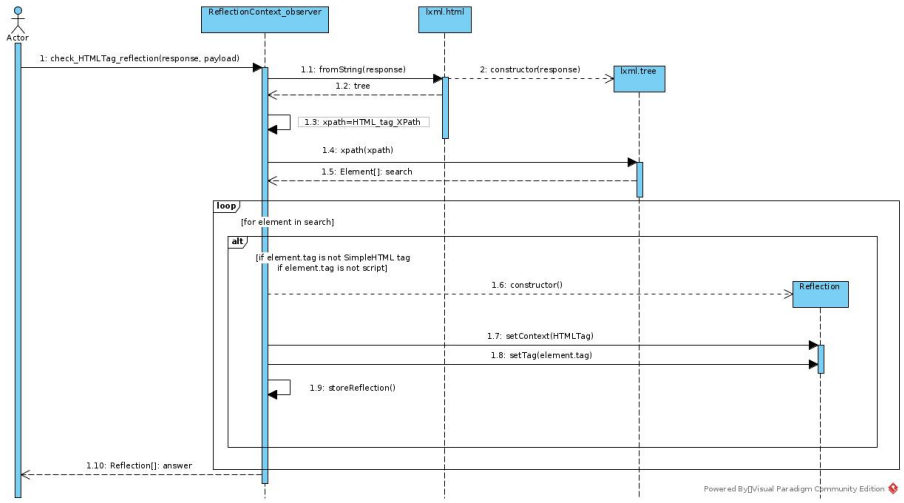

Figure 5.11: check_HTMLTag sequence diagram

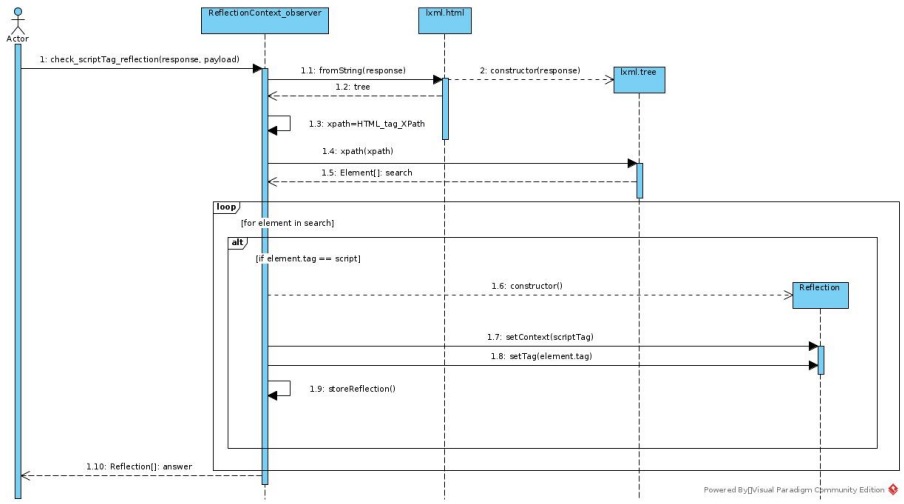

Figure 5.12: check_scriptTag sequence diagram

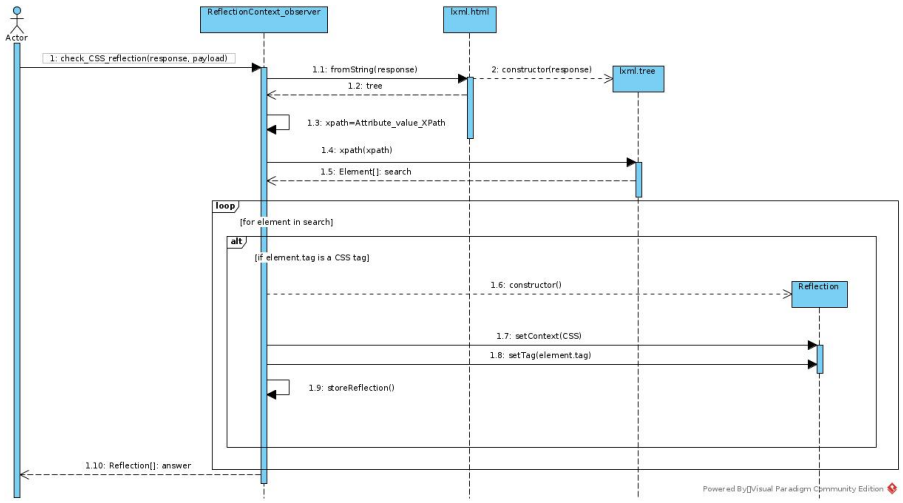

Figure 5.13: check_CSS sequence diagram

In Fig. 5.14 the sequence diagram shows how to retrieve the HTMLComment reflection context. In this case the XPath query is accurate enough, so that an additional check over the Element list returned by the lxml library is unnecessary. For each element of the list a new Reflection object can be created.

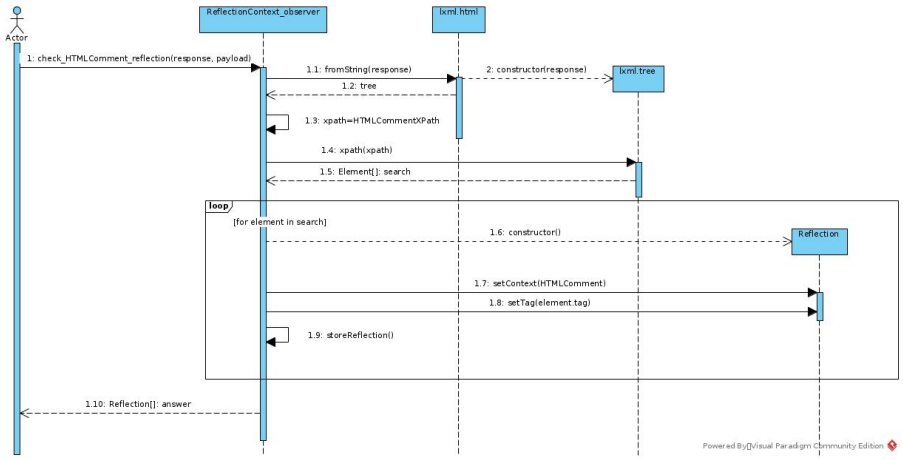

Figure 5.14: check_HTMLComment sequence diagram

In Fig. 5.15 and Fig. 5.16 a different logic for attribute name and attribute value contexts can be noticed. In such contexts, the reflection is not inside the node text, but rather inside the node specification. In particular, the reflection is respectively in the name or in the value of the node attribute. Since each element returned by the lxml library is actually an HTML node, one more "for" loop is needed in order to inspect all the attributes of the node.

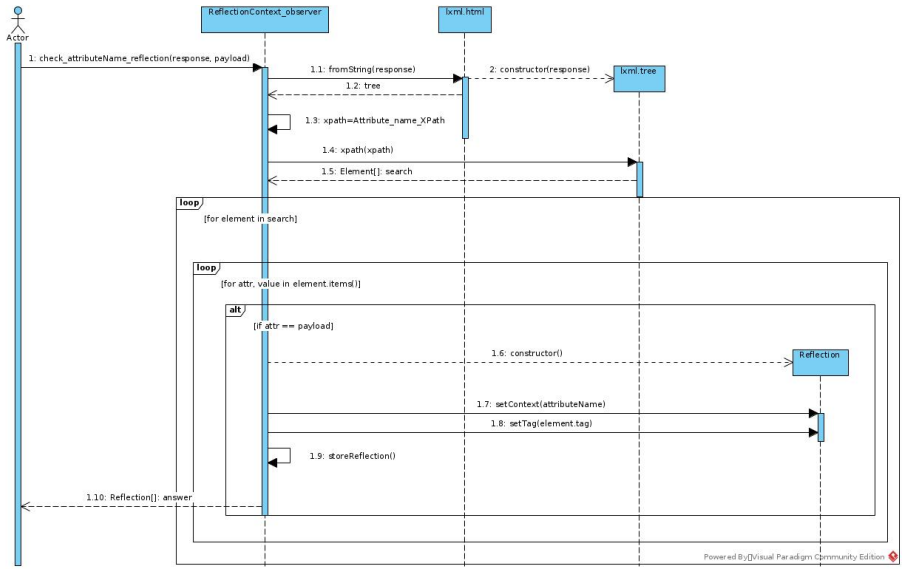

Figure 5.15: check_attributeName sequence diagram

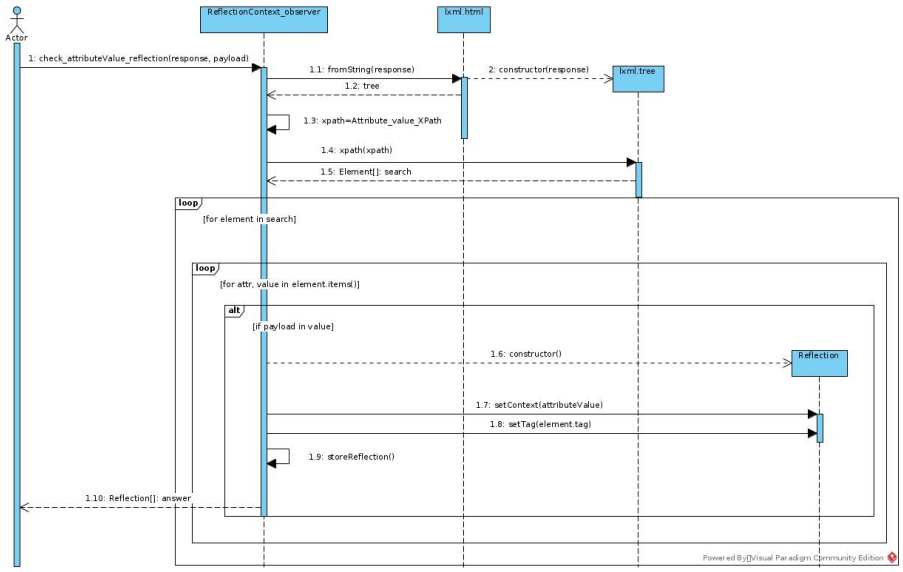

Figure 5.16: check_attributeValue sequence diagram

In fig.5.17 we can see how to retrieve a JavaScript context. A JavaScript context can be opened in two ways: through either a “script” tag or an event handler of any HTML tag. This double check is visible in the “check_javascript” sequence diagram.

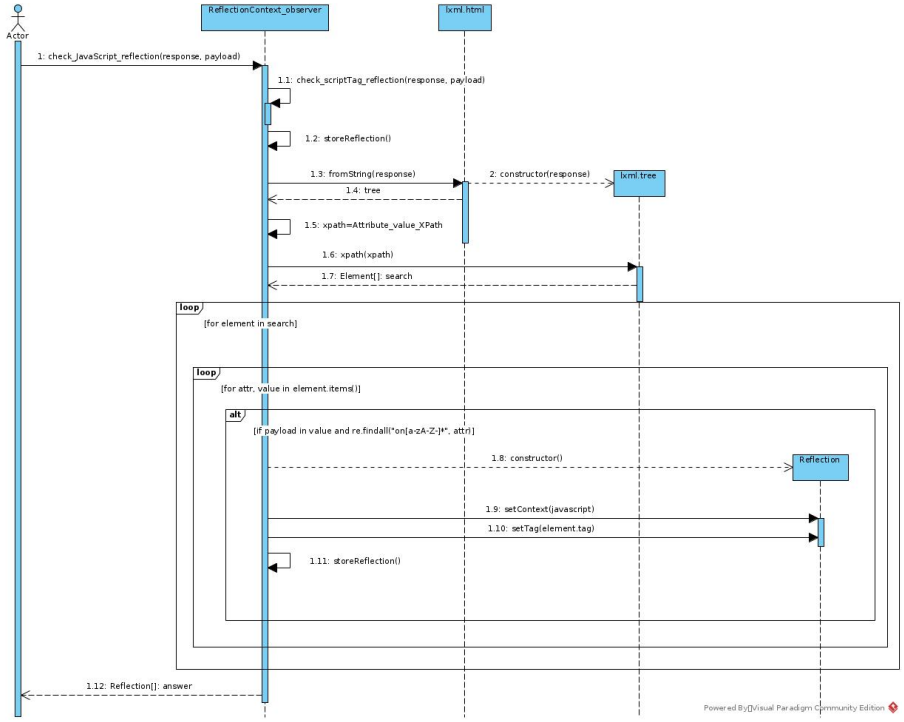

Figure 5.17: check_javascript sequence diagram

Once the reflection context is identified, Regular Expressions (RegEx) allow to find the right pattern that allows to escape or to open an HTML context inside the payload sections.

5.9 Selenium Checker

A dedicated class is used to provide proof of the effective execution of the final attack payload. In order to achieve such a goal, a Headless browser is needed. Among the available headless browsers, Selenium is the preferred choice. In fact, it allows to emulate some human interactions with the browser (e.g., mouse click, onmouseover, onresize, etc.). This feature allows to test all of the available payloads and reduce the false negatives rate. Another feature we appreciated is the opportunity to read the text inside an alert message. This lets us reduce the possibility to fall in a false positive result.

Since Selenium headless browser is used to perform such actions, the class is called “SeleniumChecker”. This class implements the “ultimate_check” function that is called by the “check_code_execution” function of the Observer interface. The “ultimate_check” function receives as input a URL and a payload to test. For each supported browser (e.g., Chrome and Firefox), the function performs an HTTP request to the web page. Before looking for a pop-up window, the function performs a mouse click on each button and a mouse-over action over each HTML element with an “onmouseover” attribute. At this point, if the payload worked, an alert pop-up should be visible. Figure 5.18 shows the sequence diagram of the "ultimate_check" function.

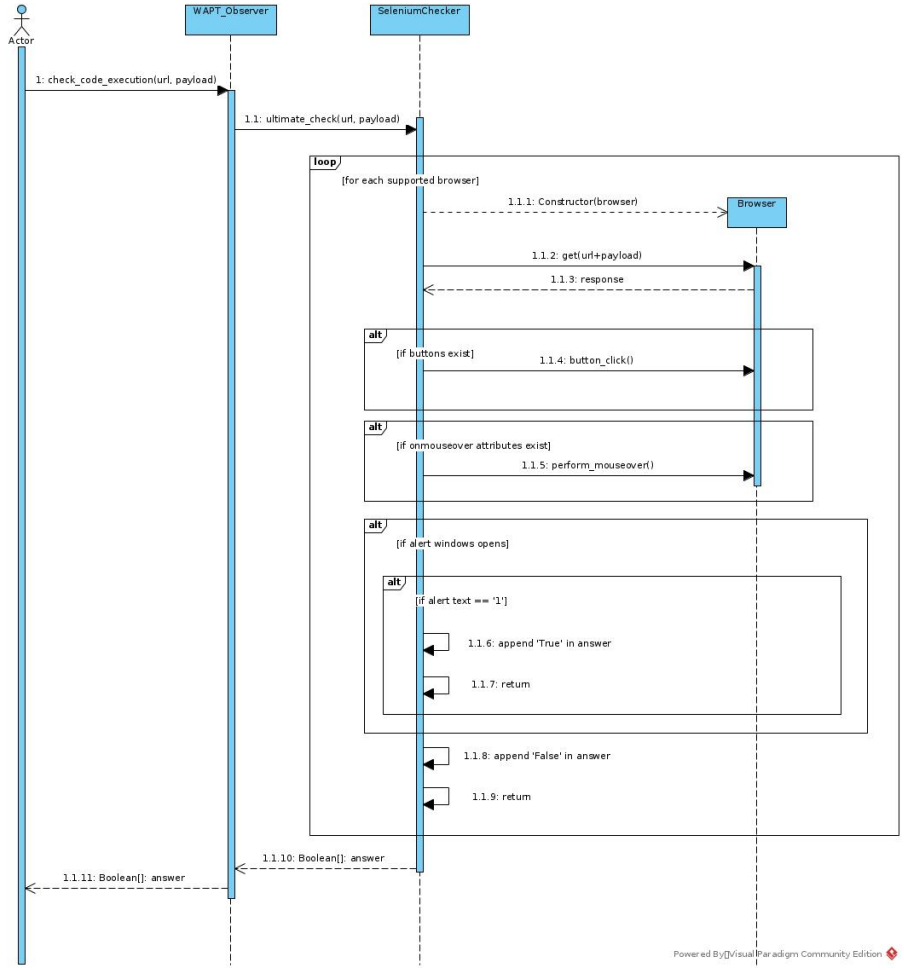

Figure 5.18: ultimate_check sequence diagram

5.10 Deploy

A container-based testbed, using Docker, to test the automated XSS scanner against a benchmark platform was engineered. The advantages of using Docker containers instead of virtual machines are the following:

- a more flexible environment;
- a lighter environment with a faster boot-up phase;
- high maintainability.

In particular, four docker containers are defined:

- one container for the Observer module;
- one container for the target web application;
- one container for a Chrome browser;
- one container for a Firefox browser.

The micro-services architecture we wanted to produce is shown in fig. 5.19.

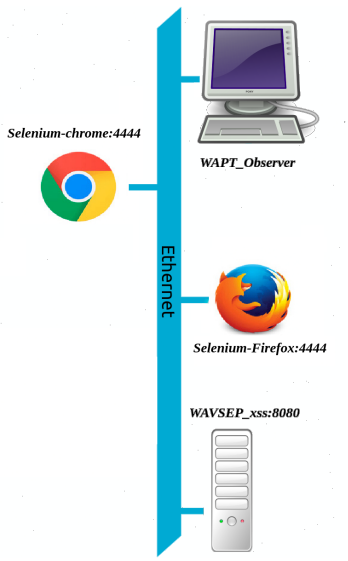

Figure 5.19: micro-services architecture

While the containers with Chrome and Firefox browsers and related Selenium drivers are available, a container for the Observer was defined, as well as one for the target web application, using the Dockerfile syntax. The following code snippet is the Dockerfile used to build our custom WAVSEP container.

```
FROM ubuntu:14.04

# Update Ubuntu
RUN apt-get update
RUN apt-get -y upgrade

# Install Oracle Java
RUN apt-get -y install openjdk-7-jre

# Install wget
RUN apt-get -y install wget

# Install tomcat
RUN mkdir /opt/tomcat/
WORKDIR /opt/tomcat
RUN wget https://mirror.nohup.it/apache/tomcat/tomcat-8/v8
    .5.54/bin/apache-tomcat-8.5.54.tar.gz
RUN tar xvfz apache-tomcat-8.5.54.tar.gz
RUN mv apache-tomcat-8.5.54/* /opt/tomcat/.

# Download WAVSEP
WORKDIR /opt/tomcat/webapps
COPY ./wavsep /opt/tomcat/webapps/wavsep/

# We connect to this application on port 8080
EXPOSE 8080

# Start Tomcat
CMD ["/opt/tomcat/bin/catalina.sh", "run"]
```

Listing 5.2: WAVSEP Dockerfile

A similar process was used to create the container for our tool.

```
FROM python:3

COPY ./WAPT_observer/ /usr/src/app
WORKDIR /usr/src/app

RUN pip3 install -r requirements.txt

CMD python3 wavsep_test.py
```

Listing 5.3: WAPT_Observer Dockerfile

In order to define the architecture, configure the application's service dependencies and run the entire architecture, docker-compose was used. docker-compose is a tool for defining and running multi-container Docker applications. Once the containers configuration is defined, with a single command it is possible to create and start the services.
The following code snippet describes the docker-compose architecture.

```
# docker-compse.yml
version: '3'

services:
  selenium-firefox:
    image: selenium/standalone-firefox
    expose:
      - 4444
    logging:
      driver: "none"

  selenium-chrome:
    image: selenium/standalone-chrome
    expose:
      - 4444
    logging:
      driver: "none"

  wavsep-xss:
    image: rznanni/wavsep_xss
    expose:
      - 8080
    logging:
      driver: "none"

  observer:
    image: rznanni/wapt_observer_test
    depends_on:
      - selenium-firefox
      - selenium-chrome
      - wavsep-xss
```

Listing 5.4: Docker Compose architecture

Fig.5.20 shows a portion of the docker-compose output.

```
giovanni@lenovo-X01:~/Documents/tesi/my_observer/test_docker$ sudo docker-compose up
Starting test_docker_selenium-chrome_1  ... done
Starting test_docker_selenium-firefox_1 ... done
Starting test_docker_wavsep-xss_1       ... done
Starting test_docker_observer_1         ... done
Attaching to test_docker_selenium-chrome_1, test_docker_selenium-firefox_1, test_docker_wavsep-xss_1, test_docker_observer_1
selenium-chrome_1   | WARNING: no logs are available with the 'none' log driver
selenium-firefox_1  | WARNING: no logs are available with the 'none' log driver
wavsep-xss_1        | WARNING: no logs are available with the 'none' log driver
observer_1          |
observer_1          | URL:             http://wavsep-xss:8080/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/Case01-Tag2HtmlPageScope.jsp?userinput=
observer_1          | payload:         <script>alert(1);</script>             works on FIREFOX
observer_1          | payload:         <script>alert(1);</script>             works on CHROME
observer_1          |
observer_1          | URL:             http://wavsep-xss:8080/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/Case01-Tag2HtmlPageScope.jsp?userinput=
observer_1          | payload:         <img src=a onerror=alert(1);>          works on FIREFOX
observer_1          | payload:         <img src=a onerror=alert(1);>          works on CHROME
observer_1          |
observer_1          | URL:             http://wavsep-xss:8080/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/Case02-Tag2TagScope.jsp?userinput=
observer_1          | payload:         </textarea><script>alert(1);</script>          works on FIREFOX
observer_1          | payload:         </textarea><script>alert(1);</script>          works on CHROME
observer_1          |
observer_1          | URL:             http://wavsep-xss:8080/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/Case03-Tag2TagStructure.jsp?userinput=
observer_1          | payload:         "><script>alert(1);</script>           works on FIREFOX
observer_1          | payload:         "><script>alert(1);</script>           works on CHROME
observer_1          |
observer_1          | URL:             http://wavsep-xss:8080/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/Case04-Tag2HtmlComment.jsp?userinput=
observer_1          | payload:         --><script>alert(1);</script>          works on FIREFOX
observer_1          | payload:         --><script>alert(1);</script>          works on CHROME
observer_1          |
observer_1          | URL:             http://wavsep-xss:8080/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/Case05-Tag2Frameset.jsp?userinput=
observer_1          | payload:         javascript:alert(1);           works on FIREFOX
observer_1          | payload:         javascript:alert(1);           works on CHROME
observer_1          |
observer_1          | URL:             http://wavsep-xss:8080/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/Case05-Tag2Frameset.jsp?userinput=
observer_1          | payload:         "onMouseover="alert(1);        works on FIREFOX
observer_1          |
observer_1          | URL:             http://wavsep-xss:8080/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/Case06-Event2TagScope.jsp?userinput=
observer_1          | payload:         "onerror="alert(1);            works on FIREFOX
observer_1          | payload:         "onerror="alert(1);            works on CHROME
observer_1          |
```

Figure 5.20: Docker Compose output

Chapter 6

Fully automated reinforcement learning agent

This chapter describes the implementation of a reinforcement learning environment for the discovery of Cross-Site Scripting vulnerabilities. It also shows how to integrate the Observer module described in Chapter 5 with the Suggester module described in Chapter 4 in order to create a fully automated intelligent agent. At first, the design will be presented, then the low-level implementation details will be shown. Everything will be documented using the UML style of code documentation, both from a static point of view, using Class Diagrams, as well as by showing the execution flow of the main functions through Sequence Diagrams. Finally, a summary of the tests performed will be shown.

The presented methodology is implemented in Python 3.8. The choice of this programming language is linked to the use of **Test-Driven Development** and **Continuous Integration (CI)**. In addition, Python offers sophisticated libraries and tools that allow to:

1. create a virtual environment in order to manage the module dependencies in isolation. *venv* is the tool used to keep the development environment isolated from the underlying system;

2. run tests locally before committing to the main development branch. *pytest* is the tool used to create and run automated test batches.

6.1 Project design

The developed code is object-oriented with a design aimed at generality and extensibility.

In the literature there is an archive of environments and respective agents of RL called *OpenAI Gym* [1]. This is a collection of games turned into environments that Reinforcement Learning agents can interact with. Gym environments have an object-oriented interface, so that, in many cases, they can be used as a basis for the development of inherited environments. Many works in the literature are based on Gym object-oriented definitions, including those presented by Anderson et al. in [71] and by Wu et al. in [72] in the field of malware detection. This enables integration with those algorithms compatible with Gym.

In order to comply with the principles of software engineering, such as, for example, loose coupling and high cohesion, the implemented classes are self-dependent. Interaction with objects outside of the class is generalized by using the **Factory Pattern**, a design pattern that parameterizes the creation of an object, through the Factory Method. This makes the object creation logic invisible to the class. Then the object is provided to the class through a common interface. This allows to keep the main classes independent from the implementation of the other classes used, and from the management of external files. Core classes use factory methods to get a generic instance of an element without knowing how it is implemented.

In the work presented, the external data sources are CSV (Comma-separated value) files, these being easily manageable through the standard Python libraries. Using the factory methods and intermediate interfaces, one can easily add support to other formats without changing the rest of the code. This is done by creating a class that implements the interface and uses the factory method to create the object.

Figure 6.1 shows the design Class Diagram for the implementation of the Environment. As a design class diagram, it only shows high-level features and relationships.
The Environment class creates and manages the State and Objective classes. In the model designed there is the elementary entity BasicElement which

[1] https://gym.openai.com/

represents one of the sub-strings that can be inserted in a section. The related class also maintains information about the description and test-coverage.

The sections to split the payload into are set via the SectionName enumeration type. This is the only object to modify if one wishes to add or remove sections. A CSV file is provided for each section, representing the tables shown in section 4.2.2: each row corresponds to a BasicElement. The class that interfaces with external files is CSVSectionStrings, which uses the Python libraries to handle CSV files. This has a ComposedElement object which represents the set of substrings of the section.

The Environment class needs to know the sections and their substrings, but it does not need to know how these data are represented in memory. Using the SectionFactory class one gets a set of SectionStrings which represents a set of sections with their substrings.

The environment has a **state** and this is represented by the State class. This class is a composition of sections. The **sections** are implemented through the class Section which represents the pair *nameSection-substring*.

The environment, upon creation, receives an **objective** as input parameter. The objectives are provided via CSV files, one for each row of table 6.2 below. The logic is the same as presented for SectionStrings objects.
The Objective interface is a specialization of the State class. This is because an objective is to be considered as a successful final state. The objective also carries additional information, such as the reflection context and the optimal sequence of sections to be modified. Such information is only used to categorize the several available objectives. They are never provided to the agent during training, as this would violate the very foundation behind reinforced training.

6.1.1 Agent

Figure 6.2 shows the design Class Diagram for the Agent implementation. As a design class diagram, again, it only shows high-level features and relationships.

The **Agent** class constitutes the entry-point of the developed tool. It corresponds to the application launcher. It has an *Environment* object and a

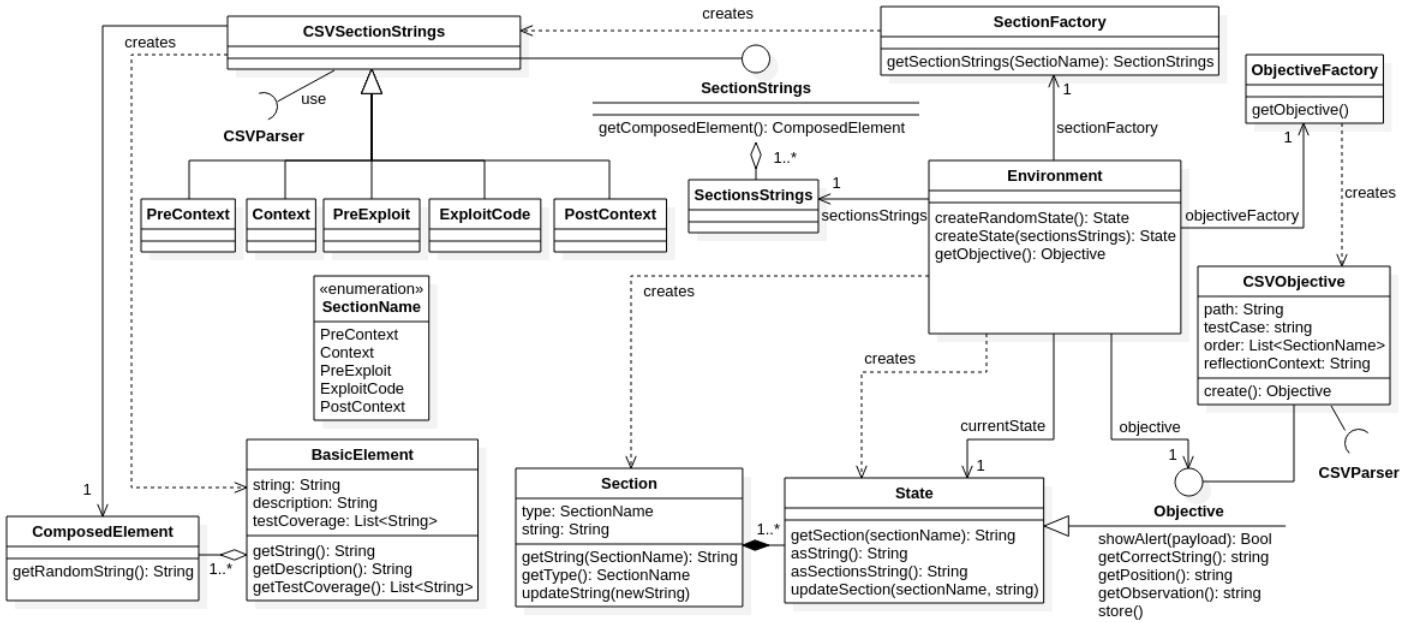

Figure 6.1: Environment - Design Class diagram

Learner object.

The Learner object is an interface that must be implemented by the used learning algorithms. The use of this standard interface and the relative factory method allows to introduce further learning algorithms without having to modify the rest of the code.
Since the Agent class is the launcher, it is the one containing the main() method. Users can customize the execution of the application through the call parameters. In this way a learning algorithm can be selected. Users can choose to train an agent either on one specific objective or on all of the available ones. Users can also decide to test only one policy or all the available ones. In order to support new attack vectors found in the future, users can introduce a new objective by following the command line wizard (CLI), and then train the agent to reach it.

The considered learning algorithms are based on a State-Action function. This is represented by the StateActionSource interface. It will require the implementation of the *exploit()* and *explore()* methods..

The class diagram shows only the **QLearner** class as an implementation of the Learner interface. This has all the parameters and functions to manage ϵ*-decay* using the ϵ-greedy algorithm (*getAction()* method). The *qLearn()* function corresponds to the QLearning update function. QLearning needs the environment to interact with.

QLearning uses the QTable as a representation of the State-Action function. Therefore the QTable object implements the StateActionSource interface.
A QTable can be represented in memory in various ways and, to maintain the extensibility and low coupling of the developed application, the factory pattern is used: the **QTable** object is an interface that it is implemented by the CSVQTable class. The QLearner class will use the QTableFactory class to get the reference to its QTable.

From a logical point of view, a QTable is a composition of **QEntry** entities: each QEntry corresponds to a row of the table and is made up of a state and the list of available actions.

The **Action** class stores the section to be modified, the string to be modified with and its *value*.

The **Environment** class, in addition to the methods seen in the previous section, also carries the methods used by the agent to interact with the environment.

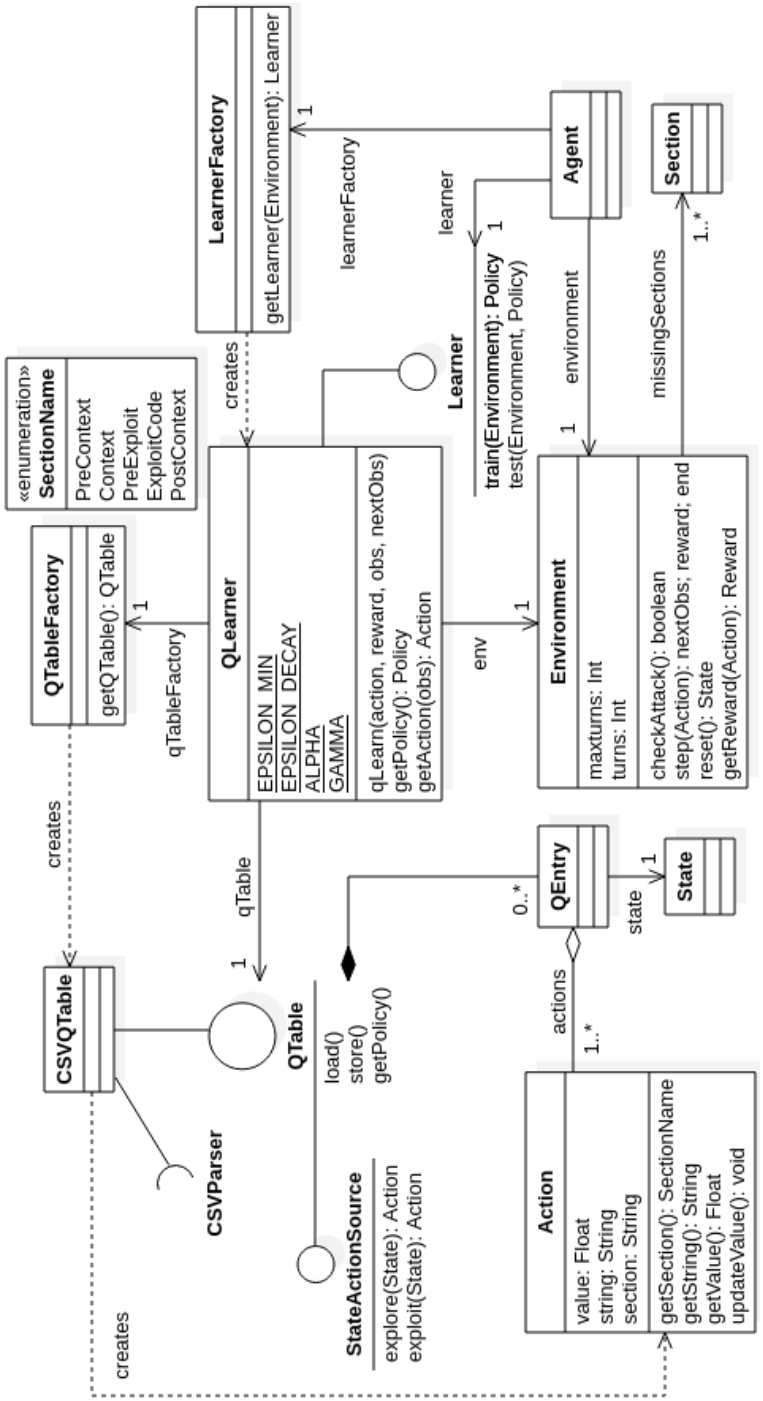

Figure 6.2: Agent - Design Class Diagram

6.2 Implementation

This section presents the implementation details of the design shown in the previous sections.
For the management of CSV files, the CSVUtil class was introduced. This static method class is used to read or write a CSV file and is used by any class whose name carries the CSV word as a prefix.

6.2.1 Environment

Figure 6.3 shows the implementation class diagram for the environment.
The specializations of the CSVSectionStrings class, inherent to each section, have been squashed in a single class, using the type attribute. In fact, the only distinctive element between the two is the name.

One-to-many associations ($1..*$) are replaced by set data structures.
The concrete entity ComposedElement disappears as it is represented by an array containing the BasicElement objects associated with a section.

To implement the state as an aggregation of sections, a Python dictionary was used, whose keys are the section names, whereas the values are Section objects.
Similarly the SectionsStrings class was designed as an aggregation of SectionStrings. The environment has a Python dictionary whose keys are section names and whose values are instances of the SectionStrings class.

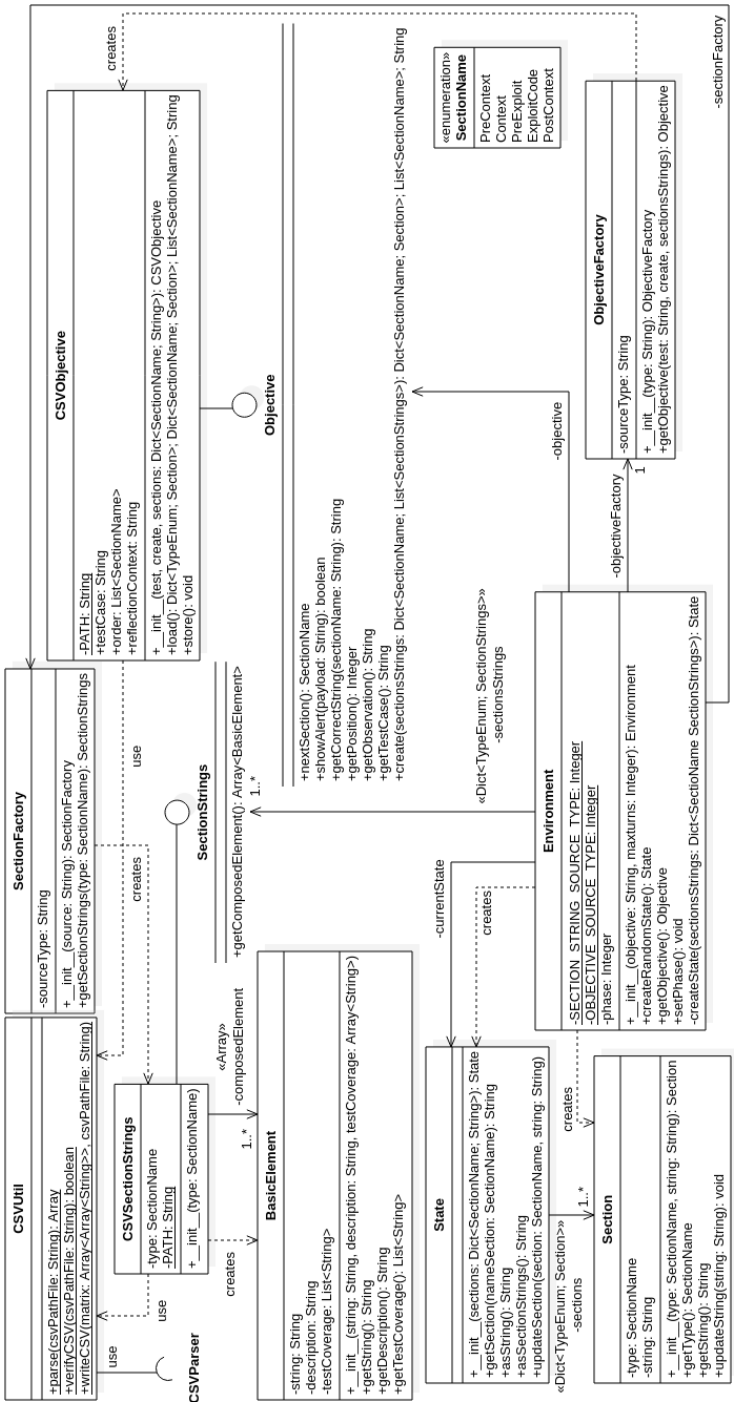

Figure 6.3: Environment - ClassDiagram di implementazione

6.2.2 Agent

Figure 6.5 shows the implementation class diagram for the realization of the agent.

The QEntry entity disappears and is replaced by a Python dictionary whose keys are strings representing states and whose values are lists of actions. Here is an example:

Key: *'"PreContext": "</textarea>", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert (1);", "PostContext": "</script> "'*
Value: *Action_1, Action_2, ..., Action_n*

Where each action is composed of a Section, a String and a Value.
This representation is stored in memory via CSV files. Table 6.2.2 shows an extract.

The **reward function** in the diagram corresponds to that described in chapter 4. This is implemented by the getReward() function which uses the constants and static functions defined within the Environment class. These constants and functions correspond to the values used for the cases of the above function. Snippet 6.1 shows the implementation of the reward function.

```
def getReward(self, action):
  if self.phase == 1: ##### REWARD PHASE 1 #####
    if self.objective.getCorrectString(action.getSection()) ==
    action.getString():
      if action.getSection() in self.missingSections #A1
        self.missingSections.remove(action.getSection())
        reward = CORRECT_STRING_NOT_ALREADY_CORRECT_SECTION(len
    (ActionName) - len(self.missingSections))
      else #A2
        reward = CORRECT_STRING_ALREADY_CORRECT_SECTION
    else:
      if not action.getSection() in self.missingSections: #A4
        reward = WRONG_STRING_ALREADY_CORRECT_SECTION
        self.missingSections.append(action.getSection())
      else: #A3
        reward = WRONG_STRING_NOT_ALREADY_CORRECT_SECTION
  else: ##### REWARD PHASE 2 #####
    if self.objective.getCorrectString(action.getSection()) ==
    action.getString():
      if self.missingSections and self.missingSections[0] ==
    action.getSection(): #B1
        self.missingSections.remove(action.getSection())
        self.order += 1
        reward = CORRECT_STRING_CORRECT_POSITION(self.order)
      else: #B2
        reward = CORRECT_STRING_WRONG_POSITION
    else:
      if self.missingSections and self.missingSections[0] ==
    action.getSection(): #B3
        reward = WRONG_STRING_CORRECT_POSITION
      else: #B4
        if not action.getSection() in self.missingSections:
          self.missingSections.append(action.getSection())
          self.missingSections.sort(key = self.objective.
    getPosition)
          if self.missingSections[0] == action.getSection():
            self.order -= 1
        reward = WRONG_STRING_WRONG_POSITION
    ####### END PHASE 2 ######
  return reward
```

Listing 6.1: function getReward()

To take into account, during training, how the current state approaches to the objective, the list missingSections is used, which contains the ordered list of the sections that are not yet correct. Whenever a correct string is inserted into a section, it is removed from the list. When a section not present in missingSections(corrected) is changed to an incorrect string, the section is reinserted and the sort is restored.
The order variable, used in step two, keeps track of the number of sections correctly filled in, based on the sort order. This is incremented every time a correct string is inserted in the section indicated by the first element of the missingSections list. The variable order is decremented when an already corrected section is modified and, by reinserting the section in missingSections, it results in the first position.

The Agent class, as seen in paragraph 6.1.1, represents the application launcher and provides the main() method. Using Python library ArgumentParser, the possibility for the user to customize the execution has been implemented using the parameters shown in the figure 6.4:

```
usage: Agent.py [-h] [-c TestCaseName | -l TestCaseName | -t TestCaseName | -ta] [--sarsa | --qlearning |

Train and/or test the agent with all objectives. By default it trains and tests all objectives with QLearning algorithm.

optional arguments:
  -h, --help            show this help message and exit
  -c TestCaseName, --create TestCaseName
                        Create and use new objective
  -l TestCaseName, --load TestCaseName
                        Use only the objective in input
  -t TestCaseName, --test TestCaseName
                        Only test (without training) the objective in input
  -ta, --test_all       Only test (without training) all objective
  --sarsa               Use SARSA Learning algorithm for the training phase
  --qlearning           Use QLearning algorithm for the training phase
```

Figure 6.4: Application's help message - supported parameters

State	PreContext:""	...	"Context":""	...	"PreExploit":""	...	"ExploitCode":""	...	"PostContext":""
PreContext: ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; ""; "PostContext"; ""	"0"	...	"0"	...	"0"	...	"0"	...	"0"
PreContext: ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; ""; "PostContext"; "</script>"	"0"	...	"0"	...	"0"	...	"0"	...	"0"
PreContext: ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; ""; "PostContext"; "//"	"0"	...	"0"	...	"0"	...	"0"	...	"0"
PreContext: ""; "Context"; ""; "PreExploit"; ""; ExploitCode"; ""; "PostContext"; "/*"	"0"	...	"0"	...	"0"	...	"0"	...	"0"
PreContext: ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; ""; "PostContext"; """	"0"	...	"0"	...	"0"	...	"0"	...	"0"
"PreContext"; ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; ""; "PostContext"; ""	"0"	...	"0"	...	"0"	...	"0"	...	"0"
"PreContext"; ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; "alert(1);"; "PostContext"; ""	"0"	...	"0"	...	"0"	...	"0"	...	"0"
"PreContext"; ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; "alert(1);"; "PostContext"; "</script>"	"0"	...	"0"	...	"0"	...	"0"	...	"0"
"PreContext"; ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; "alert(1);"; "PostContext"; "//"	"0"	...	"0"	...	"0"	...	"0"	...	"0"
"PreContext"; ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; "alert(1);"; "PostContext"; "/*"	"0"	...	"0"	...	"0"	...	"0"	...	"0"
"PreContext"; ""; "Context"; ""; "PreExploit"; ""; "ExploitCode"; "alert(1);"; "PostContext"; """	"0"	...	"0"	...	"0"	...	"0"	...	"0"
"PreContext"; ""; "Context"; ""; "PreExploit"; """; "ExploitCode"; ""; "PostContext"; ""	"0"	...	"0"	...	"0"	...	"0"	...	"0"
"PreContext"; ""; "Context"; ""; "PreExploit"; """; "ExploitCode"; ""; "PostContext"; "</script>"	"0"	...	"0"	...	"0"	...	"0"	...	"0"
"PreContext"; ""; "Context"; ""; "PreExploit"; """; "ExploitCode"; ""; "PostContext"; "//"	"0"	...	"0"	...	"0"	...	"0"	...	"0"
...	...	...	...	...	...	...	...	...	...

Table 6.1: QTable Sample

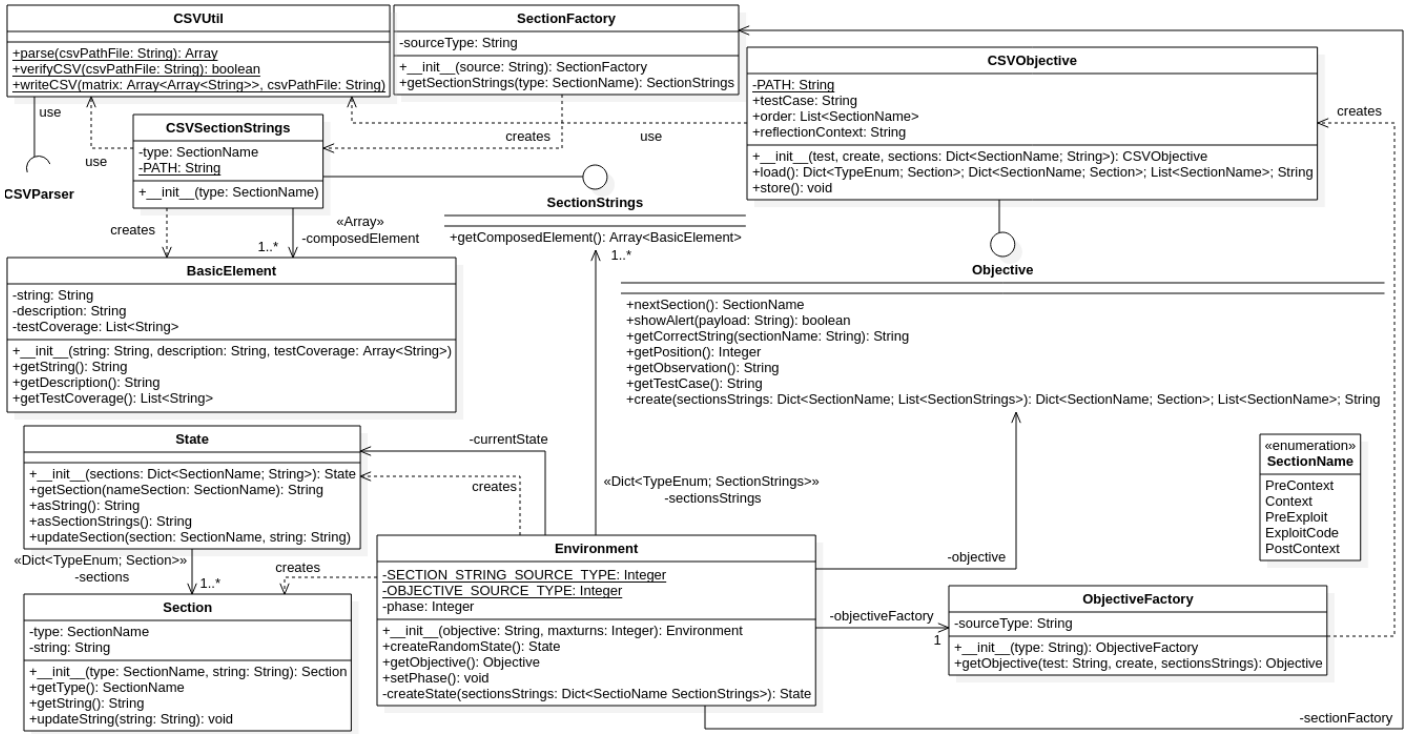

Figure 6.5: Agent - Implementation class diagram

6.3 Observation simulation

As mentioned in section 6.2, the observations obtained through the Observer module in this work are simulated during training. This allows us to consistently speed up training times. The downfall of this approach is that the test cases need to be serialized into objects, which requires manual intervention. The Objective object handles the test cases simulation. It consists of:

- a sequence of $sectionName_1$: $string_1$, $sectionName_2$: $string_2$, ... which corresponds to the list of correct sub-strings for each section. When the current state reaches the configuration expressed in the objective, it means that the payload is functioning correctly.
- an ordered sequence of $sectionName_1$, $sectionName_2$, ... representing the order of payload changes expected by a given methodology. This represents the need to modify or add a substring in a section based on an observation received.
- a reflectionContext representing the key attribute that allows the target to be selected by the agent when it works in an unknown environment.

The first two elements allow users to simulate, for example, the need to escape using the PreContext section; then, to inject a new context using the Context section; to close the context injected via PostContext. In this example the PreExploit section would be the last section to be modified as it is not necessary to enter any characters. The exploit would have happened before having to modify the PreExploit section.
For each test case, this information was manually extracted by applying the correct exploitation model.

Table 6.2 shows all the objectives identified starting from the WAVSEP test cases. Each row of the table represents a CSV file which, as mentioned previously, is managed by the CSVObjective class.

Note that the developed tool allows the manual addition of new simulated objectives, through the CLI interface.

Objective	Order	Reflection Context
PreContext: "", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": ""	ExploitCode, "PreContext", "Context", "PreExploit", "PostContext"	JavaScriptCode
PreContext: "", "Context": "onerror=", "PreExploit": "'", "ExploitCode": "alert(1);", "PostContext": "'"	ExploitCode, "Context", "PreExploit", "PostContext", "PreContext"	AttributeName
PreContext: "", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"	ExploitCode, "Context", "PostContext", "PreContext", "PreExploit"	SimpleHTML
PreContext: "</textarea>", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"	ExploitCode, "PreContext", "Context", "PostContext", "PreExploit"	HTMLTag
PreContext: ">", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	AttributeValue
PreContext: "–>", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"	ExploitCode, "PreContext", "Context", "PostContext", "PreExploit"	HTMLComment
PreContext: "", "Context": "JavaScript:", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": ""	ExploitCode, "Context", "PreContext", "PreExploit", "PostContext"	URLAttribute
PreContext: """, "Context": "onerror=", "PreExploit": """, "ExploitCode": "alert(1);", "PostContext": ""	ExploitCode, "PreContext", "Context", "PreExploit", "PostContext"	URLAttribute
PreContext: "'>", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"	ExploitCode, "PreContext", "Context", "PostContext", "PreExploit"	JavaScriptEvent
PreContext: """, "Context": "onMouseOver=", "PreExploit": """, "ExploitCode": "alert(1);", "PostContext": ""	ExploitCode, "PreContext", "Context", "PreExploit", "PostContext"	URLAttribute
PreContext: "'>", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"	ExploitCode, "PreContext", "Context", "PostContext", "PreExploit"	JavaScriptEvent
PreContext: "'", "Context": "onerror=", "PreExploit": "'", "ExploitCode": "alert(1);", "PostContext": ""	ExploitCode, "PreContext", "Context", "PreExploit", "PostContext"	URLAttribute
PreContext: "'", "Context": "onMouseOver=", "PreExploit": "'", "ExploitCode": "alert(1);", "PostContext": ""	ExploitCode, "PreContext", "Context", "PreExploit", "PostContext"	URLAttribute
PreContext: ">", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"	ExploitCode, "PreContext", "Context", "PostContext", "PreExploit"	JavaScriptEvent
PreContext: ";", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	JavaScriptEvent
PreContext: "");", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	ScriptTag
PreContext: "";", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	JavaScriptEvent
PreContext: "');", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	ScriptTag
PreContext: ";", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	JavaScriptEvent
PreContext: "</style>", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"	ExploitCode, "PreContext", "Context", "PostContext", "PreExploit"	HTMLTag
PreContext: "*/</style>", "Context": "<script>", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "</script>"	ExploitCode, "PreContext", "Context", "PostContext", "PreExploit"	HTMLTag
PreContext: "";", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	JavaScriptEvent
PreContext: "';", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	URLAttribute
PreContext: "";", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	ScriptTag
PreContext: "';", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "//"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	ScriptTag
PreContext: "", "Context": "", "PreExploit": "%0A%0D", "ExploitCode": "alert(1);", "PostContext": ""	ExploitCode, "PreExploit", "PreContext", "Context", "PostContext"	JavaScriptSingleLineComment
PreContext: "*/", "Context": "", "PreExploit": "", "ExploitCode": "alert(1);", "PostContext": "/*"	ExploitCode, "PreContext", "PostContext", "Context", "PreExploit"	JavaScriptMultiLineComment
PreContext: "*/", "Context": "", "PreExploit": "%0A%0D", "ExploitCode": "alert(1);", "PostContext": "/*"	ExploitCode, "PreContext", "PreExploit", "PostContext", "Context"	JavaScriptMultiLineComment

Table 6.2: Objective extracted from WAVSEP test cases

6.4 Configuration

This section illustrates the configuration of the parameters in order to setup the training of the agent.

Snippet 6.2 shows the values used for the reward function discussed in section 6.2.2.

```
#REWARD PHASE 1

BONUS_CORRECT_STRING = +2
WRONG_STRING = -100
CORRECT_STRING = +10

#A1
def CORRECT_STRING_NOT_ALREADY_CORRECT_SECTION(x):
  return CORRECT_STRING + (BONUS_CORRECT_STRING * x)
#A2
CORRECT_STRING_ALREADY_CORRECT_SECTION = WRONG_STRING / 2
#A3
WRONG_STRING_NOT_ALREADY_CORRECT_SECTION = WRONG_STRING
#A4
WRONG_STRING_ALREADY_CORRECT_SECTION = 2 * WRONG_STRING

#REWARD PHASE 2

BONUS_CORRECT_POSITION = +10
WRONG_POSITION = -100

#B1
def CORRECT_STRING_CORRECT_POSITION(y):
  return (CORRECT_STRING + BONUS_CORRECT_POSITION * y)
#B2
CORRECT_STRING_WRONG_POSITION = WRONG_POSITION
#B3
WRONG_STRING_CORRECT_POSITION = WRONG_STRING
#B4
WRONG_STRING_WRONG_POSITION = WRONG_POSITION + WRONG_STRING
```

Listing 6.2: Defining case values reward

Snippet 6.3 shows the configuration parameters used by the QLearner class.

```
class QLearner(Learner):

  EPSILON_MIN = 0.005
  ALPHA = 0.05
  GAMMA = 0.98
```

```
6
7   STEPS_FOR_EPISODE_1 = 5
8   STEPS_FOR_EPISODE_2 = 5
9
10  MAX_NUM_EPISODES_1 = 50000
11  MAX_NUM_EPISODES_2 = 80000
12
13  BALANCE_1 = 400
14  BALANCE_2 = 800
15
16  MAX_NUM_STEPS_1 = STEPS_FOR_EPISODE_1 * MAX_NUM_EPISODES_1
17  MAX_NUM_STEPS_2 = STEPS_FOR_EPISODE_2 * MAX_NUM_EPISODES_2
18
19  EPSILON_DECAY_1 = BALANCE_1 * EPSILON_MIN / MAX_NUM_STEPS_1
20  EPSILON_DECAY_2 = BALANCE_2 * EPSILON_MIN / MAX_NUM_STEPS_2
```

Listing 6.3: QLearner configuration

The parameters ALPHA and BETA are, respectively, the discount factor and the learning rate. Set up in this way, they favor long-term rewards.

6.4.1 Hierarchical training optimization

In order to speed up the training, the option to split the training into two separate, consequent phases, has been provided. We call it **Hierarchical training**. In the first phase the agent takes care of learning how to correctly fill the five sections of the attack payload, regardless of the sequence of sections being modified. In the second phase, the agent learns the optimal sequence of actions to perform. Having already learnt, in the previous training phase, how to correctly fill the 5 sections, the current training session should be easier and converge faster. We will show, in the evaluation section, how this optimization grants shorter training epochs. Some parameters are different for the two training phases. In both phases, each episode consists of more than five steps. This is because an agent must learn to perform at most five actions to get the five correct sections. This number can be decreased if the initial random state has already sections in their optimal state.
The number of episodes (MAX_NUM_EPISODES) for the second phase is greater, having to consolidate both the knowledge acquired in the first phase and that acquired in the second.
The parameters EPSILON_DECAY_1 and EPSILON_DECAY_2 are used for

decaying linear ϵ. The parameters BALANCE_1 and BALANCE_2 have respectively lower value for the first phase and higher for the second. In this way exploration will be favored in the first phase, while in the second phase exploiting will be favored.

6.4.2 Objective selection: module Observer

To obtain the high-level policy that allows you to select the correct target for the agent when testing an arbitrary web application, the Observer module developed in 5 is used.
The Observer module is used to obtain the reflection context (if any) of the input sent to the web application under test. The agent sends, through the Observer module, an initial payload containing only the ExploitCode(alert(1);). The agent sends the HTML request to the application under test and receives a response. This is parsed in order to identify the context of reflection. Starting from this, as seen previously, the objective is selected to extract the optimal policy. This policy will be used to build the final payload.
For each action performed, the Observer module, through the function observe_reflection_context, returns an observation for each section of the payload as seen in section 5. Once the final payload has been reached, the Observer module allows you to check, through the "check_code_execution" function, the actual execution of the payload using the Python *Selenium* extension. This library, through the Firefox and Chrome drivers , allows you to check the actual behavior of a browser upon receipt of an HTML response.

In figure 6.6 the execution flow for the choice of the objective is shown, whereas figure 6.7 shows the way the final payload is produced.

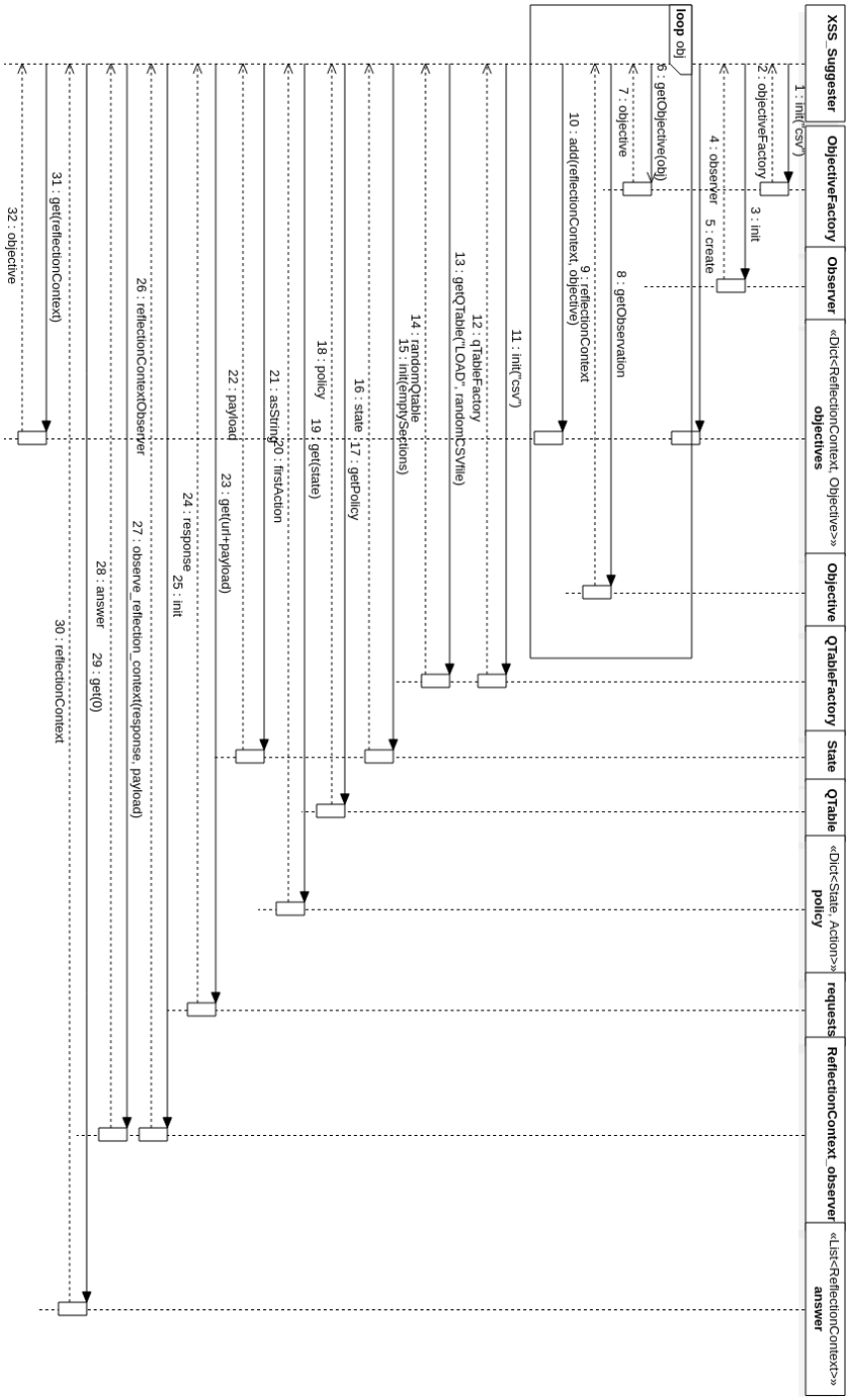

Figure 6.6: Objective selection - Sequence Diagram

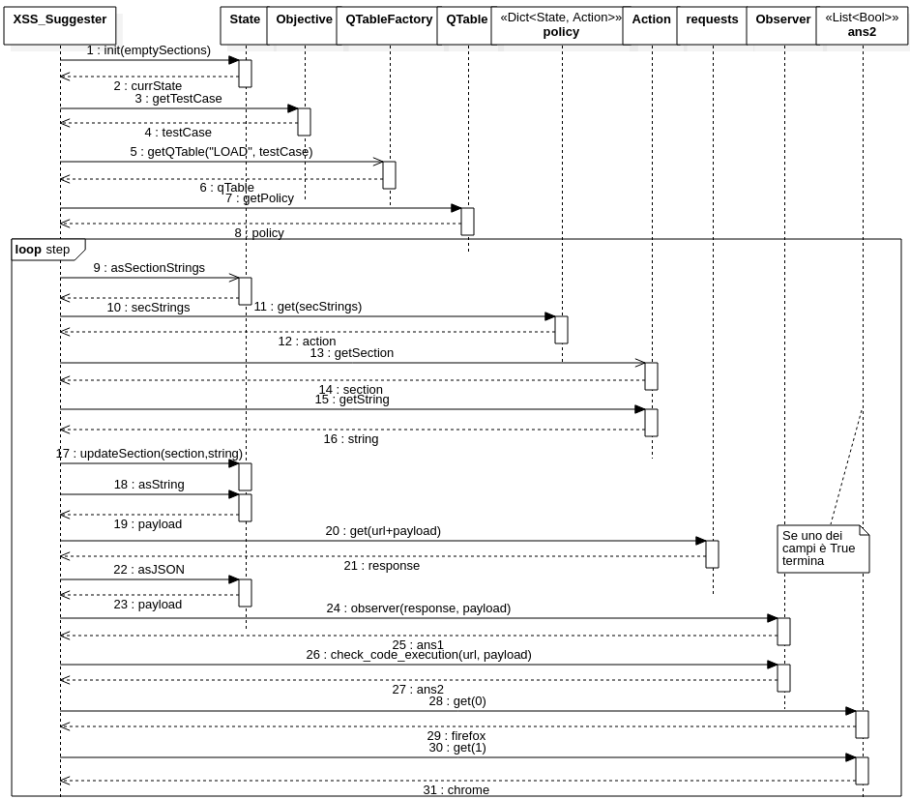

Figure 6.7: Payload execution check - Sequence Diagram

6.5 Test Driven Development

The work discussed in this thesis was developed with a Test Driven Development (TDD) approach [73]. This approach allows to design the tests before implementing the respective functionality. George et al. in [74] run some experiments on two groups of programmers: one writing java code following the TDD approach, the other (control group) following a standard waterfall approach. Authors proved that, even though TDD programmers took much more time to implement the same set of functions, a moderate statistical correlation exists between time spent and the resulting quality. Also, the programmers in the control group often did not write the required automated test cases after completing their code. Hence it could be perceived that waterfall-like approaches do not encourage adequate testing. This intuitive observation supports the perception that TDD has the potential for increasing the level of unit testing in the software industry. At first, tests are designed, then the code that satisfies the test cases is written and only a final refactoring allows to clean up and reorganize the code. Using this programming methodology allows to focus on the essential functions that will be used. These functions are broken down into simpler sub-functions to be able to work on simpler tasks. Once the development of the individual parts is complete, they can be easily integrated. Having already developed and structured the tests, these can be used as a basis for Unit testing. In figure 6.8 the report of the tests carried out through pytest is shown.

20 passed, 0 skipped, 0 failed, 0 errors, 0 expected failures, 0 unexpected passes

Results

Show all details / Hide all details

Result	Test
Passed *(show details)*	tests/test_State.py::TestState::test_updateSection
Passed *(show details)*	tests/test_State.py::TestState::test_getSection_PreExploit
Passed *(show details)*	tests/test_State.py::TestState::test_getSection_PreContext
Passed *(show details)*	tests/test_State.py::TestState::test_getSection_PostContext
Passed *(show details)*	tests/test_State.py::TestState::test_getSection_ExploitCode
Passed *(show details)*	tests/test_State.py::TestState::test_getSection_Context
Passed *(show details)*	tests/test_State.py::TestState::test_asString
Passed *(show details)*	tests/test_State.py::TestState::test_asSectionsString
Passed *(show details)*	tests/test_QTable.py::TestQTable::test_store
Passed *(show details)*	tests/test_QTable.py::TestQTable::test_load
Passed *(show details)*	tests/test_QTable.py::TestQTable::test_explore
Passed *(show details)*	tests/test_QTable.py::TestQTable::test_exploit
Passed *(show details)*	tests/test_QLearner.py::TestQLearner::test_getAction_explore
Passed *(show details)*	tests/test_QLearner.py::TestQLearner::test_getAction_exploit
Passed *(show details)*	tests/test_Objective.py::TestObjective::test_load
Passed *(show details)*	tests/test_Objective.py::TestObjective::test_create
Passed *(show details)*	tests/test_Environment.py::TestEnvironment::test_step
Passed *(show details)*	tests/test_Environment.py::TestEnvironment::test_reset
Passed *(show details)*	tests/test_Environment.py::TestEnvironment::test_checkAttack_true
Passed *(show details)*	tests/test_Environment.py::TestEnvironment::test_checkAttack_false

Figure 6.8: Report Unit Testing

Chapter 7

Performance evaluation

In this section we will evaluate our approach from three perspectives:

- analyse the growth of the state space and how it affects the training process;
- show the improvements in terms of accuracy by comparing the Suggester to XSS automated scanners;
- show how the amount of requests performed by each automated scanner is affected by parameters that identify its level of 'intelligence'.

7.1 State space explosion

The presented environment is very extensible, in the sense that new attack strings can easily be added, as long as they comply with the defined state structure. This can be useful whenever a new way to discover a vulnerability is found.
Adding a new attack string means introducing a new objective, upon which the agent needs to be trained. If the actions defined in the environment already allow for the right construction of the new string, the only requirement is to train the agent in order to reach the new objective. However, if new substrings are included - for instance, a new context is found - the action and state space grows. This means that the agent needs to be re-trained upon the separate objectives. We performed several experiments to analyse how the growth of the state space influences the number of episodes needed for the training.
We added one objective at a time to our environment and repeated the training.

We noted down, for each experiment, the state and action space cardinality and, of course, the number of episodes that allow the training to converge. We consider the training to be complete whenever the columns of the q-table representative of the five best actions to be performed in the five sections of the string assume values that are greater than those of the remaining columns.
Training an agent involves two aspects of randomness. First, when an episode of training is over, the environment is initialised with a random new state. Second, depending on the exploration factor, in the early episodes of training, actions are chosen more randomly than in the last ones. To deal with randomness, we employed an amortised analysis. We repeated the training 30 times; then, at the end of each experiment, we computed the statistical mean upon the number of training episodes. The results of each experiment are reported in Table 7.1 and are plotted in Figure 7.3.

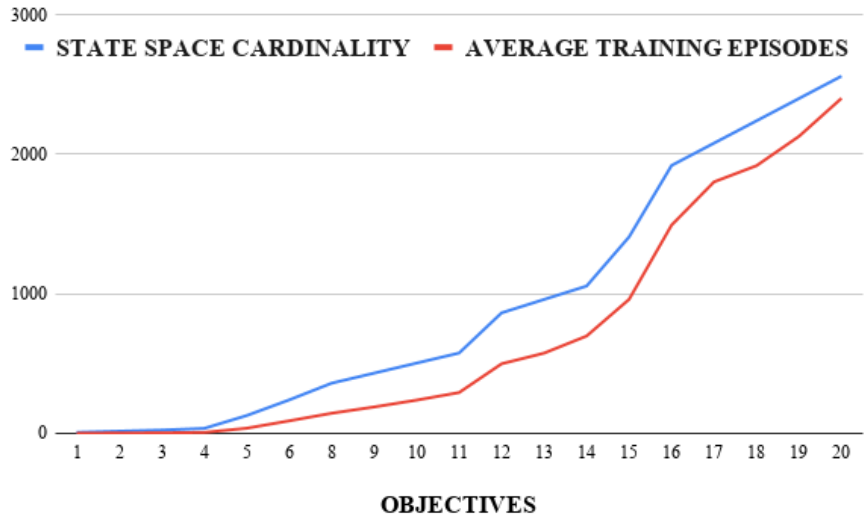

Figure 7.1: Increase in training episodes and state space as the number of objectives grows

The plot in Figure 7.3 shows that the state space increases linearly as new objectives are added to the environment. If we take into account the objectives that cover all of the cases described in section 4.1, the final amount of states is 2560.

Objective	State Space	Action Space	Average Training Episodes
HTML	8	8	1.166667
HTML tag	16	9	2.7
HTML comment	24	10	4.066667
URL attribute	36	11	7.066667
tag attribute	128	14	37.166667
attribute filtering	240	16	90.133333
attribute name + event attribute quote-less	360	17	144.566667
other attribute single quote	432	18	189.6
other attribute double quote	504	19	238.633333
other attribute quoteless	576	20	292.833333
javascript variable single quote	864	22	499
javascript variable double quote	960	23	575.333333
javascript numerical variable	1056	24	697.8
javascript single line comment	1408	25	960.6
javascript multi line comment	1920	27	1492.666667
code function single quote	2080	28	1802.333333
code function single quote	2240	29	1918.5
css	2400	30	2129
css comment	2560	31	2401.366667

Table 7.1: Increase in number of training episodes, state and action space as the number of objectives grows

The curve highlighted in red, which represents the increase in the average amount of episodes of training, also grows linearly with the number of objectives and follows the growth of the state space. However, we notice that the curve of the training episodes increases at a rate that is always upper-bounded by the one of the state space cardinality. This means that we need an amount of episodes of training that is, on average, lower than the number of states that make up the environment. In terms of asymptotic behavior, we can affirm that the state space cardinality function is a *big O* for the amount of training episodes:

$$S(n) = \mathcal{O}(E(n))$$

where *S(n)* represents the function for the states and *E(n)* the function for the episodes of training. This assures that no increase of the state space will cause an unsustainable increase of the training episodes. In fact, in order to train an agent, one does not need to undertake a number of episodes of training which is greater than the amount of states in the environment.
We believe that this behavior is due to the agent being able to explore, in the worst case, up to 200 actions, before the current episode of training ends. This allows the agent to visit all the states in the environment a sufficient amount of times and let the training converge. Furthermore, this signifies that the design of the reward function is good enough to capture the desired behavior of the agent under training.

7.2 Automated scanners accuracy comparison

To show that the implemented methodology results in an improvement in terms of accuracy, the Suggester was compared to other open source tools. Six automated scanners were selected: four of them are specifically designed for the discovery of XSS vulnerabilities, the remaining ones are modules of popular web application scanners, such as *wapiti*[1] and *w3af*[2].
The Yahoo Webseclab[3] was chosen as evaluation platform. It encompasses 31 Cross-Site Scripting test cases: 20 are actual vulnerabilities, whereas the remaining 11 are intentional false positives. In the case of Cross-Site Scripting, false positives are injected inputs that get reflected in the HTML of the response, but do not lead to execution of code. Such test cases are particularly

[1] `https://wapiti.sourceforge.io/`
[2] `http://w3af.org/`
[3] `https://github.com/yahoo/webseclab`

Scanner	TP	FN	TN	FP	True Positive rate (Recall)	False Positive rate	Youden Index	Precision
Suggester	14	6	11	0	0,70	0,00	0,70	1,00
BruteXSS	4	17	7	3	0,19	0,30	-0,11	0,571
XSpear	6	13	10	2	0,316	0,167	0,149	0,75
XSSer	14	5	0	12	0,737	1,00	-0,263	0,538
XSSmap	6	13	12	0	0,316	0,00	0,316	1,00
w3af	14	5	4	8	0,737	0,667	0,007	0,636
wapiti	10	11	6	4	0,476	0,40	0,076	0,714

Table 7.2: XSS scanners confusion matrix

effective to show the accuracy of those automated scanners that only take into account the reflection of a parameter in order to mark it as vulnerable.
Each of the selected tools was provided with the URLs of Yahoo Webseclab test cases and then launched against them. Though different tools provide scan results using different outputs, each of them reports whether the parameter was marked as vulnerable or not. We analyzed the reports and categorized tools' results into one of the following classes:

- *True Positives* (TP): vulnerable parameters correctly classified as vulnerable;
- *True Negatives* (TN): non vulnerable parameters correctly classified as non vulnerable;
- *False Negatives* (FN): vulnerable parameters not classified as vulnerable;
- *False Positives* (FP): non vulnerable parameters incorrectly classified as vulnerable.

Table 7.2 summarizes the results of the experiments and reports some fundamental metrics. Precision and recall charts are reported in Figure 7.2
The Suggester reaches high levels of both precision and recall rates (higher than 70%). High precision values ensure that the tool does not produce False Positives. This is made possible by the fact that the tool recommends users to check for the execution of the exploit code. On the other hand, high recall rates

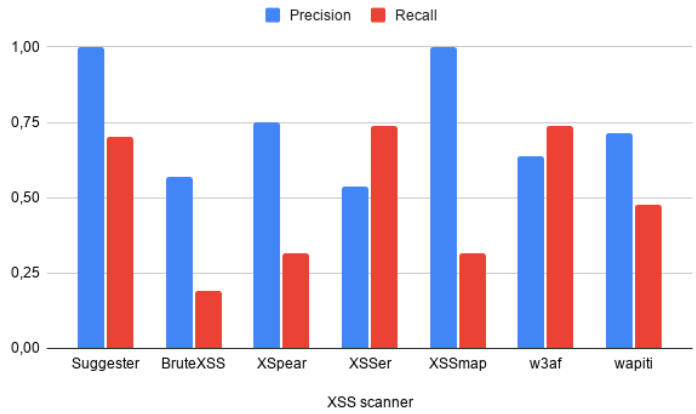

Figure 7.2: Cross-Site Scripting scanners precision and recall comparison

ensure that the tool was able to discover the majority of vulnerabilities present in the target benchmark. This is to be attributed to the number of discovery techniques the tool is able to suggest, that in turn depend on the introduced contexts of reflection. The other tool that reached 100% of precision is XSSmap. In fact, XSSmap has a set of hardcoded rules that allow to distinguish between different contexts of reflection. Moreover, the use of a headless browser allows to detect those javascript events triggered when the exploit code is executed. Therefore, it truly allows to avoid false positives. However, the tool fails at recognizing many reflection contexts, which explains the low recall values.
The remaining tools do not rely upon systems that allow to detect the execution of javascript code, reason why they all report false positives. This explains the lower precision values they attain. Such tools only take into account the reflection of the parameter under test, that is not a reliable way to decide whether the application is vulnerable to XSS. In fact, many injected inputs are reflected back after being correctly sanitized by the application, resulting in no vulnerability. This discovery technique results in a behavior that is close to randomness: in fact, such tools rightfully mark some parameters as vulnerable, but they fail at identifying the attack string that enables the exploit. This explains why tools like XSSer, w3af and wapiti have good recall values, although they report low precision values. Penetration testers can rely on them just to obtain a general idea of those request parameters that are reflected in the response after an injection. However, these would still need further manual inspection to verify whether the reflections conduct to actual vulnerabilities. If tools report

a reflection, but fail at identifying the right attack payload, penetration testers need to start over and craft their own attack strings.
The same reasoning applies also to tools like BruteXSS[4] and XSpear[5], which do not perform checks on the execution of the exploit code. They only look for reflection of injected request parameters, but they also fail at identifying many vulnerabilities, hence resulting in a low recall value.
To underline random discovery behaviour in scanners' performance, the OWASP benchmark evaluation project (OWASP WBE) [75] proposes an effective evaluation system. It was used by Amankwah et al. in [76] as a scoring solution to perform a comparison between open-source and commercial web security scanners. It also provides a visual representation of a tool's discovery performance based on false positive and recall rates.
The benchmark is based on the computation of the so-called Youden Index [77], a metric proposed to evaluate the performance of analytical (diagnostic) tests. It outputs values in the range $[-1, 1]$. A value of 1 indicates discovery of all vulnerabilities, with no false positives. A value of -1 indicates all false positives and no true positives (no actual vulnerabilities discovered). A Youden index of 0 means the tool recorded the same result for a web application with vulnerabilities and without vulnerabilities.
It is calculated with the following formula:

$$J = \frac{TP}{TP + FN} + \frac{TN}{TN + FP} - 1$$

Figure 7.3 shows the True Positive rates associated with the tools under evaluation, with respect to their False Positive rates. The red line that crosses the plot represents the guessing line. Tools that fall along this line have a Youden Index equal to 0, which means they have the same behavior with both vulnerable and non vulnerable applications. The distance from this line is the actual Youden Index.
The Suggester is the tool with the highest distance from the guessing line and hence reaches the top score. XSSmap follows behind, with a lower score. Tools like XSpear, w3af and wapiti perform better than random guessing, but the low Youden Index signifies that penetration testers cannot completely rely on their discovery capabilities. Finally, BruteXSS and XSSer fall in the red area of tools that would have performed better if they randomly guessed.

[4] https://github.com/shawarkhanethicalhacker/BruteXSS-1
[5] https://github.com/hahwul/XSpear

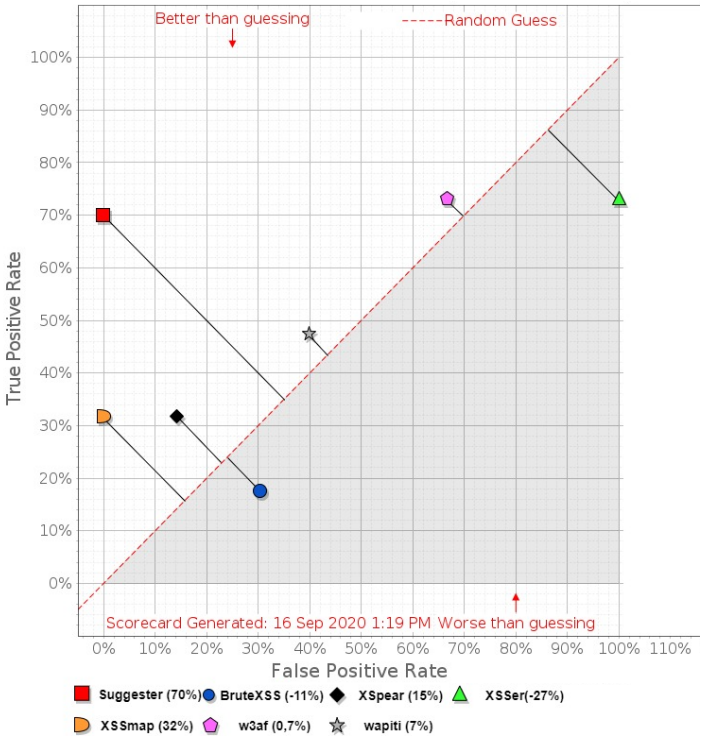

Figure 7.3: Cross-Site Scripting scanners score comparison

7.2.1 Number of requests

The results discussed in Section **??** show that the improvements in terms of accuracy depend on two parameters that signify the difference between a brute force and a more intelligent approach:

- the ability to identify the context of reflection;
- the ability to recognise whether the exploit code was actually executed by the browser.

Another performance indicator that is directly affected by such parameters is the number of requests that a tool performs in order to discover a vulnerability. Brute force tools tend to perform a high number of requests, depending on the size of the word lists they rely upon. More intelligent tools apply rules

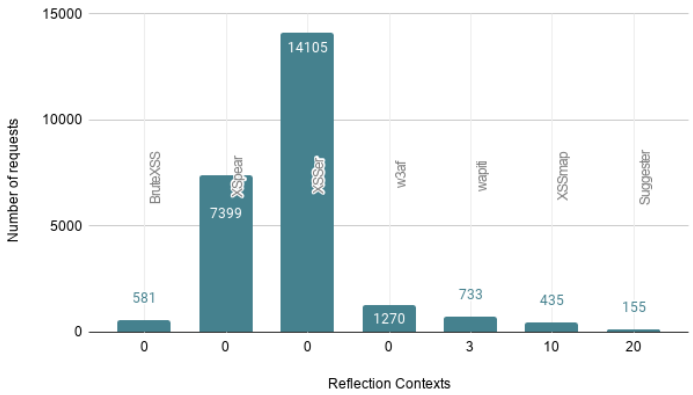

Figure 7.4: Automated scanners amount of requests comparison

that refine the actions triggered against the target application. The amount of requests becomes important during penetration testing campaigns, which are usually conducted when systems are already in production. Brute force scanners certainly stress the system under test, incurring the risk, in some cases, to perform actual Denial of Service attacks.
To show how the knowledge of the context of reflection affects tools' behavior, the total amount of requests performed to solve the Yahoo Webseclab's test cases was recorded. Results are shown in Figure 7.4. In order to correctly interpret the results, a distinction between two categories of tools needs to be made:

- *context-blind tools*: scanners that are not able to recognize the exact spot in the HTML response where the reflection of the injected parameter happens. They are BruteXSS, XSpear, XSSer and w3af;
- *context-aware tools*: scanners that are able to recognize the spot in the HTML where the reflection of the injected parameter takes place. They are wapiti, XSSmap and Suggester.

For context-blind tools, the amount of HTTP requests depends on the size of the word list they employ. In fact, BruteXSS has the shortest word list, reaching a total amount of 581 HTTP requests. XSSer reaches 14105 HTTP requests, to solve all 31 Webseclab test cases.
Coming to context-aware tools, the amount of HTTP requests decreases as the number of contexts encompassed by the scanner increases. As to "wapiti", it

is still a brute force tool. Though, it categorizes the attack payloads on the basis of three different reflection contexts and attempts to solve the 31 test cases with 733 total HTTP requests. When "XSSmap" is able to recognize the context of reflection, it automatically sends an associated attack string. However, it falls back to a classical brute force attack tool, with a small word list, when it is not able to recognize the context of reflection. This behavior is able to ensure a lower amount of HTTP requests, namely 435. The Suggester performs better than any other tool, with a total amount of 155 HTTP requests across the 31 test cases. In fact, it is able to recommend up to 20 different contexts of reflection. Moreover, it does not rely on a brute force approach, but dynamically constructs the attack payload request by request, by analyzing the feedbacks returned by the web application.

7.3 Algorithm comparison

This section shows how the developed methodology converges in a relatively small number of episodes. By convergence we mean that the State-Action function, for each state, has an action with a dominant value. In other words, an algorithm reaches convergence when negative rewards are no longer received and errors decrease dramatically.
Note that the errors will tend to zero but are unlikely to reach it, since the value of ϵ, however small it gets, will never reset, continuing to allow exploration.
The results obtained through the following Reinforcement Learning algorithms will be compared below.

- QLeraning (QL): Off-Policy algorithm (assumes that the agent follows the best policy at all times). The State-Action function is represented by a table structure (QTable).
- SARSA: On-Policy algorithm (the agent follows the current policy). Also in this case the State-Action function is represented through a tabular structure.
- SARSA (λ): Optimization of the SARSA algorithm in which the Eligibility trace mechanism is used. The trace identifies the usability of a state for training.
- DeepQLearning (DQN): Similar to QLearning where there is no tabular structure for the Action-State function but this is approximated via a neural network.

- MonteCarlo Tree Search (MCTS): Heuristic search algorithm for decision making, especially in game theory. Represents sequences of choices as paths in a tree.

The tests were carried out with the help of the following machine:

- CPU: Intel® Core i7 2600K
- RAM: 16 GB (4 x 4 GB - 1333 MHz)
- SSD: Samsung 840 EVO 250 GB
- GPU: Intel® HD 3000
- S.O .: Fedora 33 (Kernel 5.8.18-300)
- Python: 3.8.3

The implemented DQN is based on the version provided by OpenAI Gym, developed through PyTorch. The following optimizations are used:

- Experience Replay
- Batch sampling
- Target Network
- $epsilon$ -Decay Scheduler

Note that the experiments are conducted without using a dedicated GPU.

The MCTS algorithm is based on an open-source version compatible with Gym [6] environments.

The results obtained during the training phase on all objectives are taken into account. In particular, the following parameters will be analyzed as the training episodes increase:

- **Errors**: The average number of errors made while training the agent on all targets.
- **Rewards**: The average of the rewards received by the agent during the agent's training on all objectives.

[6] https://gist.github.com/blole/dfebbec182e6b72ec16b66cc7e331110

As expected, the reward will tend to increase and errors will tend to decrease as the episodes increase.

The results are divided into the two phases of exploration. For the first phase, as explained in the previous chapters, fewer episodes were used than in the second phase: *500000* and *800000* (five hundred thousand and eight hundred thousand) episodes.
The hyperparameters used for the algorithms are shown in appendix **??**. These have been obtained experimentally trying to speed up the convergence.

Figure 7.5 and figure 7.6 show the results obtained in the first training phase, whose objective is to make sure the final payload is reached, regardless of the sequence of actions performed. In figure 7.7 and figure 7.8 results obtained in the second training phase are reported. In the second phase, the reward function is designed to train the agent to perform the optimal sequence of actions.

Figure 7.5: QLearning (QL) Vs SARSA Vs SARSA(λ) Vs Deep-QLearning (DQN) - Errori fase 1 (well-formedness)

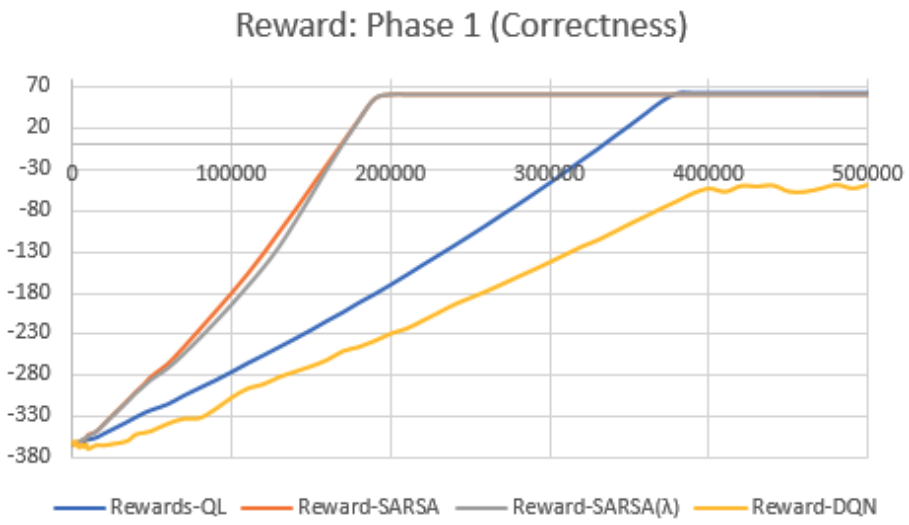

Figure 7.6: QLearning (QL) Vs SARSA Vs SARSA(λ) Vs Deep-QLearning (DQN) - Reward fase 1 (well-formedness)

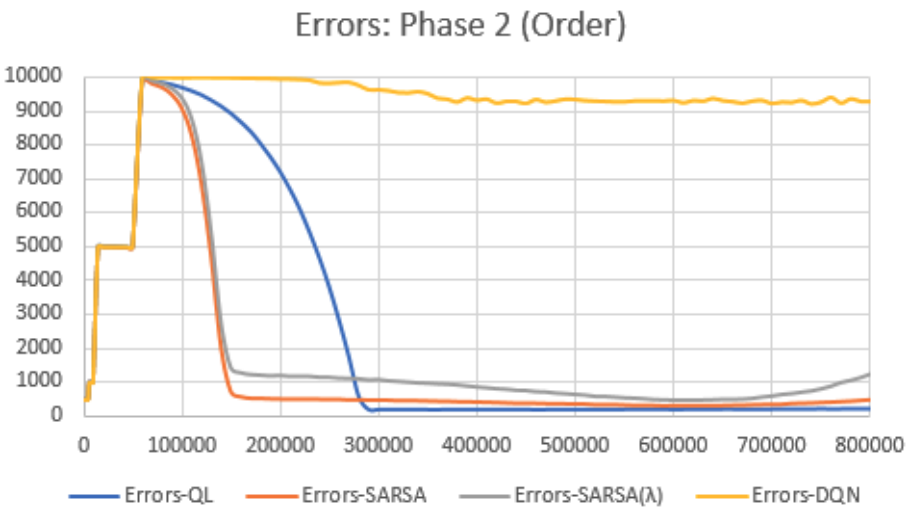

Figure 7.7: QLearning (QL) Vs SARSA Vs SARSA(λ) Vs Deep-QLearning (DQN) - Errori fase 2 (right order)

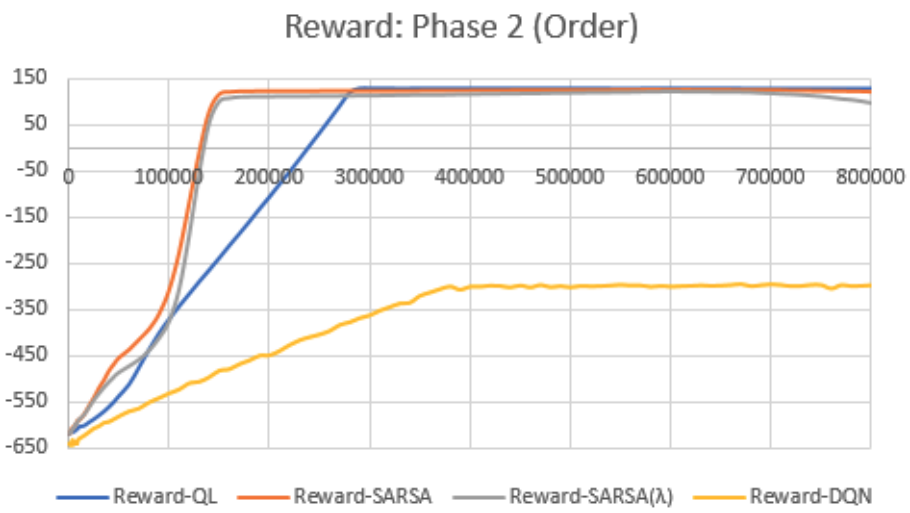

Figure 7.8: QLearning (QL) Vs SARSA Vs SARSA(λ) Vs Deep-QLearning (DQN) - Reward fase 2 (right order)

MCTS produced poor quality results. This is due to the fact that many state-action pairs fail to be visited. This problem is related to the matter of keeping exploration. Even using a high exploration rate, however, the algorithm does not scale sufficiently in the environment presented in this work, remaining locked in a single branch of the tree. In the literature this is a well known problem and is described by Sutton et al. in [78].

The best results are obtained through the QLearning and SARSA algorithms. In particular, the greater convergence speed of the SARSA algorithm stands out. In both phases, errors in SARSA drop more abruptly than those produced by QLearning which have a smoother trend. An opposite trend is found when evaluating the rewards received: SARSA reaches the maximum reward about 150,000 (one hundred and fifty thousand) episodes before QLearning, with faster growth. This behavior is due to the fact that, once the safest path has been identified, SARSA has a lower probability of choosing wrong actions than QLearning. SARSA tends to consider the future more, updating the QValue of the current action only after performing the next action.
In the developed environment, for each state, the majority of the actions turn out to be wrong. Therefore SARSA will tend to avoid those actions that would lead to a state in which the probability of choosing the right action would be even lower.

By applying the *eligibility trace* technique to the SARSA algorithm, results comparable to those of the traditional version are obtained, without justifying the longer time needed for training. The increase in time comes from having to manage an additional table for keeping track. This table has a size equal to that of the QTable.
Using a neural network to approximate the State-Action function is not a good choice. Deep-QLearning, in literature, is successfully used to tackle problems whose state space is continuous and not manageable through QTable. In the model presented, the state space is categorical. Analyzing the results, it can be seen how the values are fluctuating and far from convergence. More stability would probably be achieved by increasing the number of episodes. The main problem with the use of DQN is that training times increase dramatically, making the increase in the number of episodes unacceptable. This is due to the fact that although a small number of internal nodes are used, the propagation of the input and the updating of the weights take longer to compute than it

takes to update a QTable entry.
The results of the first phase are better than those of the second. This is because phase two uses the knowledge gained in the first phase as a starting point.

Since the results are reported in the form of arithmetic means, it is also important to evaluate the standard deviation. As regards the errors of the first phase, as the episodes increase, the standard deviation increases reaching a maximum value between 40 and 50 around 150,000 episodes of training for SARSA and SARSA (λ) and 300,000 episodes for QLearning. This is due to the fact that the agent will initially choose an action at random. From this point on, the standard deviation is lowered, stabilizing in a range between 15 and 20. This behavior is due to the strong decline in the number of errors following the achievement of convergence.
In the second phase, the standard deviation trend mirrors that of the first phase, reaching a maximum value between 25 and 35 near the 200,000 episodes for SARSA and SARSA (λ) and 250,000 episodes for QLearning. From here the standard deviation will tend to stabilize within a range between 2 and 5.

As for the reward values, these present a decreasing standard deviation. Starting from a value between 3 and 5 for the first phase and between 15 and 20 for the second, the standard deviation will decrease until it stabilizes around 0. This behavior is due to the fact that at the beginning of learning the actions they are chosen randomly, producing reward values included in a very large set; as the number of episodes increases, the reward values will tend to stabilize following the achievement of convergence.

As far as Deep-QLearning is concerned, the standard deviation of the errors, in both phases, underlines the profound instability of the data reaching an ever increasing value and exceeding the 2000 threshold during the first phase and 1500 in the second phase.
The reward values also have an ever-increasing standard deviation in both phases, exceeding the 60 threshold.
Such a standard deviation is due to the fact that Deep-QLearning is far from achieving convergence.

7.3.1 Comparison between hierarchical and unified training

This section illustrates the difference, in terms of performance, between the two learning approaches described in chapter 6, section 6.4.1. For unified

training we use *800000* (eight hundred thousand) episodes with a standard " balance " value (500) for the ϵ-decay.

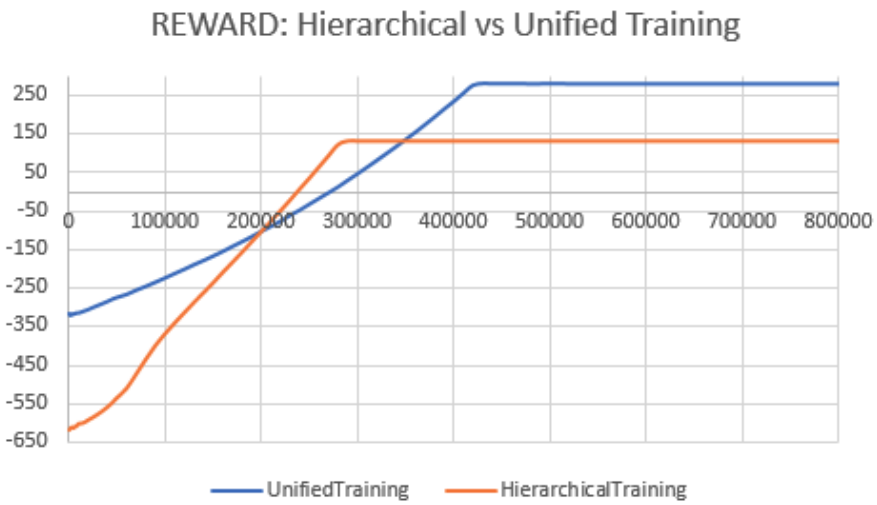

Figure 7.9: Unified vs. hierarchical training - Rewards

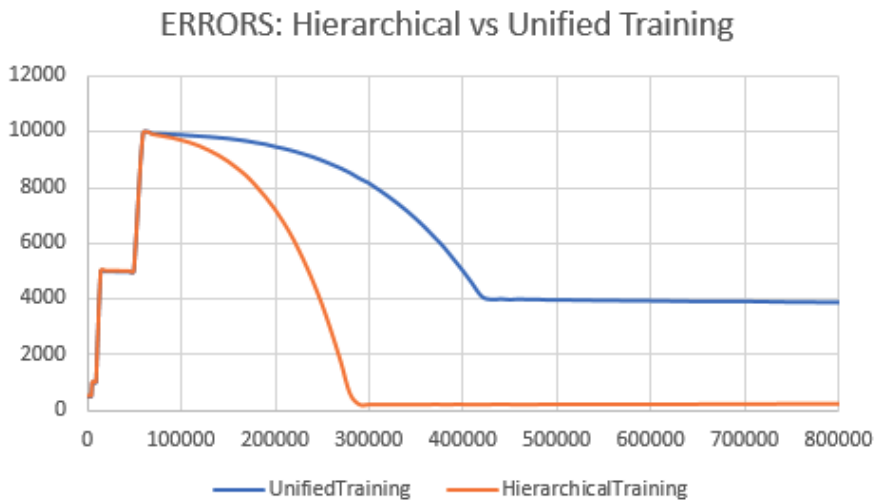

Figure 7.10: Unified vs. hierarchical training - Errors

Results are obtained from QLearning training. Hierarchical training takes into consideration the second training phase in which errors are counted both in the ordering and in the correctness of the payload.
It is noted how, even if the rewards are positive, the errors are high. These derive from the lack of ordering (second parameter to be optimized). This is due to the complexity of the reward function shown in **??** which does not allow the agent to capture the desired behavior. Even by increasing the number of episodes it is not possible to reach a stability equal to that obtained with the hierarchical approach.

The standard deviation of the errors increases continuously when exceeding the 2000 threshold, as well as the standard deviation of the rewards that exceeds the 90 threshold. This behavior is due to the difficulty in reaching convergence through unified learning.

Chapter 8

Other approaches to security testing automation

In this section, two promising approaches to provide automation to penetration testing are described. Both of them follow the philosophy behind this entire thesis, which is to create automated tools for security testing, improving their intelligence. The first one explores one of the traditional ways known in Artificial Intelligence to design systems that support experts' decisions, by creating a penetration testing ontology and supporting it with an inference engine able to relationships among entities of the domain. The novelty of the presented approach is in the representation of the domain knowledge using knowledge graphs. Such design choice is useful because it allows to take advantage of graph theory to compute interesting measurements, as well as because it grants good visualization of attack paths by means of nodes and their connections.
The second approach, on the other hand, leverages a very well known Artificial Intelligence pattern, which is supervised learning. The purpose is to create a platform which is able to capture a penetration testing session, create a feature representation (e.g. network traffic, screenshots, browser events) and store a dataset. Such dataset would enable the use of artificial neural networks for the realization of intelligent agents for penetration testing.

8.1 A penetration testing expert system based on knowledge graphs

Penetration testing can be very time-consuming and does not ensure that all vulnerabilities in the system under investigation are found. It is also possible that the penetration tester is asked to focus on a particular type of vulnerability or a particular component of the computer system (e.g. database). In general, the success of the penetration test depends on the skills and on time available to the penetration tester. For this reason, many efforts have been made in recent years to automate the penetration testing activity. In fact, the automation of penetration testing can lead to significant advantages in terms of time, compared to the activity manually performed. However, many tools that automate penetration testing can suffer from the problem of false positives, i.e. vulnerabilities detected as such, but which are not so. Furthermore, sometimes, with such tools, it is not possible to customize the penetration test to fit the customer's needs. Generally, results got by the tools, need to be reviewed by the penetration tester and possibly deepened (based on his/her knowledge). The Owasp Testing Guide [79] provides a standard approach, that the security expert can follow when carrying out penetration testing on a web application. The purpose of this work is to provide a tool that assists the penetration tester during his/her activity, recommending the best action to take based on his/her needs. The methodology of the developed tool follows the one contained in the OTG, used for testing web applications. It should be noted, however, that it is possible to extend the range of use of the built system to other areas, simply by adapting the data contained therein based on the area of testing.

8.1.1 Design

In this section, the design of the final developed tool is proposed. Its purpose is to suggest to the user the sequence of security tasks to perform according to the state of the system under test, in order to complete a Hacking Goal (a testing objective). The core concepts are explained in detail and the choices will be motivated with reference to the studied resources.

8.1.2 Entity Relationship diagram

This section primarily focuses on a high level view of data structure, building an Entity Relationship (E-R) diagram. This will be the core part of the application, since data must be compliant to this model in order 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. The following image shows the implemented E-R diagram:

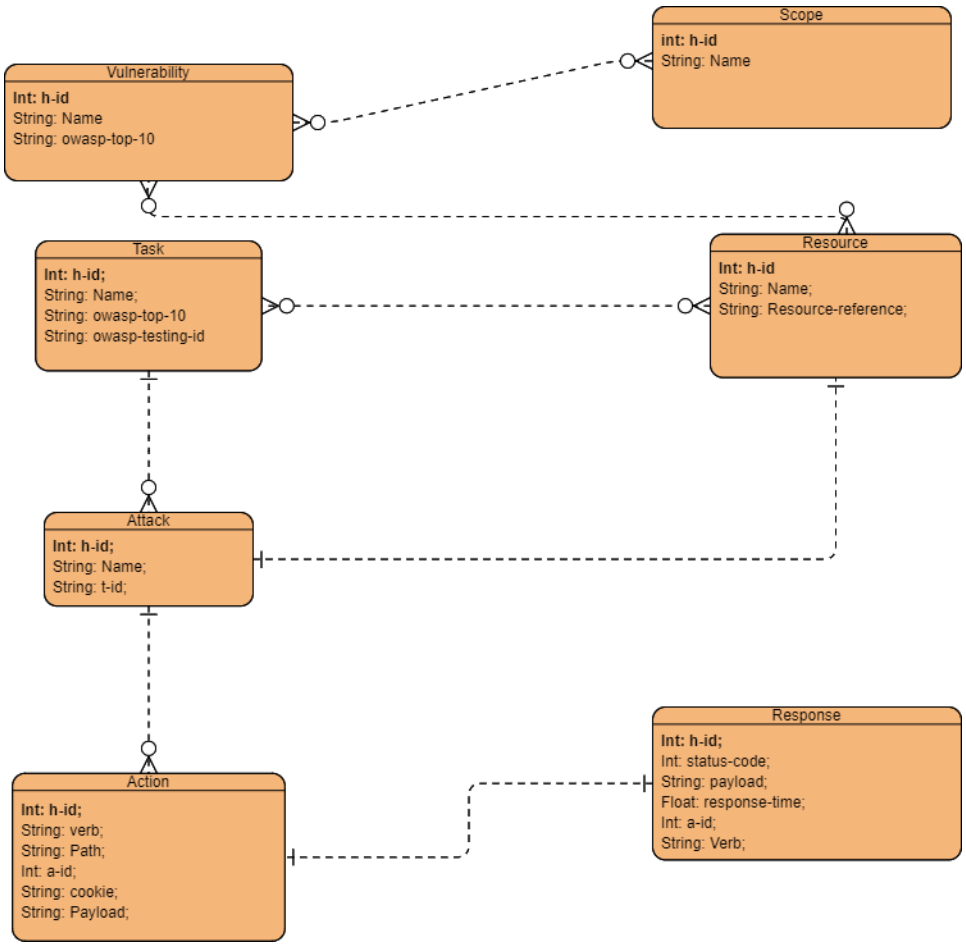

Figure 8.1: E-R Diagram

Here a detailed explanation of entities and their attributes is given:

- 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." [80]. This entity has been modeled 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 basing on their id, or search for a group of vulnerabilities that belong to the same category. A vulnerability has a many-to-many relation with the entity "Scope" and "Resource", because: 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 resources (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" [80]. 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 whatever web application's entity which can be either used or discovered by an action performed on the application itself. It can be considered as an application asset, whose knowledge can be either provided by a task or used attempting to exploit a vulnerability. The unique identifier is the "h-id", whereas the "Name" briefly describes the resource. The "Resource-Reference" is the category the resource belongs to, and allows for the coarse-grained queries. 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 used by different Vulnerabilities; a resource can be used by many different tasks, or knowledge about a resource can be given/needed by many task. For example:

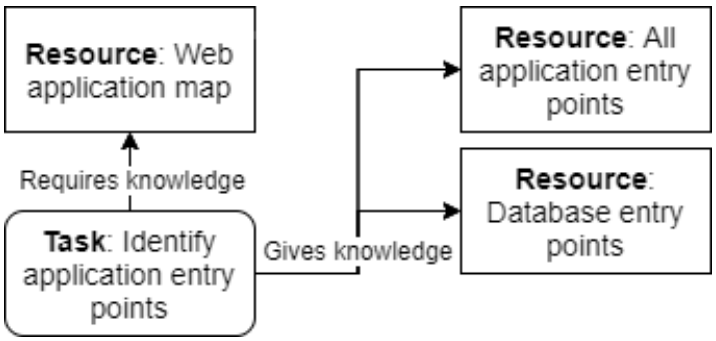

Figure 8.2: Knowledge relationship with resource example

- 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 well described in the Relation Diagram paragraph. "Fingerprint Web Server" (i.e. try to understand the operative system that runs on the Web Server), "Testing for Reflected Cross site scripting" (i.e. perform a test to assess whether the Cross site scripting vulnerability is present), "Identify application entry points" (i.e. application's parameters used in HTTP requests), are only three examples of what a task is. 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 to 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 by multiple attacks and is linked to multiple resources. In fact, there may be different links among tasks and resources, for example there's a relationship of "acquired knowledge", when a task provides the knowledge about a resource (e.g. knowing the resource "Operating System" means having knowledge about the specific type of Operating system that is running on the target; knowing the resource "FTP area" means knowing whether the area really exists in the target application and if it is reachable or not, which files are contained in it, etc.), or "required knowledge", when a task needs the knowledge about a resource, or a "task-to-resource" when a task uses a resource in order to be completed. Here's an example of a task composed by multiple attacks:

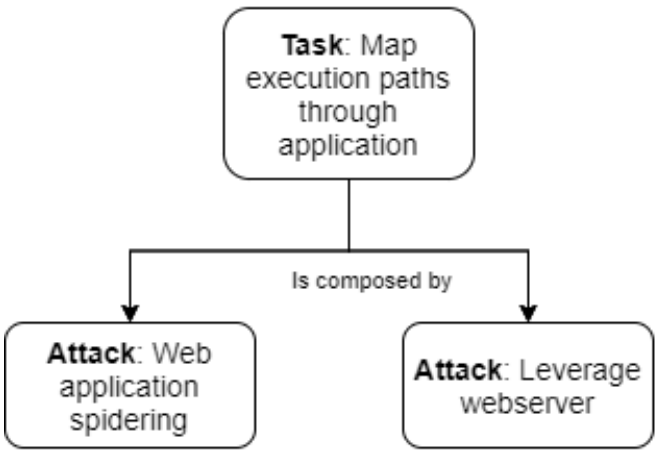

Figure 8.3: Composition of a Task

- Attack: An attack is a specific set of actions that can be performed on the target; for example, "Web application spidering" is an attack which includes many actions; it primarely represents a command that must be executed by a tool. For example: the "OWASP ZAP" tool can perform spidering of web applications, 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, whereas 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. In this work, the tasks were composed 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 has also 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 the corresponding Task has a relationship of acquired-knowledge with.

- Action: An action is atomic, meaning that it is the smallest operation that can be carried on the system. It has the following parameters:
  - "h-id": unique identifier;
  - "verb": it is the request method of the communication protocol used by the system (for web applications may be GET, SET, 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 in the action (it is expressed in a general form, independently from the used communication protocol);
  - "payload": it is the content of the action. It can also be empty (e.g. in GET requests).

  The Action entity is linked with a one-to-many relation with the Attack entity, because an action is specific for a certain attack. 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 viceversa).
- Response: The entity Response is specific with 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 eventually 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 with;
- "verb": the response phrase, used to emphasize the response result (according to the used communication protocol).

8.1.3 Relation diagram

The following diagram highlights the relations among the previously listed entities; here relations are described in an ontology-like fashion. Relations play a crucial role in an ontology, in order to write rules that can enable the inference process. In a relation, two or more entities are linked and each one plays a role in it.

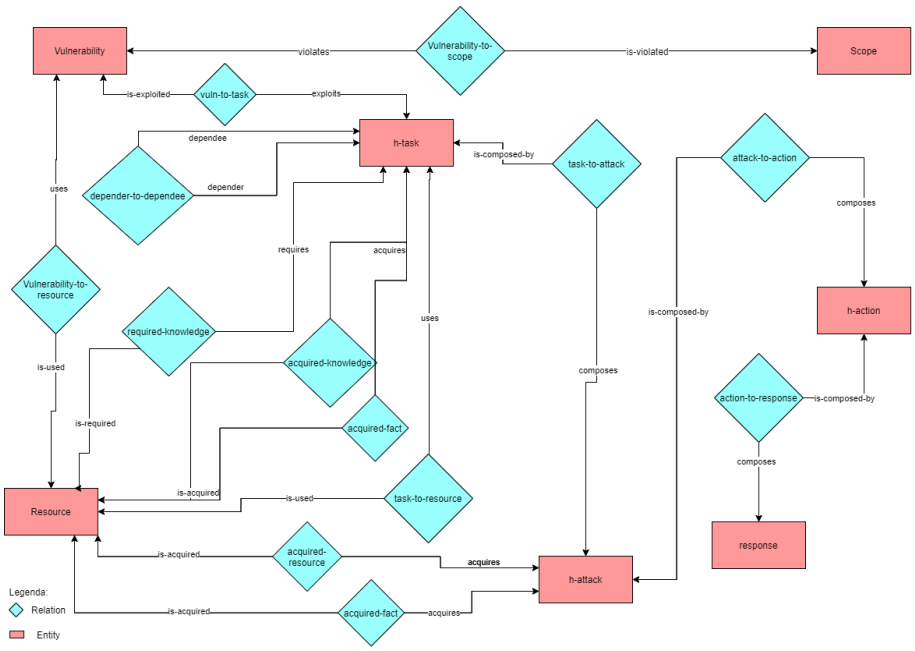

Figure 8.4: Relation Diagram

- Vulnerability-to-Scope: relation among Vulnerability and Scope entity. A vulnerability can violate one or more Scope and a Scope is violated by one or more Vulnerabilities. This relationship allows us to look for a vulnerability which violates a certain scope and, eventually, find the task that is linked to that specific vulnerability. This kind of relation should be retrieved from stored data, rather than being created with some inference rules.

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

- Vulnerability-to-Resource: relation between Vulnerability and Resource entity. A Vulnerability uses a Resource and a Resource can be used by one or more Vulnerabilities. This kind of relation should be retrieved from stored data, rather than being created with some inference rules.

- Depender-to-Dependee: relation among two H-Task entities. This relation allows the creation of an H-Task chain which are related in some way (e.g. an H-Task needs the knowledge about a Resource which is given by another H-Task) through the usage of Rules. This relation is fundamental when the penetration tester sets a Goal Task to reach: the system is able to calculate the chain of tasks which ends with the chosen task and, going backward in the chain, it can suggest the first task that must be executed by the penetration tester. The chain of tasks is updated each time the penetration tester executes the first task of the chain, allowing the system to guide the penetration tester to the execution of the final Goal task.

- Required-Knowledge: relation 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 relation should be retrieved by studying the covered Tasks (note that in this work these relations have been modeled after the OTG methodology).

- Acquired-Knowledge: relation among an H-Task and a Resource entity. A task, provides the knowledge about a Resource; in this way, it is pos-

sible to calculate the chain of tasks, in order to execute the final Goal task. Anyways, this relation only highlights the fact that a task, when is completed, gives the knowledge about a Resource; an additional information about the effective completion of the task is required, in order to rightly suggest the next task to execute in the task chain. (e.g. the H-Task "Identify application entry points" gives knowledge of Resource "All application entry points", but if the task has not been executed, then the knowledge has not been acquired yet).

- Acquired-Fact (H-Task): relation among an H-Task and a Resource. This relation 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 (in order to execute the final Goal H-Task).

- Task-to-Resource: it is a relation between an H-Task and a Resource entity, which states that an H-Task, in order to be completed, uses a specific Resource entity. This relation 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: relation among an H-Task and an H-Attack entity. As stated before, an H-Task is composed by more H-Attacks. This relations are formed by studying the tasks presented in the OTG.

- Acquired-Resource: relation 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 the H-Attack is linked to.

- Acquired-Fact (H-Attack): relation among a Resource and an H-Attack entity. This relation is used in order to state that an attack has been completed, and the knowledge about the related Resource has been acquired.

- Attack-to-Action: relation among an H-Attack and an H-Action entity. As stated in the previous chapter, an Attack is composed by multiple Actions.

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

8.1.4 Hacking Goal

It is important to define what an 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, such as "Testing for Local File Inclusion", or it can be a particular set of queries in order to Find all tasks related to vulnerabilities that violate the integrity scope. It is clear that, basing on the Hacking Goal, the system has to behave in a different way. In the first example, the system will perform different queries to find the task chain and will suggest the task (and attacks) that can be executed; in the second one, the system will give back to the penetration tester a certain set of tasks.

8.1.5 Rule diagrams

In this section the most relevant rules used to obtain relationships are described.

8.1.6 Attack acquires knowledge

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

In a more formal fashion, the rule is:

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

8.2 A toolset to build penetration testing datasets

The purpose of the presented tool is to capture the interaction of a user with a web application. It follows a client-server architectural style, with the inclusion of a middle-tier to serve as a proxy. Such proxy works, in fact, as an intermediary for all HTTP protocol requests/responses which are exchanged between client and server and is equipped with the business logic that allows users to manage the recording of a penetration testing session performed by an expert. The ability to make a dynamic recording of the interaction with any web application chosen by an end user represents an advancement in the state of the art and the main contribution provided by this work.

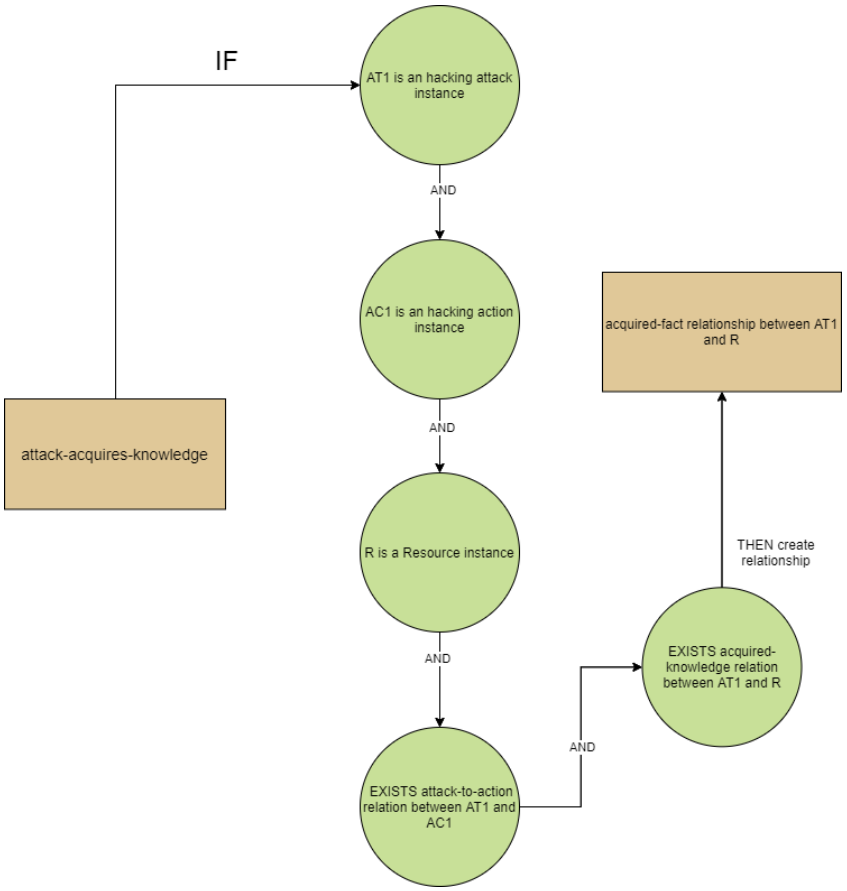

Figure 8.5: Rule for the relation Acquired-Fact among H-Attack and Resource

To achieve this goal, a protocol that can handle the sequence of actions performed by the user on any browser has been designed. Such protocol makes the recording of the actions taken with the mouse and keyboard possible, without knowing in advance how web pages are made. It also blends these actions together with some information directly collected by the proxy.

The protocol introduced in this section was designed to enable recording of those actions performed during a web application penetration test, as well as to store the network traffic that flows through client and server. The module that takes care of capturing both the actions and the network traffic is called “In-

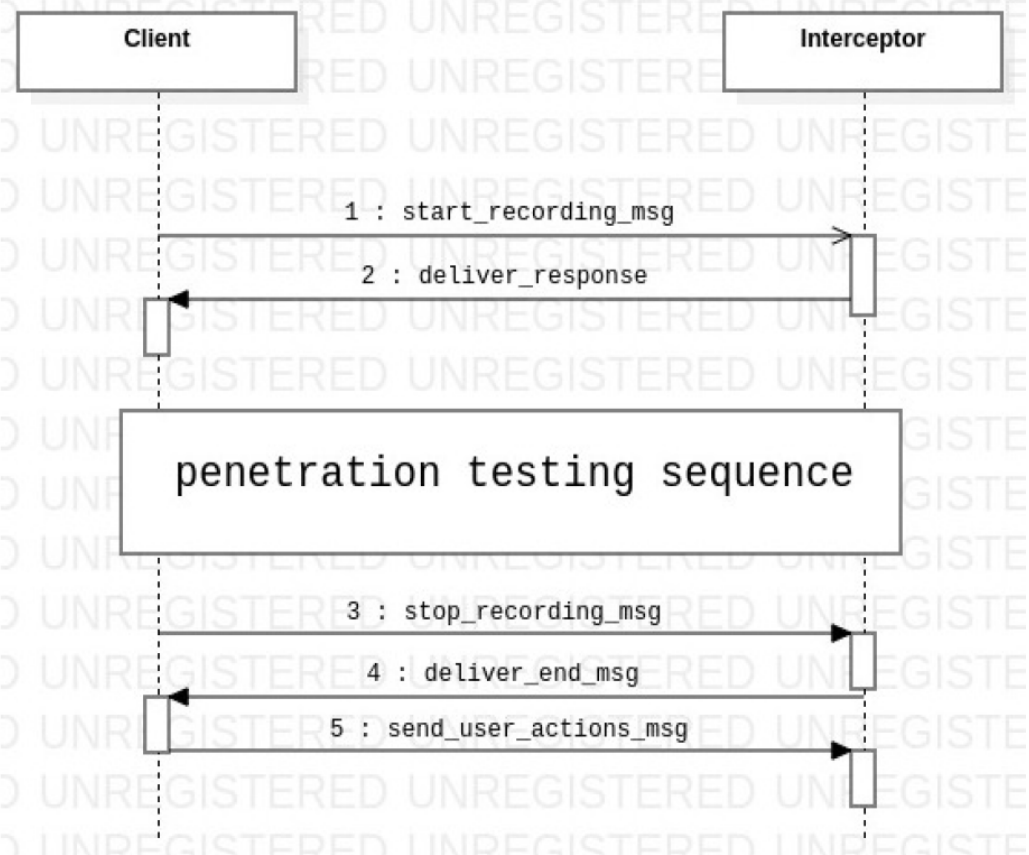

Figure 8.6: Session recording protocol

terceptor", which works as a proxy. To ensure that the actions recorded on the client are forwarded to the interceptor, which is required to store all network traffic by the end of the session , a specific exchange of messages has been designed that in order to collect the necessary information. An optimization has been made in order to store the actions only at the end of a session, instead of sending data along with each http request, using cookies, for example. Modern browser, in fact, allow to store a certain amount of data and send them in a single message. The protocol is composed by the following messages:

1. start_recording_msg. A GET request, containing the parameter "record" with value true. This message contains the page url from which the

recording of the episode provided by the expert will start. The message has the following form:

```
1    http://vulnerable_webpage:port?record=true
2
```

2. deliver_response. It is the answer provided by the web application containing the requested web page. After receiving this message, all interactions with the web application, (e.g. mouse and keyboard actions or new pages navigation), will be registered.

3. stop_recording_msg. A get request, with a parameter value "record" set to false. The message has the following form:

```
1    http://vulnerable_webpage:port?record=false
2
```

4. deliver_end_msg. It is the message that informs the client that the registration is over. However, it does not represent the end of communications between client and interceptor. In fact, the client will send another message, carrying the actions performed by the penetration tester on the web page.

5. send_user_actions_msg. It is the last message of the protocol, expected by the interceptor before being able to reconstruct the episode. It contains the actions collected during the recorded session. This message is forwarded autonomously by the client, unlike messages 1 and 3 which are sent via manual user requests on the browser, through an http request with the POST method.

The business logic of this system is effectively distributed between the client and the interceptor. In fact, the client is not just the tool used by the actor to interact with the web application but is equipped with a logic that allows the implementation of the protocol just described. There strategy adopted consists in manipulating the web pages through the interceptor. The latter, since it acts as a proxy, has the ability to manipulate all traffic to and from the client and can inject each web page directed to the user with javascript code necessary for recording the actions as well as implementing the protocol. The functionality of the client and interceptor components is hereby described:

- Client.

- Capture the interaction by mouse and keyboard on the page.
- Send the sequence of actions captured at the end of the recording.

- Interceptor.
  - Intercept http traffic between Client and Server.
  - Inject Javascript code into each HTML page directed towards the Client.
  - Generation of the episode on the file system after the end of the exchange.

8.2.1 Class Diagram

The Recorder object, described in Figure 5.8, contains the necessary methods to manage the recording of the events triggered by the mouse and keyboard. It resides on the web page that will be forwarded by the proxy.

8.2.2 Main functionality

This section briefly present the implementation of the main functionality provided by the presented platform. The Interceptor was implemented as an addon for mitmproxy, an open source interactive HTTPS proxy that provides a python API. Such API exposes the request and response methods which allow to modify messages, redirect traffic, visualize messages, or implement custom commands. The main functions implemented are:

- run_interceptor. Main method that instantiates the mitmproxy addon and sets up the reverse proxy.
- handle_request. Management of the requests coming from the client. It handles the requests that identify the start and stop of the recordings.
- handle_response. Main functionality that takes care of code injection that allow to capture the actions performed by the penetration tester.
- playback_episode. Functionality that allows to perform a playback, in the form of a video, the entire penetration testing session.

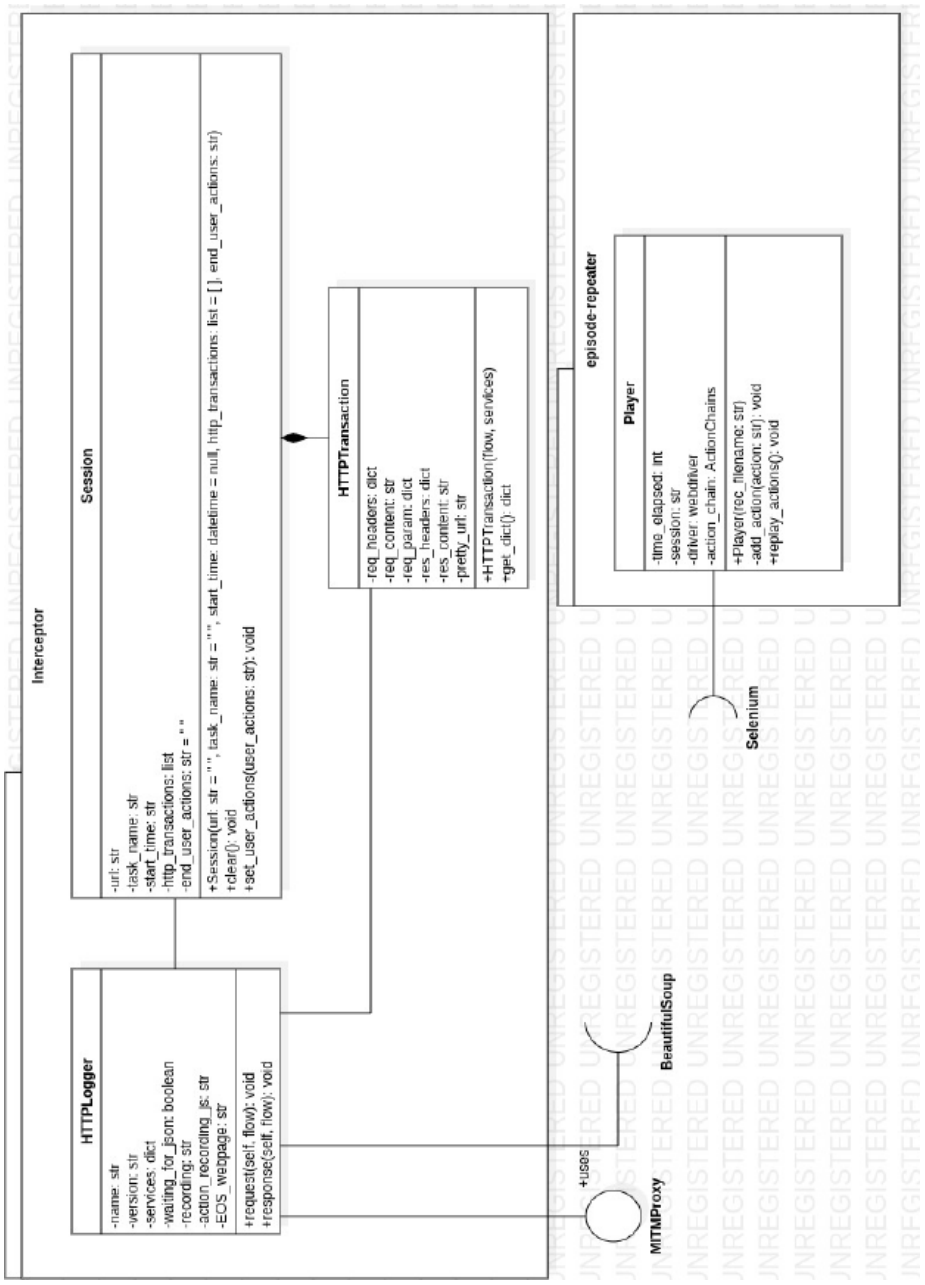

Figure 8.7: Domain model class diagram

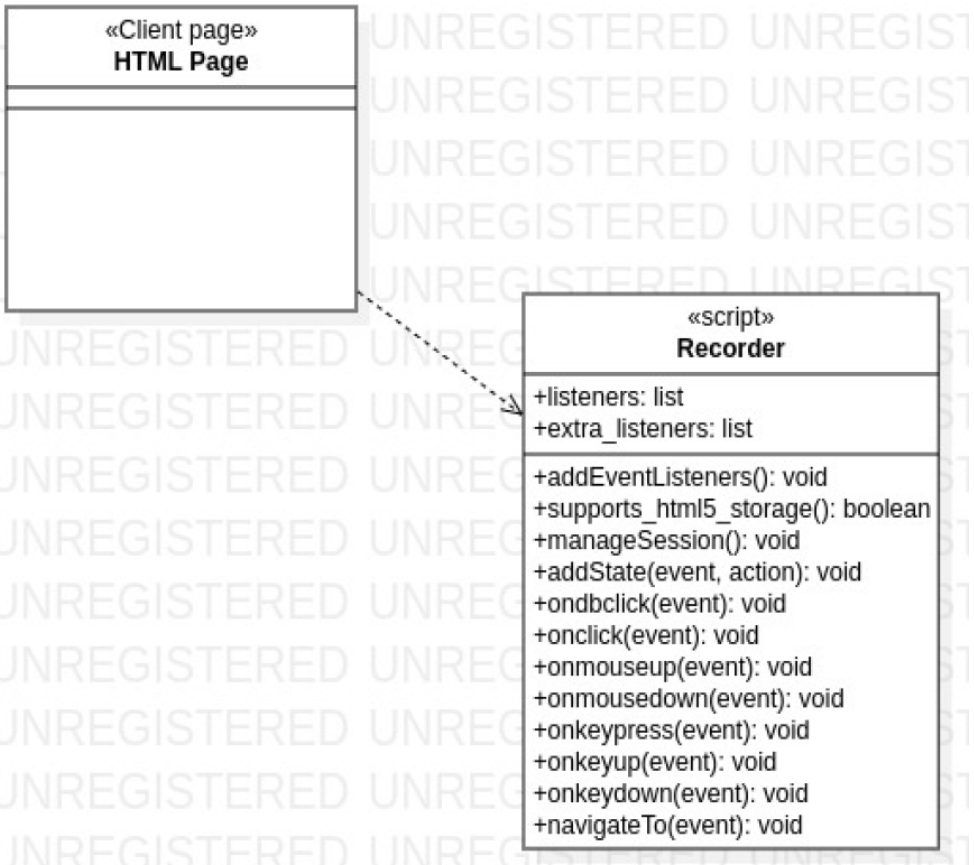

Figure 8.8: Recorder class diagram

Chapter 9

Honorable mentions

This chapter presents two additional contributions to the state of the art of cyber security, not necessarily related to the creation of intelligent agents for penetration testing. The first one regards the use of OS virtualization (containers) as building blocks to create cyber security exercises, usually hosted on complex virtual environments known as cyber ranges. Such exercises are used for educational purposes, for example in an university scenario, as well as for training expert security professionals. Such work argues the technical benefits that arise from the application of docker containers, discusses their limitations as well as proposes a way to overcome them. The second work presented in this chapter describes a distributed measurement solution to assess the cyber security exposure of an ICT Infrastructure. A way to define cyber security indicators through an automated and repeatable measurement process is provided. Results obtained in two different scenarios are discussed: a comparison of networks with different characteristics and a real-time monitoring of the defined metrics. Both works presented in this chapter are products of research experiments done on real-world infrastructure.

9.1 Capturing flags in a dynamically deployed microservices-based heterogeneous environment

The work presented in [81] tackle the challenges that from the introduction of OS virtualization to support cyber security exercises. A solution that allows to rely as much as possible on the use of containers is presented, as well as a way integrate them with legacy virtualization approaches when the vulnerabilities to be emulated do not lend themselves to a container-based implementation.

The Infrastructure-as-Code (IaC) paradigm is used, to enable automation of both provisioning and configuration of the emulated scenarios, as well as integrate heterogeneous virtualization technologies. After showing the design and implementation of the proposed solution, follows a discussion on how such approach leverages a cyber range instantiation platform, that can be designed and tested on a single laptop, before being deployed on an enterprise system infrastructure.

9.1.1 Design

9.1.2 OS Virtualization and Vulnerabilities

In this work, OS virtualization is considered as a mean 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;

- *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* [82] 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.

9.1.3 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**", the component that allows to implement and deploy the scenario is identified. It is made up by a type-2 hypervisor and its guest virtual machines. The "Virtual Scenario" lands on what is called 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.

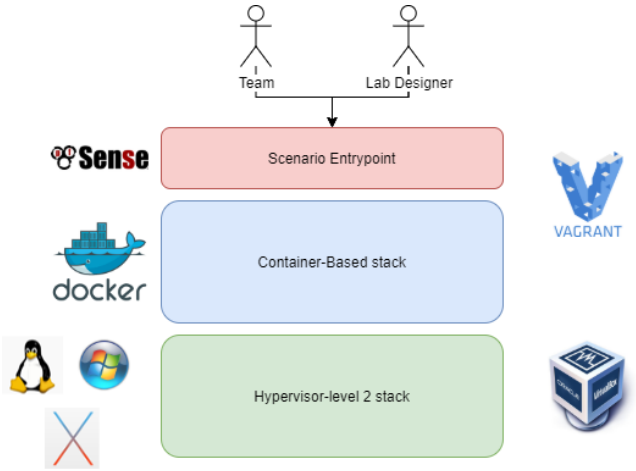

Figure 9.1: Virtual Scenario Architecture

9.1.4 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, the entry point of the scenario corresponds to 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, it is necessary to make sure all guest virtual machines are part of the same VPN. Then, segments of the network have to be populated with both containers and virtual machines. For instance, ideally, some services would handled by containers in one segment of the network and

only by virtual machines in another. These design choices are made in order to provide the scenario designers with enough flexibility. To tackle this issue, the concept of "**container as a router**" is introduces. 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 by 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.

9.2 A distributed security tomography framework to assess the exposure of ICT infrastructures to network threats

The work presented in [83], describes a framework to assess the exposure of ICT infrastructures to network threats base upon a measurement model. The purpose of the model is to provide the various assets of a company with a thorough guidance towards the minimization of the ICT infrastructure's exposure level. To do so, the starting point is the evaluation of the attack surface in order to extract metrics that will eventually become information needs. The latter can be targeted to several recipients in a company to define the security countermeasures to put in place. In fact, within a company, there is a need for information at different levels:

- Executive level;

- Business/Process level;

- Implementation/Operations level.

The execution level provides the mission priorities, the available resources and the overall risk tolerance to the business/process level, having a more direct feeling of the health status of the system. That said, it usually lacks an understanding of which risks are associated with a certain exposure profile. To help overcome this issue, the information need will be in the form of a risk assessment, by providing a finer granularity.
At the Business/Process level, the information need is the input for the risk management process. This allows to create a direct channel with the Implementation level in order to share the business needs and build a Profile. The Business level is constantly updated during the profile implementation process, taking advantage of the technical interpretation of the information need performed by the implementation level. This activity leads to the draft of an impact assessment, which is of fundamental interest to both the executive and the implementation level: it allows to keep the execution level updated about the overall risk management, but also to make the implementation/operations level aware of the business impact [84].
At this point, it is clear that the output of the model should be the information need, that is an evaluation of the extent of exposure to potential attacks, making specific references to the parts of the system involved. In this case, the purpose is to provide a set of concrete metrics that reflect the exposure to network threats.
The introduced model describes in a conceptual way how to quantify relevant attributes acquired by real on-field probes and how to translate them into the aforementioned metrics. Such measurement information model was constructed according to the guidelines proposed in [85], with the idea of keeping it flexible and easy to implement.
The model is structured in five levels as presented in Figure 9.2. Starting from the bottom, each level is characterized by a growing degree of information aggregation, up to information need, which is the goal.

- On the first level there are entities, engineering tools that help extract significant environmental attributes. Three sources of information are taken into account: a signature-based Intrusion Detection System, NetFlows and the target network infrastructure specifications. The first two provide files in JSON format and represent the sources of the measurement model. By combining them with the infrastructure design specification, the information needs of this work can be reached.

- The Attributes layer includes the relevant information selected by the

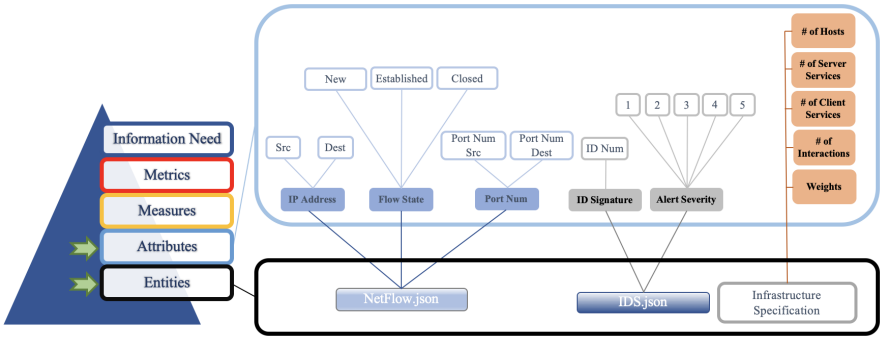

Figure 9.2: Model definition: Entities and Attributes

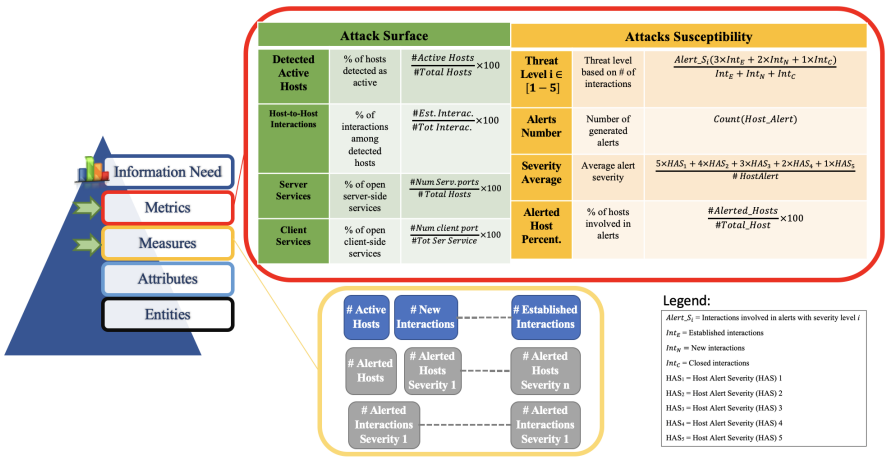

Figure 9.3: Model definition: Measures and Metrics

respective Entities. In the case of the NetFlow.json file: “IP address”, “Flow State” and “Port Num”. “IP address” attribute includes the source (“IP Src”) and the destination address (“IP Dest”) of each flow. “Flow State” can assume three values: “New”, if the data flow has just begun, “Established” if it is running, and “Closed”, if the flow is terminated. “Port Num” is the indication of the source (“Port Num Src”) and destination (“Port Num Dest”) ports used for each detected flow. The IDS.json file is the output of the IDS service, which refers to a database of signatures of well-known violations. The signatures are characterized by various attributes, among which the Signature ID and the Alert Sever-

ity have been selected. The Signature ID is the unique identifier of a violation. The Alert Severity indicates the alarm criticality level, ranging from 1 to 5 (*Critical*, *Dangerous*, *Medium Hazard*, *Low-Priority*, *Non-Priority*).

- A base or derived measure is the result of a measurement method, that is, a logical sequence of operations used to quantify one or several attributes with respect to a specified scale. The Measurements Layer in Figure 9.3 is made up of different quantities, *Active actions*, *Interactions*, *Ports*, *Host alerts*, *Host alerts severity*. It is important to note that all of the processes specified in the model are carried out in a well-defined time interval.

- Metrics are divided in two groups, as shown in Figure 9.3. The "attack surface" group includes the metrics that measure the physical and logical assets that could certainly be exploited by an attack. Such metrics have been defined as follows:

  - 'Detected Active Hosts' (DAH) quantifies the number of detected active hosts in an established time frame;
  - 'Host-to-Host Interactions' (HTHI) quantifies the number of host-to-host ("IP-Source", "IP-Dest") interactions, differentiated according to the flow state;
  - 'Ports' quantifies the total number of detected open ports in a given time interval. Well-known ports have been considered as server ports ("Server Service" – SS), the remaining are considered as client ports ("Client Service" – CS).

  The second group is associated with "Network Susceptibility" since it provides indications about weaknesses in the attack surface, not necessarily leading to an attack. The group in question comprises the following elements;

  – "Threat level" (TL) is a severity degree indicator in the range between 1 (Critical) and 5 (Non Priority). It is better detailed in Fig. 9.4;
  – "Alert Number" (AN) is the total alerts number in which the detected active hosts are involved in a well-defined time interval;

– "Severity Average" (SA) measures the average severity level of alerts detected in a given period of time. It is basically a weighted average of the alerts number based on their respective severity level, as indicated in Fig. 9.5. Speaking about alerts' severity, the model relies upon two different measures:

  1. "Host Alert Severity": it quantifies each type of alert detected for all the active hosts in a defined time interval, after a classification of the severity level for each alert has been performed;
  2. "Host-to-Host Alert Severity": it is similar to the previous measure, with the difference that in this case the process is done by host-to-host interaction and not by single host.

– "Alerted Hosts Percentage", indicating the number of hosts involved in an alert.

Metric Name	Threat Level *i (i* = 1,2,3,4,5*)*	**Formula**	TLi is computed through the weighted average of the interactions involved in an alert with severity level i. Weights are imposed on the interaction state, giving 3 (High) to established interactions (Int_E), 2 (Medium) to new interactions (Int_N), and 1 (Low) to closed interactions (Int_C). $\mathrm{TL_i} = \frac{Alert\_S_i(3 \times Int_E + 2 \times Int_N + 1 \times Int_C)}{Int_E + Int_N + Int_C}$
Description	It measures the average level of danger of the interactions involved in an alert with severity level *i*.		
Type	Technical		
Audience	Security Operations		
Question	What is the average hazard of the interactions involved in alert with severity level *i?*		
Answer	A positive real value is greater than zero but less than or equal to 3. A value of 3 indicates that all interactions involved in alerts are in an established state and therefore very dangerous, so there is maximum exposure to possible threats. The value 1 indicates that most of the interactions involved in alerts are in a closed state, therefore less dangerous.	**Units**	Average danger
		Frequency	Weekly, Monthly, Quarterly, Annually
		Targets	TLi values should evolve over time. Lower values are better as it means that exposure is limited, but not cancelled. Furthermore, the value of TL1 is more dangerous than that of TL5.
Sources	The NetFlow.json and the Infrastructure Specifications entities will provide information on which ports are used.	**Visualization**	Bar and Time Chart X-axis: Time (Week, Month) Y-axis: TLi

Figure 9.4: Metrics definition: Threat Level

Each metric provides useful information to measure network infrastructure exposure. Through the interpretation of this set of metrics, security countermeasures can be derived starting from the simplest ones, like the disposal of unsafe applications, to the more sophisticated ones, like Firewall rules, ACL lists and adoption of Intrusion Prevention Systems.

Metric Name	Severity Average
Description	Measures the average severity level of alerts detected in a given period of time.
Type	Technical
Audience	Security Operations
Question	What is the average severity level of alerts that occurred during the time period?
Answer	A non-negative integer value between 1 (Severity Average Low) and 5 (Severity Average High). The value "0" indicates that no safety warnings have been identified.
Sources	The IDS.json entity will provide information on the number of detected alerts and the classification of the alerts based on the severity level. Weights can be managed in the Infrastructure Specifications.

Formula	To calculate Severity Average (SA), a weighted average of the alerts number is made based on the severity level. Weight 5 is associated with Host Alert Severity (HAS) 1, 4 with Host Alert Severity 2 and so on. $SA = \frac{5 \times HAS_1 + 4 \times HAS_2 + 3 \times HAS_3 + 2 \times HAS_4 + 1 \times HAS_5}{Host\ Alerts\ \#}$
Units	Alerts for the time period
Frequency	Weekly, Monthly, Quarterly, Annually
Targets	The SA value should decrease over time, assuming perfect detection capabilities. A value of "0" indicates hypothetical perfect security because there were no security alerts. Obviously, it is desirable that the average level of severity is as low as possible.
Visualization	Time Chart X-axis: Time (Week, Month) Y-axis: SA

Figure 9.5: Metrics definition: Severity Average

Conclusions

This Thesis presented different approaches to the automation of offensive security practices, with the purpose of filling the gap between the way automated tools perform and the behavior of a security expert. Such distance is especially evident when one compares the results of the tests in terms of accuracy and efficiency.
A first approach, based on the application of a Reinforcement Learning model, shows that is possible to create an intelligent agent that performs discovery of Cross-Site scripting vulnerabilities. In particular, the framework of reinforcement learning allows agents to autonomously learn the same methodology that expert penetration testers would employ. Performance evaluations prove the accuracy in discovering such vulnerabilities can be significantly improved using an intelligent model that draws inspiration from human experience. The same goes for efficiency levels, noticing how the number of interactions with the target system consistently drops when the agent responds to a business logic that is distant from standard brute force.
An approach based on an ontology for web application penetration testing has been presented, with the same ontology being represented in the form of a knowledge graph. A system built on top of such an ontology allows to provide recommendations to a penetration tester, using predefined rules and an inference engine that outputs the most promising attack paths. Such an approach allows to address the scalability, which becomes an issue with growing action-state spaces.
Finally, a platform capable of creating datasets for web application penetration tests, is showed. Such a platform is based on a toolset that allows to collect ethical hackers' actions, such as browser interactions as well as generated network traffic. An encouragement to the research in the field of machine learning applied to cybersecurity is supposed to come from the release of open source datasets of hacking exercises.

Bibliography

[1] Eric S Raymond and Guy L Steele. The new hacker's dictionary, 1991.

[2] Charles C. Palmer. Ethical hacking. *IBM Systems Journal*, 40(3):769–780, 2001.

[3] Richard R Linde. Operating system penetration. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 361–368, 1975.

[4] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87, 2005.

[5] Kumar Shravan, Bansal Neha, and Bhadana Pawan. Penetration testing: A review. *Compusoft*, 3(4):752, 2014.

[6] Dafydd Stuttard and Marcus Pinto. *The web application hacker's handbook: Finding and exploiting security flaws*. John Wiley & Sons, 2011.

[7] Sergey Bratus, Iván Arce, Michael E Locasto, and Stefano Zanero. Why offensive security needs engineering textbooks. *Yale Law & Policy Review*, page 2, 2013.

[8] Tianlin Tim Shi, Andrej Karpathy, Linxi Jim Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3135–3144. JMLR. org, 2017.

[9] Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. *arXiv preprint arXiv:1802.08802*, 2018.

[10] Chunming Liu, Xin Xu, and Dewen Hu. Multiobjective reinforcement learning: A comprehensive overview. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(3):385–398, 2014.

[11] Warwick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement learning with parameterized actions. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[12] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

[13] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204, 2013.

[14] Fabrizio D'Amore and Mauro Gentile. Automatic and Context-Aware Cross-Site Scripting Filter Evasion. DIAG Technical Reports 2012-04, Department of Computer, Control and Management Engineering, Universita' degli Studi di Roma "La Sapienza", April 2012.

[15] Yong Fang, Cheng Huang, Yijia Xu, and Yang Li. Rlxss: Optimizing xss detection model to defend against adversarial attacks based on reinforcement learning. *Future Internet*, 11(8):177, 2019.

[16] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. Pomdps make better hackers: Accounting for uncertainty in penetration testing. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[17] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. Penetration testing== pomdp solving? *arXiv preprint arXiv:1306.4714*, 2013.

[18] Mohamed C Ghanem and Thomas M Chen. Reinforcement learning for intelligent penetration testing. In *2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*, pages 185–192. IEEE, 2018.

[19] Mohamed C Ghanem and Thomas M Chen. Reinforcement learning for efficient network penetration testing. *Information*, 11(1):6, 2020.

[20] Jonathon Schwartz and Hanna Kurniawati. Autonomous penetration testing using reinforcement learning. *arXiv preprint arXiv:1905.05965*, 2019.

[21] Jonathon Schwartz, Hanna Kurniawati, and Edwin El-Mahassni. Pomdp+ information-decay: Incorporating defender's behaviour in autonomous penetration testing. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 235–243, 2020.

[22] Fabio Massimo Zennaro and Laszlo Erdodi. Modeling penetration testing with reinforcement learning using capture-the-flag challenges and tabular q-learning. *arXiv preprint arXiv:2005.12632*, 2020.

[23] Fabio Massimo Zennaro and László Erdodi. Modeling penetration testing with reinforcement learning using capture-the-flag challenges: Trade-offs between model-free learning and a priori knowledge. *arXiv preprint arXiv:2005.12632*, 2021.

[24] Laszlo Erdodi, Åvald Åslaugson Sommervoll, and Fabio Massimo Zennaro. Simulating sql injection vulnerability exploitation using q-learning reinforcement learning agents. *arXiv preprint arXiv:2101.03118*, 2021.

[25] László Erdődi and Fabio Massimo Zennaro. The agent web model: modeling web hacking for reinforcement learning. *International Journal of Information Security*, pages 1–17, 2021.

[26] Ankur Chowdhary, Dijiang Huang, Jayasurya Sevalur Mahendran, Daniel Romo, Yuli Deng, and Abdulhakim Sabur. Autonomous security analysis and penetration testing. In *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*, pages 508–515. IEEE, 2020.

[27] Arcangelo Castiglione, Francesco Palmieri, Mariangela Petraglia, and Raffaele Pizzolante. Vulsploit: A module for semi-automatic exploitation of vulnerabilities. In *IFIP International Conference on Testing Software and Systems*, pages 89–103. Springer, 2020.

[28] Ryusei Maeda and Mamoru Mimura. Automating post-exploitation with deep reinforcement learning. *Computers & Security*, 100:102108, 2021.

[29] John A Bland, Mikel D Petty, Tymaine S Whitaker, Katia P Maxwell, and Walter Alan Cantrell. Machine learning cyberattack and defense strategies. *Computers & security*, 92:101738, 2020.

[30] Katia P Mayfield, Mikel D Petty, John A Bland, and Tymaine S Whitaker. Composition of cyberattack models. In *Proceedings of the 31st International Conference on Computer Applications in Industry and Engineering, New Orleans, LA*, pages 3–8, 2018.

[31] Katia P Mayfield, Mikel D Petty, Tymaine S Whitaker, John A Bland, and Walter A Cantrell. Component-based implementation of cyberattack simulation models. In *Proceedings of the 2019 ACM Southeast Conference*, pages 64–71, 2019.

[32] Katia P Mayfield, Mikel D Petty, Tymaine S Whitaker, Walter A Cantrell, Scott M Hice, Jeremiah McClendon, and Pedro J Reyes. Component selection process in assembling cyberattack simulation models. In *Proceedings of the International Conference on Security and Management (SAM)*, pages 168–174. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2019.

[33] Richard Elderman, Leon JJ Pater, Albert S Thie, Madalina M Drugan, and Marco A Wiering. Adversarial reinforcement learning in a cyber security simulation. In *ICAART (2)*, pages 559–566, 2017.

[34] Kalle Kujanpää, Willie Victor, and Alexander Ilin. Automating privilege escalation with deep reinforcement learning. *arXiv preprint arXiv:2110.01362*, 2021.

[35] William Blum. Gamifying machine learning for stronger security and ai models, Apr 2021.

[36] Erich Walter, Kimberly Ferguson-Walter, and Ahmad Ridley. Incorporating deception into cyberbattlesim for autonomous defense. *arXiv preprint arXiv:2108.13980*, 2021.

[37] Maxwell Standen, Martin Lucas, David Bowman, Toby J Richer, Junae Kim, and Damian Marriott. Cyborg: A gym for the development of autonomous cyber agents. *arXiv preprint arXiv:2108.09118*, 2021.

[38] Li Li, Raed Fayad, and Adrian Taylor. Cygil: A cyber gym for training autonomous agents over emulated network systems. *arXiv preprint arXiv:2109.03331*, 2021.

[39] Thanh Thi Nguyen and Vijay Janapa Reddi. Deep reinforcement learning for cyber security. *arXiv preprint arXiv:1906.05799*, 2019.

[40] John Mern, Kyle Hatch, Ryan Silva, Jeff Brush, and Mykel J Kochenderfer. Reinforcement learning for industrial control network cyber security orchestration. *arXiv preprint arXiv:2106.05332*, 2021.

[41] Stefan Niculae. Reinforcement learning vs genetic algorithms in game-theoretic cyber-security. 2018.

[42] Ahmad Hoirul Basori and Sharaf Jameel Malebary. Deep reinforcement learning for adaptive cyber defense and attacker's pattern identification. In *Advances in Cyber Security Analytics and Decision Systems*, pages 15–25. Springer, 2020.

[43] Rohit Gangupantulu, Tyler Cody, Abdul Rahman, Christopher Redino, Ryan Clark, and Paul Park. Crown jewels analysis using reinforcement learning with attack graphs. *arXiv preprint arXiv:2108.09358*, 2021.

[44] Rohit Gangupantulu, Tyler Cody, Paul Park, Abdul Rahman, Logan Eisenbeiser, Dan Radke, and Ryan Clark. Using cyber terrain in reinforcement learning for penetration testing. *arXiv preprint arXiv:2108.07124*, 2021.

[45] Ryan Christian, Sharmishtha Dutta, Youngja Park, and Nidhi Rastogi. Ontology-driven knowledge graph for android malware. *arXiv preprint arXiv:2109.01544*, 2021.

[46] Yifan Wang, Zhi Sun, and Ye Han. Network attack path prediction based on vulnerability data and knowledge graph.

[47] Weilin Wang, Huachun Zhou, Kun Li, Zhe Tu, and Feiyang Liu. Cyber-attack behavior knowledge graph based on capec and cwe towards 6g.

[48] Milan Cermak and Denisa Sramkova. Granef: Utilization of a graph database for network forensics. 2021.

[49] Injy Sarhan and Marco Spruit. Open-cykg: An open cyber threat intelligence knowledge graph. *Knowledge-Based Systems*, page 107524, 2021.

[50] Kexiang Qian, Daojuan Zhang, Peng Zhang, Zhihong Zhou, Xiuzhen Chen, and Shengxiong Duan. Ontology and reinforcement learning

based intelligent agent automatic penetration test. In *2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, pages 556–561. IEEE, 2021.

[51] Damian Hermanowski and Rafał Piotrowski. Network risk assessment based on attack graphs. In *International Conference on Dependability and Complex Systems*, pages 156–167. Springer, 2021.

[52] Kabul Kurniawan, Andreas Ekelhart, and Elmar Kiesling. An att&ck-kg for linking cybersecurity attacks to adversary tactics and techniques.

[53] Aviad Elitzur, Rami Puzis, and Polina Zilberman. Attack hypothesis generation. In *2019 European Intelligence and Security Informatics Conference (EISIC)*, pages 40–47. IEEE, 2019.

[54] Erik Hemberg, Jonathan Kelly, Michal Shlapentokh-Rothman, Bryn Reinstadler, Katherine Xu, Nick Rutar, and Una-May O'Reilly. Linking threat tactics, techniques, and patterns with defensive weaknesses, vulnerabilities and affected platform configurations for cyber hunting. *arXiv preprint arXiv:2010.00533*, 2020.

[55] Siwar Kriaa and Yahia Chaabane. Seckg: Leveraging attack detection and prediction using knowledge graphs. In *2021 12th International Conference on Information and Communication Systems (ICICS)*, pages 112–119. IEEE, 2021.

[56] Peter E Kaloroumakis and Michael J Smith. Toward a knowledge graph of cybersecurity countermeasures. Technical report, Technical report, 2021.

[57] Ge Chu and Alexei Lisitsa. Ontology-based automation of penetration testing, 2020.

[58] Taiana Stepanova, Alexander Pechenkin, and Daria Lavrova. Ontology-based big data approach to automated penetration testing of large-scale heterogeneous systems.

[59] Valdemar Švábenskỳ, Jan Vykopal, Pavel Seda, and Pavel Čeleda. Dataset of shell commands used by participants of hands-on cybersecurity training. *Data in Brief*, page 107398, 2021.

[60] Valdemar Švábenskỳ, Jan Vykopal, Daniel Tovarňák, and Pavel Čeleda. Toolset for collecting shell commands and its application in hands-on cybersecurity training. 2021.

[61] Reza M Parizi, Kai Qian, Hossain Shahriar, Fan Wu, and Lixin Tao. Benchmark requirements for assessing software security vulnerability testing tools. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 825–826. IEEE, 2018.

[62] Owasp vulnerable web applications directory https://owasp.org/www-project-vulnerable-web-applications-directory/, Oct 2013.

[63] Port swigger web security academy https://portswigger.net/web-security.

[64] Fernando Román Muñoz, Iván Israel Sabido Cortes, and Luis Javier García Villalba. Enlargement of vulnerable web applications for testing. *The Journal of Supercomputing*, 74(12):6598–6617, 2018.

[65] Doug Rathbone. Doug rathbone. leadership and internet musings from the perl generation..., Oct 2013.

[66] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[67] Martijn Van Otterlo and Marco Wiering. Reinforcement learning and markov decision processes. In *Reinforcement Learning*, pages 3–42. Springer, 2012.

[68] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[69] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[70] Michael Wooldridge. Intelligent agents. *Multiagent systems*, 6, 1999.

[71] Hyrum S. Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static pe machine learning malware models via reinforcement learning, 2018.

[72] Cangshuai Wu, Jiangyong Shi, Yuexiang Yang, and Wenhua Li. Enhancing machine learning based malware detection model by reinforcement learning. In *Proceedings of the 8th International Conference on Communication and Network Security*, pages 74–78, 2018.

[73] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[74] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and software Technology*, 46(5):337–342, 2004.

[75] Owasp benchmark project. `https://owasp.org/www-project-benchmark/`. Accessed: 2020-10-19.

[76] Richard Amankwah, Jinfu Chen, Patrick Kwaku Kudjo, and Dave Towey. An empirical comparison of commercial and open-source web vulnerability scanners. *Software: Practice and Experience*, 50(9):1842–1857, 2020.

[77] William J Youden. Index for rating diagnostic tests. *Cancer*, 3(1):32–35, 1950.

[78] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[79] Elie Saad and Rick Mitchell. Owasp web security testing guide, Dec 2020.

[80] Cwe glossary, Apr 2018.

[81] Francesco Caturano, Gaetano Perrone, and Simon Pietro Romano. Capturing flags in a dynamically deployed microservices-based heterogeneous environment. In *2020 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–7. IEEE, 2020.

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

[83] MA Brignoli, AP Caforio, F Caturano, M D'Arienzo, M Latini, W Matta, SP Romano, and B Ruggiero. A distributed security tomography framework to assess the exposure of ict infrastructures to network threats. *Journal of Information Security and Applications*, 59:102833, 2021.

[84] Matthew P Barrett. Framework for improving critical infrastructure cybersecurity. *National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep*, 2018.

[85] International Organization for Standardization/International Electrotechnical Commission et al. Systems and software engineering—measurement process. *ISO/IEC 15939: 2007*, 1, 2007.