Reinforcing Penetration Testing Using AI

Alessandro Confido
PhD in Artificial Intelligence
Mediterranea University
89134 Reggio di Calabria, Italy
Alessandro.Confido@unicampus.it

Evridiki V. Ntagiou
Internal Research Fellow
OPS-GDA, ESA-ESOC
64293 Darmstadt, Germany
Evridiki.Ntagiou@esa.int

Marcus Wallum
Operations Data Systems Manager,
OPS-GDA, ESA-ESOC
64293 Darmstadt, Germany
Marcus.Wallum@esa.int

***Abstract*—In a space mission operations context, assets of very high value are operated by several data systems. The security and resilience of these data systems has become a primary concern, due to the increasing ubiquity of the internet and mutability of the threat landscape. Penetration testing is a well established method to identify system security weaknesses, however, is typically high-cost and effort-intensive. The European Space Agency has developed a prototype automated penetration testing framework, called PenBox. Given the volume of parameters and possibilities for the PenBox automated test execution sequence, applied Machine Learning theory presents an interesting opportunity for enhancement. This paper presents the approach and results obtained towards research on the integration of Artificial Intelligence (AI) techniques, in particular Reinforcement Learning (RL), to PenBox. The adopted approach to frame this case study involves the Q-learning paradigm that seeks to learn a policy that maximizes the total reward, according to the quality associated with the certain action to take; the reward itself is the pivotal element that shapes the intelligent agent's behaviour through the trial-and-error theory, without any human interaction. To build this representative space, an existing network simulator was used and further adopted to improve representativeness and alignment with possible PenBox actions. In this paper, we highlight how Deep Q-Learning ensures a certain level of randomization, thus providing the basis for future evolution of the developed AI model to suit the real-world autonomous penetration testing software, with the aim of optimizing the process in terms of performance and cost effectiveness, and enabling an innovative autonomous capability for optimal attack path definitions without human in the loop.**

TABLE OF CONTENTS

1. INTRODUCTION

A conventional way of assessing security takes a defensive position, analysing and hardening systems from the perspective of a defender. An offensive position provides a complimentary proactive perspective. Penetration testing, often known as ethical hacking, is the process of conducting approved simulated cyber-attacks on a computer system in order to find flaws and assess overall security. Penetration testing is a time-consuming and expensive operation that necessitates developing an in-depth understanding of the target system as well as the various attacks that may be launched against it. As a result, pentesting requires specialists that can meticulously explore a system and identify known and, preferably, still undisclosed vulnerabilities in order to offer meaningful insights.

Some elements of penetration testing have been automated using software programs, nevertheless they generally boil down to tools that perform specific tasks under the supervision of a human user. Traditional artificial intelligence approaches, such as scheduling, were also used in the hopes of further automating penetration testing by generating attack plans; however, human input is still required to model the context and target system, as well as to draw conclusions about the actual weaknesses.

Artificial intelligence and machine learning breakthroughs may provide a method to overcome some of the existing constraints in automated penetration testing. The Reinforcement Learning methodology, in particular, has been shown to be a flexible and successful technique for addressing complicated issues involving agents attempting to act effectively in a given environment. RL applications include a wide range of techniques and approaches with various levels of computational and data complexity, as well as procedures that need little expert modeling.

Paper structure

This paper is organized as follows.

Section 1 traces a brief history of pentesting, indicating basic ideas and features.

Section 2 outlines the general purposes of automatic penetration testing and security analysis, contemplating the characteristics of the already developed software, PenBox;

Section 3 focuses on the analysis of the chosen ML algorithm. We will start with an overview of the general principles of RL, then shifting to the phases that allowed us to implement the application and the technologies underlying the current version will be described;

Section 4 shows the tests about the algorithm's efficiency and the linked results are explained;

Section 5 concerns the contribution of the thesis work with final discussion and future proposals.

State of the Art

Schneier [8] suggested an attack tree model for identifying an attack sequence, in which the root node represents the attack's objective and the child nodes reflect the criteria for meeting the parent nodes. The attack tree model's drawback is that it only discovers single-objective attack paths, which does

978-1-6654-3760-8/22/$31.00 ©2022 IEEE

2022 IEEE Aerospace Conference (AERO) | 978-1-6654-3760-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/AERO53065.2022.9843459

not account for multi-objective cases. Sheyner created an attack graph model to handle this challenge, which represents a collection of probable penetration scenarios for a certain network, with each scenario being a sequence of intrusive actions.

Vulnerability analysis was implemented by Roberts [9], who merged personal psychology with the attack graph model. The attack graph model is frequently used in security evaluations, but due to the state scaling problem, it can only be employed in tiny network penetration testing scenarios. Obes [10] solved this challenge by converting the penetration scenario into the plan domain definition language (PDDL) and using a traditional planning technique to obtain attack paths. The "problem" and "domain" files are two aspects of the penetration scenario, with the first describing network configuration information and the second containing vulnerability and action information. Khan and Parkinson [11] implemented automated vulnerability assessment using PDDL. Although PDDL has had a lot of success with the commercial product "Core Impact" [12], it still has significant limitations, such as requiring precise information about a network scenario.

Sarraute [13] used Partially Observable Markov Decision Processes (POMDP) to define the attack planning problem in order to include action uncertainty into attack path development in a network penetration testing scenario. POMDP was integrated into industrial control systems by Kurniawati [14], who was inspired by prior work and aimed to automatically verify the security of industrial control systems. Furthermore, Shmaryahu used a conditional planning [15] tree technique to design attack paths after modeling penetration testing as a partially observable stochastic problem.

Sarraute [16] combined probability with classical planning algorithms to determine the best attack paths in nondeterministic conditions, rather than creating new models. Even when uncertainty is taken into account, solving POMDP for a large network remains impossible (a network with 20 computers is already unfeasible for the POMDP solver). To address this issue, Sarraute developed a decomposition technique that splits a vast network into smaller ones based on network structure and handles each one using POMDP.

In spite of its academic success, the POMDP model is based on a strict assumption in which the network structure and software configuration remain the same between any successive penetration tests, making it difficult to apply to practical penetration scenarios. From an engineering perspective, there are some more research efforts that have contributed to the field of automated attack strategy. To execute a protocol-oriented security analysis, Samant [17] provided an efficient, reliable, and automated testing tool.

To perform modern automated penetration testing, Stefinko and Piskuzub [18] created an intelligent system consisting of an inference engine and a skillset. Backes [19] applied mitigation analysis to implement a complete security evaluation in a simulated penetration test scenario, however they didn't have a firm theoretical basis. From the viewpoint of resource limitations, Steinmetz [20] suggested an attack planning algorithm capable of dealing with this challenge.

In this work of research, instead of using fixed generic attack trees as utilised in the previously mentioned solutions (and in PenBox), which do enable accomplishing an automated penetration test, we investigate upon the capability to derive a dynamic attack tree utilising AI/ML to drive automated testing. Despite the value of past research, the attack planning strategies, as well as our current approach requires prior knowledge about almost the entirety of the network configuration, are still incapable of accomplishing penetration testing without prior knowledge. For example, PDDL-based approaches require a full featured network topology and host status information, while POMDP-related methods require portions of that information. Without prior knowledge, none of these approaches can find attack paths. There are two problems in dealing with realistic penetration testing: how to drive an agent to penetrate automatically and intelligently, and how to determine the optimum attack actions' path in a given situation.

2. PRELIMINARIES

Penetration testing

An important part of a security assessment is the penetration test [1] (or informally "pentest"). It is an operational process of assessing the security of a system or network that simulates an attack by an attacker. The analysis includes several phases and aims to highlight the weaknesses of the platform by providing as much information on the vulnerabilities that have been enumerated and exploited to achieve a particular compromise goal and defeat or violate the security objectives of the system. The analysis is conducted from the point of view of a potential attacker and consists in exploiting the vulnerabilities detected in order to obtain as much information as possible to access the system illegally. A penetration test can help determine if the system's defenses are sufficient or if it has vulnerabilities by listing in this case which defenses the test has defeated.

The targets of a pen test are divided into four areas: communication security, spectrum / signal security, physical security, and logical security. It can be performed by simulating two different types of attack: from the inside and from the outside. The amount of information pre-shared between attacker and target is classified through a gray scale, therefore we will talk about white, gray and black box pen tests which reflect the degree of prior knowledge and insight available to the tester. There are several frameworks, open source or commercial, used to run pen tests. Some examples are Kali Linux [2], BackBox [3], Pentest Box [4], Metasploit Framework [5] and Burp suite [6]. There are attempts to outline general guidelines [7] for executing a pen test but, commonly each test is prepared and launched against a specific target, so we usually proceed on a case-by-case basis.

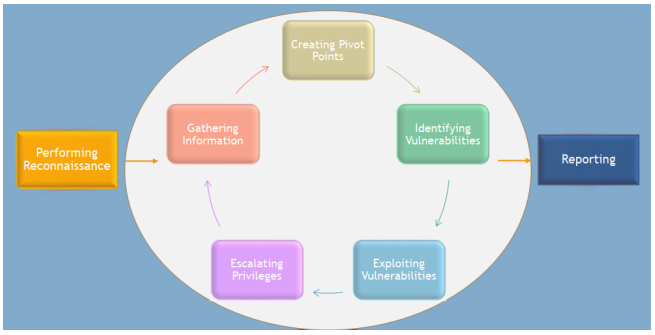


Figure 1. Penetration testing life cycle

The penetration test consists of some main steps (see Figure 1), which are:

• *Pre-engagement Interactions*, the goal of this phase is to define the rules of engagement: the purpose of the test, the timing, the targets to be verified and the cost. It is a sort of contract where all the terms of the test are established (start and end date, IP ranges, social engineering limits, etc.)

• *Intelligence Gathering*, the goal is to collect as much information as possible on the target and produce a document useful for planning the test strategy. It can be performed with three difficulty levels:

- Level 1: Data collection through the use of automatic tools

- Level 2: In addition to level 1 data, it includes the implementation of manual analysis to obtain information on the physical structure, organization, relationships with external parties, technical information

- Level 3: In addition to the data of levels 1 and 2, it includes a more in-depth analysis of the information acquired (OSINT) also through social networks.

We distinguish two types of research approaches:

- Passive, which is developed by collecting information from open sources (OSINT: newspaper, website, discussion group, social networking, blog and other open sources) or by using third-party tools and services that do not "attack" the target (tools for Information Gathering: WHOIS, Shodan, T-Shark).

- Active, in which we try to discover information directly from the target by exploiting the techniques of Social Engineering (Phishing, Pretexting, False job offers, etc.) or by interrogating the target with non-"invasive" tools.

• *Threat Modeling*, serves to define a threat modeling system useful for performing a correct penetration test. It is valid for the tester and for the recipient, as it highlights the organisation's risk appetite and priorities. The produced model must be documented and delivered together with the final report, as the results of the final report are closely linked to the Threat Model, as it highlights the specific risks of the organization.

• *Vulnerability Analysis*, is the process that allows you to discover vulnerabilities, systems and applications, which can be exploited by an attacker to steal information. It is divided into two phases: Identification (active and passive, which depends or not on the interaction with the components to be tested) and Validation (which consists in the analysis of the results obtained from the previous step in order to facilitate their identification). Not all vulnerabilities are cost-effective exploitable, or even not all of them can be discovered. When a new vulnerability is found without a solution it is called zero day(Tools: Nmap, Wireshark).

• *Exploitation*, is the phase that focuses exclusively on creating access points to a system or resource by bypassing security restrictions. If the previous steps have been performed correctly, the latter can be planned well and will allow for very precise results. The goal is to identify the main entry point into the organization and the most important target resources. If the vulnerability analysis phase has been carried out correctly, we will have a list of strategic defobjectives on which to carry out the exploit.

Often the exploitation phase must take into account the security and alert systems of computer systems, such as antivirus, firewall, intrusion detection and prevention systems, etc. (Examples: Metasploit, Burpsuite)

• *Post Exploitation*, the purpose of this phase is to assign a value to indicate the level of compromise of the machine and to maintain control of the machine for later use. The value is determined by the importance of the data stored on it and the usefulness it can have to further compromise the network. The methods used are intended to help the tester identify and document sensitive data, configuration settings, communication channels and relationships with other network devices that can be used to gain additional network access and set up one or more methods for accessing the machine later. An example of post exploitation is pillaging, which allows information to be obtained from hosts identified in the pre-evaluation phase. This information can be acquired for the purpose of the penetration test, or to obtain further access to the network.

• *Reporting*, the final purpose of which is to highlight the weaknesses of the analysed system, providing as much information on the vulnerabilities that have allowed unauthorized access. In order to communicate the objectives, methods and results of the test conducted, a detailed report is drawn up. Usually the document is divided into two main parts:

o The Executive Summary which includes the specific objectives of the Penetration Test and the results obtained,

o The Technical Report where the technical details of the test and all agreed aspects are described Penetration testing is a well-established method to identify system security weaknesses, however, is typically high-cost and effort-intensive. Testing results are often not easily interpretable for project decision makers, in particular the impact of a test result in relation to the overall system and supported mission context. To address this drawbacks the European Space Agency (ESA) developed a prototype automated penetration testing framework, PenBox.

PenBox

The European Space Agency (ESA) and the European space industry operate assets of very high tangible and intangible value. These assets are handled and operated through a large number of data systems. With the increasing pervasiveness of digital media and the internet, the security and robustness of these data systems have become extremely important. Secure software and system engineering is therefore a primary concern throughout the agency.

In order to integrate automated security penetration testing into the development life cycle of these systems a prototype tool 'PenBox' was developed [21]. This prototype, a tool called PenBox (Penetration testing in a Box), attempts to execute attack patterns to reveal weaknesses and vulnerabilities on space systems in their environment. The tool intends to complement the preceding development of security engineering building blocks (security requirements definition, risk assessment guidance) with an automated security testing capability. PenBox, although mainly intended for application at the ground system-level integration and testing phase, could also be utilised in any development/pre-production environment, and as part of an agile and iterative development process. The software architecture is shown in Figure 2.

The objective of this tool is to provide a mean of self-education to ground segment systems users and stakeholders, to make them understand the steps that are necessary to successfully execute the attacks and the impact that it has on the

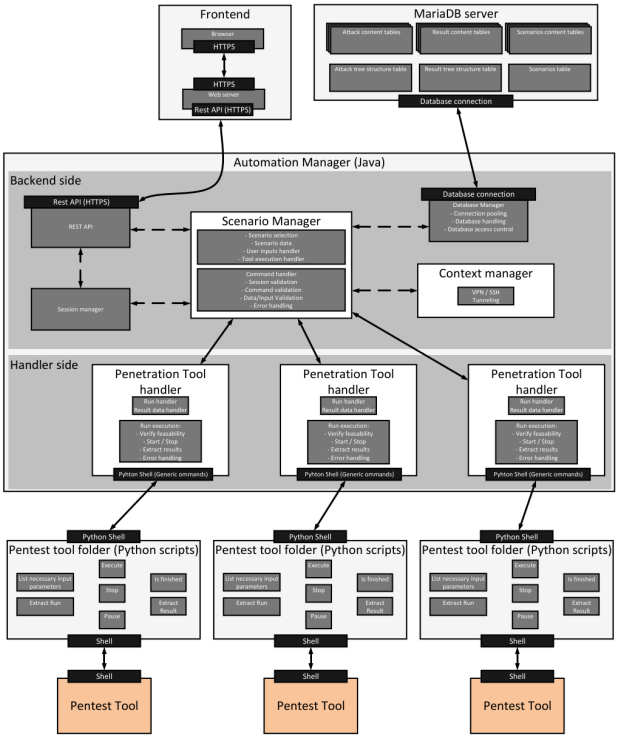


Figure 2. PenBox architecture (from PenBox Documentation)

chain. This called for a user-friendly but powerful graphical tool to carry out the above tasks. This makes security testing more accessible to non-experts, raises awareness on security and the impacts of weak systems and software at the mission level and allows them to actually attack a system and follow its attack patterns.

PenBox is capable of automatically discovering assets, finding their vulnerabilities, opening backdoors on vulnerable victim machines, access them and attack further systems from a victim's position. The PenBox is integrated with a clone of ESA's Ground Segment Reference Facility, enhancing the facility with a security testing capability.

The PenBox currently consists of five space-oriented scenarios including commanding a spacecraft, obtaining unauthorised access to TM flow, denial of service on mission-critical data flows. The PenBox not only detects and exploits flaws but also interacts with the systems to actually show the impact of an attack on the running system. This is a key distinguishing aspect compared to a standard test as it ties immediately the attack to a mission-level impact. This makes the use of the PenBox extremely relevant for the space domain user.

The PenBox moves beyond a standard vulnerability scan because it follows a scenario where each action is taken based on the direct feedback loop from the results obtained by the previous actions, including exploitation and supports mapping of test results to system security requirements verification logic, highlighting warnings and violations to the user.

Typically, scenarios start by finding information, discovering assets, their vulnerabilities and exploiting them until a point is reached where the scenario becomes specific to its goal. The PenBox is capable of reaching strategic positions in the network or on machines at which each scenario may "intervene" and perform scenario-specific actions to obtain and assess a success. Even though currently most implemented scenarios use a quite common schema, it is possible due to the modular code structure to implement heterogeneous scenarios not necessarily following existing attack patterns. This is one of the several extensions to the PenBox perceived for the future.

The definition of scenario is quite flexible and can be adjusted to implement any kind of attack. A scenario may inject specific actions or introduce custom tools to be executed at any time. Scenarios can loop several times on a phase and increment at each overpass the aggression level of the attacks. On the contrary, scenarios may perform more stealthy actions by limiting the executed tools.

The goal of the PenBox is to execute several penetration testing tools following a given attack scenario. This not only requires to be able to execute those tools in the correct order but also requires that each tool is able to retrieve input parameters from results of tools that preceded in the execution chain. This should also consider preceding results that become multipliers in follow-up tools. For example, a list of N IP addresses requires the execution of N times the follow-up tools.

The main technical challenge of the PenBox design is the automation and chaining of existing penetration testing tools. The first objective is to be able to interact homogenously with any of the penetration tools through technical means to execute those tools with the proper configuration and inputs. The second objective, once it is possible to interact with the tool is to be able to extract the relevant information from the tool's results.

Those objectives are particularly challenging since the penetration testing tools can vary a lot in terms of technology they use and interfaces they provide. Finding a common way to interact and obtain results for each of those existing tools was the main driver of this design. Penetration testing tools may use various and very heterogeneous technologies and ways to interact with the user. Some tools may only provide a Graphical User Interfaces (GUI), others may be command lines or configuration files.

With such an ambitious objective, it was improbable that all aspects of every imaginable attack scenario and existing and future pentest tool could be taken into account during the design and development of this work of research. For this reason, where possible, the design and implementation made sure to remain flexible enough to additional scenario implementation or tool integration in the future.

Considering the volume of parameters and possibilities for the PenBox automated test execution sequence as well as analysis of the results obtained, applied Machine Learning theory presents an interesting opportunity to enhance this software, aiming to remove reliance on human intuition in decision making and enables a dramatic increasing of performances, selecting only relevant tools which provide non-redundant results.

Overview of ML techniques

Machine Learning (ML) is a branch of artificial intelligence [22] based on the idea that systems can learn from data, identify patterns and make decisions with minimal human

intervention.

Since there are so many distinct types of Machine Learning systems, it's helpful to categorize them into broad groups depending on:

• Whether or not they're trained with human supervision (Supervised, Unsupervised, Semi-supervised and Reinforcement Learning);

• Whether they can learn progressively on the fly (online versus batch learning);

• If they function by comparing new data points to prior data points or by detecting patterns in the training data and producing a prediction model (instance-based versus model-based learning).

These criteria are not mutually exclusive; you can mix and match them in the most useful way. Though both supervised and reinforcement learning [23] use mapping between input and output, unlike supervised learning where the feedback provided to the agent is a correct set of actions for performing a task, rewards and punishments are used as indicators for desirable and harmful behaviour in reinforcement learning. Reinforcement learning differs from unsupervised learning in terms of objectives. While the goal in unsupervised learning is to find similarities and differences between data points, in the case of reinforcement learning the goal is to find a suitable action model that would maximize the total cumulative reward of the agent. The Figure 3 illustrates the action-reward feedback loop of a generic RL model.

Figure 3. Action-Reward-Observation cycle.

Some key terms that describe the basic elements of an RL problem are:

1. Agent — The component that observe the environment and takes decisions, so it must learn to anticipate human needs;
2. Environment — Representation of the world in which the agent operates;
3. State — Current situation of the agent;
4. Reward — Feedback from the environment;
5. Policy — Method to map agent's state to actions, nothing more than the algorithm used by the software agent to determine its actions
6. Value — Future reward that an agent would receive by taking an action in a particular state

There may not be any positive rewards at all; for example, the agent may carry out a series of unfruitful actions, getting a negative reward at every time step.

The reinforcement learning paradigm provides a flexible framework for modeling and solving complicated problems with general-purpose learning algorithms. The challenge of an agent trying to learn the best behaviour or policy in a given environment can be framed as a RL problem. A RL problem illustrates the challenge of an agent attempting to learn the best possible behavior or policy within a given environment. The agent is given very little information about the environment, its dynamics, and the nature or effects of the actions it can perform; instead, the agent is expected to learn a reasonable behavior by interacting with the environments, discovering which actions in which states are more profitable and finally defining a policy that allows it to achieve its goals in the optimal way.

MDP

Markov Decision Processes (MDPs) are mathematical frameworks for modeling an environment in RL, and MDPs may be used to face nearly all RL problems. An MDP is made up of a finite collection of states S, a set of potential actions $A(s)$ for each state, an intrinsic reward function $R(s)$ and a transition model $P(s', s|a)$. Reinforcement learning is the right approach for determining the best policy for an MDP. RL is a term that refers to the process of learning; in RL, samples created by interaction with the environment are used to improve performances. Real-world settings, on the other hand, are more likely to be lacking of any prior knowledge of environmental dynamics. In such circumstances, model-free RL techniques are in useful.

Definition of a RL problem

A RL problem [24] can be formalized through the following tuple:

(S, A, T, R)

where:

• S is the state space, that consists of all the states of the given environment;

• A is the action space, that includes all the actions available to the agent;

• $T : P(s_{t+1}|s_t, a_t)$ is the environment's transition function, which represents the likelihood of the environment changing from state s_t to state s_{t+1} if the agent takes action a_t;

• $R : P(r_t|s_t, a_t)$, i.e. the chance of the agent to obtain a reward r_t if the agent perform an action a_t in state s_t.

The agent's behaviour is stored in a policy $\pi(a_t|a_t) = P(a_t|s_t)$, which is a probability distribution of the potential actions a_t given the current state of the environment s_t. The quality of a policy is evaluated by its return, which is the total of its predicted rewards over a time interval.

Considering this conceptualization of reward, the agent's goal is to learn the optimum policy π^* that maximizes the return, that is the policy that, while not necessarily unique, provides a higher return than any other policy π. To learn an optimum policy, the agent must find the right balance between the need for exploration (identifying previously undiscovered states and actions that offer a high reward) and the desire for exploitation (greedily selecting the states and acts that are now thought to deliver the highest rewards).

Through steps and episodes, the agent interacts with the environment (and, hence, it learns). A step is a unique interaction between the agent and the environment, consisting

of the agent performing a single action in accordance with the policy π, receiving the reward r_t and watching the environment change from state s_t to state s_{t+1}. An episode is a series of steps that go from one state to the final one. It's worth noting that the setup may vary between episodes, even if the RL problem's pattern (S, A, T, R) remains the same. An RL agent is trained to solve a vast collection of problems with similar structure, rather than simply one single instance of a problem. This variation across episodes is crucial since it helps an RL agent to generalize.

Epsilon-Greedy Action Selection

In order to construct an optimal policy, the agent must simultaneously explore new states while maximizing its total reward.In order to build an optimal policy, the agent faces the dilemma of exploring new states while maximizing its overall reward at the same time. This is called Exploration vs Exploitation trade-off. When an agent explores, it gets more accurate estimates of action-values and when it exploits (that means, relying more on past experience, so, already seen states), it might get more immediate reward. To strike a balance between the two, the ideal overall strategy may include short-term compromises. As a result, the agent should gather sufficient data to make the best overall decision in the future. Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing randomly. The epsilon-greedy exploits most of the time with a little possibility to explore, where epsilon refers to the likelihood of choosing the exploration. You may wonder why we are picking a random action based on the probability given by the neural network, rather than just picking the action with the highest score. This method allows the agent to maintain the correct balance between trying something new and relying on tried-and-true actions.

In the next section, we cover the main principles of a open-source network attack simulator that can be used for training RL agents.

3. IMPLEMENTATION

Network Attack Simulator

Agents interact with what is called an environment. The environment is the outside world it observes. The choice of the environment itself constituted the crucial issue in trying to integrate AI in the PenBox, because the available environments were mainly developed for Control theory problems, minimalistic 3D interior environment simulator for robotics research, autonomous driving, maze, chess and other board games [25]. So the choice fell on the previously developed Network Attack Simulator (2019) by J. Schwarz [28], rather than using a network of virtual machines (VM). This has the advantage of having a reasonable fidelity to the real world while also being flexible. The main disadvantage of using VMs is the relatively high computational cost of running each VM, which can slow down the processing of certain types of AI algorithms, and also the considerable computational assets required to simulate wide networks.
Other network simulators are NS3 [26] and mininet [27], but they are not predisposed for implementing penetration testing.

The NAS environment, based on OpenAI-gym guidelines [29], is outlined to be simple to introduce with negligible dependencies and able to run quick on a single individual computer. It is additionally planned to model network pen-testing at a better level of abstraction in order to permit fast prototyping and testing of algorithms.

The network model defines the organization, connection and configuration of machines on the network and is defined by the tuple subnetworks, topology, hosts, services, firewalls. This model seeks to abstract away those aspects of a real-world network that aren't needed when creating autonomous agents, such as specific types of machine connections and the location of switches and routers. The objective for this abstraction is to keep the simulator as simple as possible while still working at the level that the agent is expected to work at, allowing it to decide which scans or exploits to use against a certain system and in what order. This basic network model is also employed to keep it as generic and scalable as possible.

NAS: Components

Hosts— The machine is the most basic component in the network model. Any device that may be linked to the network and hence linked with and exploited, is referred to as a host in the NAS. Each machine is identified by its address, which is represented as a $(subnetID, machineID)$ tuple, as well as its value and configuration. This is a simplification of IP addresses, which describe the network, subnet and machine address with a 32-bit string and a separate 32-bit subnet mask. Although all machines inside a subnet can interact fully, communication between subnets could be limited by firewall policies. The network topology and firewall settings govern inter-subnet communication.

Each machine's configuration is determined by the services available on it, and each host will not necessarily have the same configuration. This is planned since not every node will be the same and some may be utilized for different purposes (Web servers, file storage and user machines are just a few examples). The services are what the attacker is trying to exploit, so the services present on a machine also define its pivot point, the weakness or exposure. Services are used to describe any software that runs on a system and communicates with the network, for example software listening to an open port on a computer or linked device (see Figure 4).

```
sensitive_hosts:
  (1, 0): 1000
host_configurations:
  (1, 0):
    os: SUSE
    services: [ssh]
    processes: [mysql]
  (1, 1):
    os: SUSE
    services: [ssh]
    processes: [mysql]
  (1, 2):
    os: SUSE
    services: [ssh]
    processes: []
  (1, 3):
    os: SUSE
    services: [ssh]
    processes: []
```

Figure 4. Snapshot of the Python configuration file that describes the features of the hosts in the network

Within the NAS, services are supposed to be the weak points on any specific system and can be thought of as services with a known exploit that the attacker is informed about. Based on this logic within the NAS, each service can be hacked by

a single action, hence the agent's task is to determine which service is running on a machine and select the appropriate exploit action against it.

Topology—The network topology determines how subnets are connected and which subnets are allowed to communicate directly with each other and with the outer world. It can be represented as an undirected graph, with hosts serving as vertices and connections serving as edges. As a result, an adjacency matrix (see Figure 5) can be used to represent it, with rows and columns reflecting the various components.

```
topology: [[ 1, 1, 0, 0, 0, 0, 0, 0, 0],
           [ 1, 1, 1, 1, 1, 1, 1, 1, 1],
           [ 0, 1, 1, 1, 1, 1, 1, 1, 1],
           [ 0, 1, 1, 1, 1, 1, 1, 1, 1],
           [ 0, 1, 1, 1, 1, 1, 1, 1, 1],
           [ 0, 1, 1, 1, 1, 1, 1, 1, 1],
           [ 0, 1, 1, 1, 1, 1, 1, 1, 1],
           [ 0, 1, 1, 1, 1, 1, 1, 1, 1],
           [ 0, 1, 1, 1, 1, 1, 1, 1, 1]]
```

Figure 5. Snapshot of the Python configuration file that represents the network topology

Going back to the Markov Decision Process [30] and so the way the model is logically designed, now we will analyse how it has been possible to make the NAS suitable for the PenBox's complex architecture.

NAS adaptation to PenBox

Action—The action space A, is defined as the set of available actions that can be selected by the agent. What has been done, was a deep research through the PenBox's tables of the database, from which only the most frequently used actions from an existing simple test environment in which PenBox was validated have been chosen to be implemented in the simulator. In fact, to render the model in a more precise way, many tools, which yield redundant outcomes, have been ignored; hence, starting from the 52 penetration testing tools and the related 363 tool_modes, that include every single variation for each tool, only the following actions have been implemented in this work to demonstrate the concept:

• *nmap_host_discovery* [31] : this mode deals with the network sweeping phase through which it is possible to identify the active hosts given a certain IP range;

• *nmap_lightscan_ip* [32]: this mode is responsible for the enumeration and OS fingerprinting phase, hence the listing of all the operating systems installed on each host of the network;

• *openvas_full_and_fast* [33]: this mode detects the majority of the vulnerabilities among the known vulnerabilities and exposures database; it is optimized through the use of information previously collected.

• *openvas_full_and_very_deep_ultimate* [34]: this variation detects processes-related vulnerabilities that can crash services and hosts;

• *hydra_ssh_try_all* [35]: a tool for testing the strength of SSH security, that performs a kind of dictionary attack thanks to a vast wordlist;

• *metasploit*: this tool is capable to put in place a backdoor and consequently opens a Meterpreter shell on the target host, once a vulnerability has been exploited.

Scan actions are considered to be deterministic, always returning informations about services, OSs or processes. There is an exploit action for every potential service on the network and depending on the configuration of the environment, each exploit activity can be deterministic or non-deterministic.

As previously stated, each action, in our case, each tool_mode has a linked $cost(a)$ that can be specified in the Python configuration files.The values in the following table are the results of a deep analysis of the Penbox database: every score is the result of the product between the mean time that the single tool takes to be processed in the attack node and the "stealthiness", that reflects the tendency of the tool to be intrusive and so detectable, or not (this value is retrieved from the PenBox DB, mainly referring to the *execution_level* of each tool).

Table 1. Cost for each action

Action	Action_cost
nmap_host_discovery	1 sec * 3 = 33 ∼ 30
nmap_scan_ip	10+1 sec * 4 = 44 ∼40
openvas_full_and_fast	4 sec * 5 = 20 ∼20
openvas_full_and_deep	4+4 sec * 5 = 40∼40
hydra_ssh_try_all	60/90 sec * 5 = 375 ∼ 350
metasploit	80/95 sec * 3 = 88 ∼90

In order to prevent any fluctuation of the reward-cost model and in order to avoid to assign a negative reward to scan actions (Nmap and OpenVAS), so the ones that don't compromise any target host and, thus, don't earn any prize (the previously mentioned *host_value*), which would result in a sort of "starvation" for this type of actions, a prize of 100 points, named "Discovery value", has been introduced as a workaround to counterbalance the real cost of the action with the indispensable information needed to eventually exploit the potential target machine (address, reachability, active services, etc.).

State—A state, $s \in S$, is defined as the collection of all known information for each machine on the network; it is a sort of "snapshot" of the PenBox's result tree (result_data) at a certain instant of time (see Figure 6).

$$State(subnet\_id,\ host\_id)\begin{cases} Reachable: BOOLEAN \\ OS \\ ftp: BOOLEAN \\ ssh: BOOLEAN \\ mysql: BOOLEAN \\ VULNERABILITY: BOOLEAN \\ BACKDOOR\ SESSION: BOOLEAN \end{cases}$$

Figure 6. State components

The state specifies for each machine:

• if a host is reachable, so detectable by the network scan action, or in the same subnet of a pivot machine that has been target by an exploit action;

• for each service, whether the service is present, absent or its existence is unknown;

• if the host has a vulnerability;

• if a Backdoor has been put in place;

As a result, the state space expands exponentially in proportion to the number of devices and services on the network. For this simulator we assume it is possible for the attacker to have information about the network topology and the services present, as prior knowledge and it must instead learn the other details indirectly through the success and failure of the actions.

Reward: Domain Specific Modeling—The agent or learner is not told what to do apriori, but takes action guided by numerical reward. The reward R for an action of an agent will be composed of two parts R_{gain} and R_{cost}, satisfying $R = R_{gain} + R_{cost}$. There are two reasons for setting this term: one is to limit the number of actions to avoid an endless loop, and the other is to guide the agent to find the attack paths that are as good as possible.
$R(s', a, s) = value(s', s) - cost(a)$
where
$Value(s', s)$ corresponds to:
• the value of any newly discovered machine in s' from s;
• the value for any detected vulnerability (that, according to the table below, corresponds to a normalized value of the CVSS [37]);
• 0 if no new machines were discovered or disrupted and $cost(a)$ returns the cost of action a.

The Table 2 has been built after extracting the most common and valuable vulnerabilities from the PenBox's scenarios executions. Each vulnerability has an associated value [36] that is calculated considering the product between the CVSS rate (according to the Common Vulnerabilities Scoring System, which indicates the grade of severity for the single exposure and may help organizations in prioritizing the mitigation actions). Relying on the CVSS is one of the options to categorize those vulnerabilities, which contribute to achieve the scenario goal or open up state spaces which were previously unknown (i.e. facilitate network/system propagation).

Table 2. **Reward for each discovery**

Result_Type	Reward
IP (nmap_host_discovery)	100
OS, VERSION(nmap_scan_ip)	100
TCP timestamps	2,3 CVSS* 100
SSH Weak MAC Algorithms	2,6 CVSS* 100
SSH Weak Encryption Algorithms	4,3 CVSS* 100
FTP Unencrypted Cleartext Login	4,8 CVSS* 100
DB weak password	9,0 CVSS* 100
Blackstratus LOGStorm	9,0 CVSS* 100
BACKDOOR_SESSION	1000

Network scenario implementation

PenBox is deployed in a test environment together with a representative ground segment system chain. It is also possible to whitelist IPs to limit the test execution, so deployment in a segregated network is just a precaution.

The PenBox will run penetration-testing tools in relation to a scenario involving a specific target. For this to be feasible under the right conditions, it needs to run in a highly specialized environment for two reasons:

• It needs to be able to target the given target (i.e., a system in the ground segment chain) with its penetration- testing tool ;

• It needs to be properly isolated from the outer world and even test environments to make sure it does not disrupt other unrelated activities in its environment;

For these reasons, the PenBox needs its own instance of the ground segment and this instance shall be in a separate and isolated network. This project uses a reference ground segment engineering development environment, which provides an isolated instance of a fully representative mission operations centre operational command and control chains managed by ESOC.

The ESOC test environment (see Figure 7) currently consists of prime and backup MCS servers, two MCS clients and a mission simulator. The deployed configuration is the Gaia mission (SLES 12 baseline). TM/TC (Telemetry and Telecommand) flow is demonstrable over the private VLAN between the MCS Server and simulator. This setup is executed whilst also running the PenBox, to simulate an attacker having access to a real operational environment.

Currently PenBox is fit for being executed in a segregated test environment, for this reason it was necessary to introduce a custom network topology which would reproduce the ESOC test conditions; The previously mentioned configuration files are written using the YAML.

The PB1 in Figure 7 is meant to be the host on which the PenBox is installed and so it is framed in the wider context of an "Insider Attack", while the MCS server is the exploitable machine. According to the current settings of the benchmark environment, the gateway GW1 and the firewalls OFW1 and VFW1 haven't been included in the topology.

The render method written in Python (see Appendix A) allows to view the environment and network architecture, as well as an attack episode on the network. The first choice is used to simply represent the network as a graph, with each vertex representing a host in the network and machines on the same subnet clustered together, while edges representing connections between each machine inside a subnet and between subnets (see Figure 8). This option allows the user to view the network's topology. The second option allows the user to see how the network's state changes during an attack episode. Figure 9 illustrates the main elements characterizing the desired scenario, in our case the "esa" network (e.g. the number of hosts, vulnerabilities, services etc.)

PenBox Attack Simulator

In this section we will start to talk about the PenBox Attack Simulator, that is the result of the first attempt to integrate the PenBox's main features seen before in the network simulator (even though we are referring to an high level of abstraction compared to the complexity of the ESA software). In the previous section we introduced the main notions about ML/AI; now we can define the RL model chosen in this case study.

In RL, when a model is present it is referred to as model-based RL, while if no model is foreseen then it is model-free RL. The model of an environment is something that tells the agent something about how the environment will behave and allows the agent to make inferences [38]. Model-

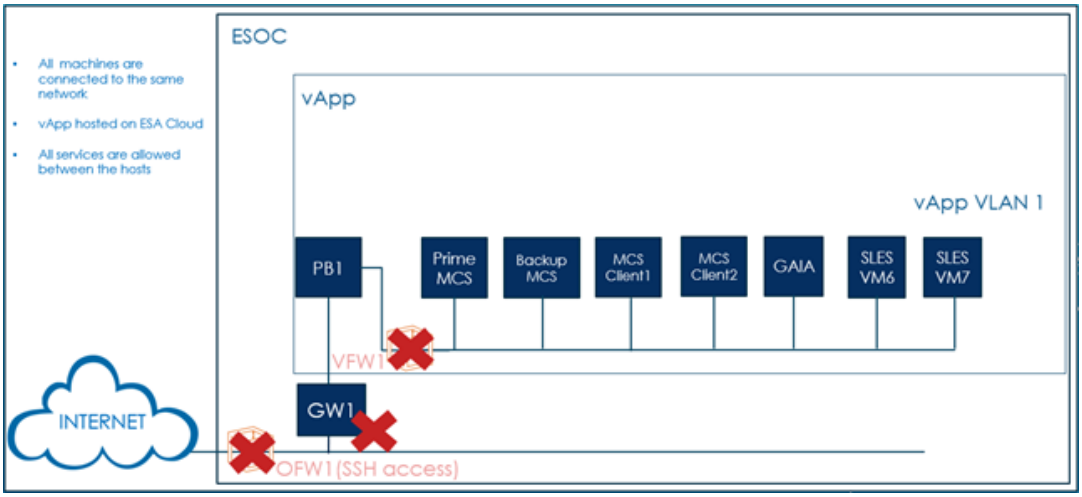


Figure 7. Representation of the ESA test environment on which PenBox machine acts. The target environment in scope has sensitive data stored in Prime Mission Control Server. In this case study, the goal of pentest is to access this VM.

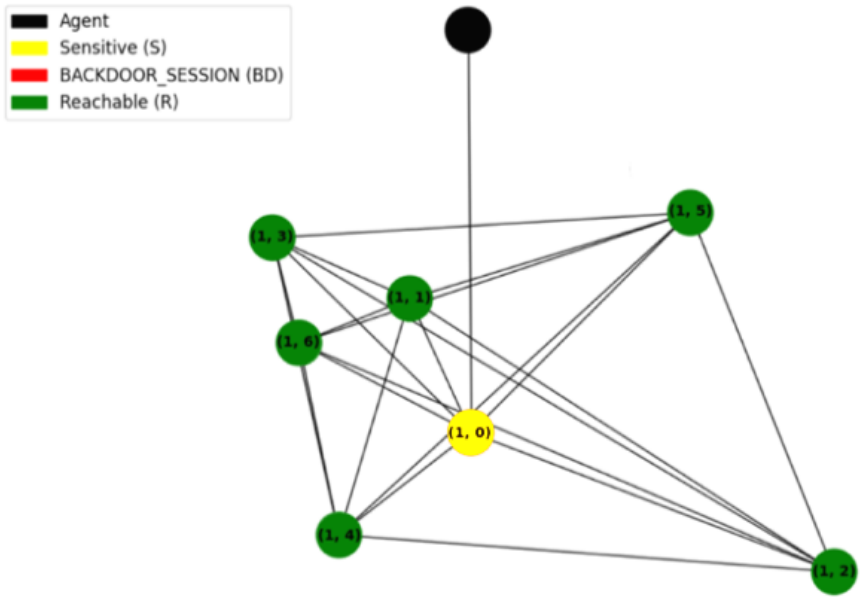


Figure 8. Representation of the "esa" network topology obtained running the visualize graph.py script

Name	Type	Hosts	OS	Services	Processes	Vulnerabilities	BACKDOOR_SESSION	Actions	Observation Dims	States	Step Limit
esa	static	8	1	4	1	1	1	42	(8, 21)	10752	2000

Figure 9. Scenario description obtained with the execution of the describe_scenario.py script

based problems are usually solved by planning while model-free problems need to rely on trial and error to identify the ideal environment policy.Thus, we are actually conceiving penetration testing as an MDP where the transition function T is unknown and then using RL in a simulated environment to achieve attack policies. One of the major benefits of an RL strategy is that it permits to face this challenge without any previous model about the probability of action results for any state and enables the agent to learn autonomously.

Then the agent learns expressly the model, interplaying with the environment. As the information security area develops, it is just required to update the tasks an agent may do and leave the agent the possibility to adapt to these new conditions.

The learning process takes place exclusively via deduction: the agent derives the worth of its actions at different stages looking at the rewards/penalties it gets through its interaction with the environment. Standard RL presupposes a small amount of prior knowledge, therefore it defines a considerably more challenging task than the one faced by a real-world attacker. Indeed, a human may lean on several additional informations [39], reasoning approaches and procedures to constrain and steer his range of choices (e.g., he can gather information on the target system on the internet from other sources, rely on OSINT, infer from the target systems or

formulate hypotheses).

Such options are generally excluded from any regular RL agent which can only count on its attitude to rapidly and efficiently collect experience and infer from it.

Action-Value algorithms for RL— Several algorithms have been proposed to solve the RL problem. One of the simplest, yet well-performing, family of RL algorithms is the family of action-value methods.

For this particular investigation, Q-learning is used to develop the best policy for model-free scenarios using an RL algorithm. It is based on experience to learn the Q value, $Q(s, a)$ in domain-specific terms; such function gives the agent the estimated value for doing a certain action that will lead to the next state. When such Q-value function converges, we may use it to identify the best action for a given state by adopting the action with the greatest Q-value, and therefore use it to determine the best policy for a specific scenario.

There are two main ways of representing the action-value function:

• *Tabular representation*: this approach is based on a matrix or a tensor Q to exactly encode each pair of (state, action) and estimate its value; tabular representations are simple, easy to examine, and statistically sound; tabular methods refer to problems in which the state and actions spaces are small enough for approximate value functions to be represented as arrays and tables. However, they have limited generalization ability and they do not scale well with the dimension of the state space S and the action space A.

• *Approximate representation*: this representation relies on fitting an approximate function $\hat{q}$; usual choices for $\hat{q}$ are parametric functions ranging from simple linear regression to complex deep neural networks; approximate functions solve the problem of dealing with large state space S and the action space A, and provide generalization capabilities; however, they are harder to interpret and they often lack statistical guarantees of convergence.

Q-Learning—Reinforcement Learning problems with a discrete actions space are frequently described as Markov Decision Processes, but the agent has no notion what the transition probabilities are and it also has no idea what the rewards will be. It must experience each state and transition at least once in order to know the rewards, and it must experience them several times in order to have a reliable approximation of the transition probabilities. Generally, we suppose that the agent initially knows nothing more than the available states and actions. The agent explores the MDP using an exploration strategy, such as a completely random policy (off-policy concept, i.e. the policy being trained is not the one being executed), and as it goes, the algorithm updates the estimates of the state values based on the transitions and rewards that are actually observable, plus the rewards it expects to get later (assuming it acts optimally); This kind of RL approach can be formalized with the following formula:

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\substack{old \\ value}} + \underbrace{\alpha}_{\substack{learning \\ rate}} \cdot \overbrace{(r_t + \gamma + \underbrace{\max_{\alpha} Q(s_{t+1}, a_t))}_{\substack{estimate\,of\,optimal \\ future\,reward}}}^{learned\,value} \quad (1)$$

where α is the learning rate and γ is the discount factor. Intuitively, at every step the estimation of $Q(s_t, a_t)$ moves towards the true action-value function $q(s_t, a_t)$ by a step α like the SGD (Stochastic Gradient Descent). This method keeps track of a cumulative average of the immediate rewards the agent receives upon leaving that state, as well as the future rewards it expects to get late.It can only genuinely converge if the learning rate is progressively reduced (otherwise it will keep bouncing around the optimum).

DQN—A DNN used to estimate Q-Values is called a Deep Q-network (DQN) and using a DQN for Approximate Q-Learning is called Deep Q-Learning. This technique helps in coping with large state spaces, thus overcoming the scaling problem of the tabular Q-Learning. Instead of taking a "perfect" value from the Q-table that match the couple $(State, Action)$ with a unique vale, we train a Neural Network to estimate the values (see Figure 10).

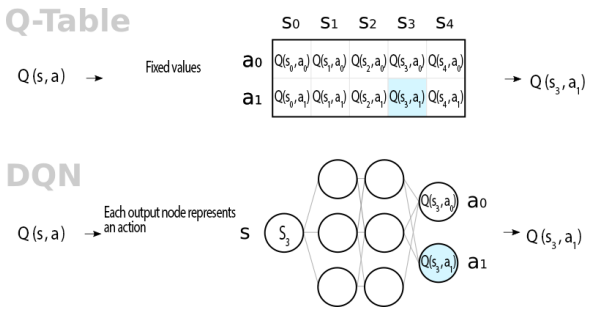


Figure 10. Visual comparison between Tabular Q-learning and DQN

When we explained Q-learning earlier, we used the Equation (1). With the neural network taking the place of the Q-table, we can simplify it. The learning rate is no longer needed, as our back-propagating optimizer will already have that.

Considering Equation (1) Learning rate is simply a global wrapper that reflects the attitude of the model to adapt to the already achieved informations, and one suffices. Once the learning rate is removed, you realize that you can also remove the two Q(s, a) terms, that represented the Q-table values to progressively update, as they cancel each other out after discarding the learning rate.

$$Q(s_t, a_t) \leftarrow r_t + \gamma max Q(s_{t+1}, a) \quad (2)$$

Although reinforcement learning is sometimes defined as a distinct category from both supervised and unsupervised learning, we shall borrow from the supervised approaches here. Although it is commonly claimed that reinforcement learning requires no training data, this is only partially true. Although no prior training data is required, it is gathered and used in a similar manner when exploring the simulation. The agent will keep track of experiences as it explores the simulation.

$$Single\,experience = (oldstate, action, reward, newstate)$$

Training our model with a single experience let the model estimate Q values of the old state; let the model estimate Q values of the new state; calculate the new target Q value for the action, using the known reward; train the model with

$$input = (oldstate), output = (targetQvalues);$$

Unlike the Q-learning function $Q(s, a)$, our network does not receive $(state, action)$ as input. This is because we are only copying the Q-table, not Q-learning as a whole. The input is just a tensor of states, or the observations, and the number of output neurons would be the number of the actions that an agent can take, while the results are the Q-values for all possible actions (forward, backward) for that particular state (see Figure 11).

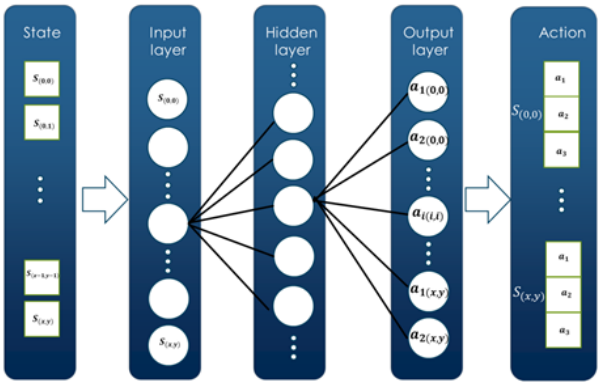


Figure 11. Outlook of the NN that combines each input state with every possible action in DQN

Batch—The simplest approach is often called online training where the system is progressively trained by continually feeding it with data, either separately or in tiny groups known as mini-batches, that can be interpreted as records of experiences. The model is fed with multiple random instances fetched from a kind of memory action register; each learning step will be quick and cheap, allowing the system to learn about new data as it occurs, generating more reliable outcomes.

Hyperparameters—One of the most difficult and time consuming parts of deep reinforcement learning is the optimization of hyperparameters. These values — such as the discount factor γ, or the learning rate — can make all the difference in the performance of your agent. There's no way to determine if a higher or lower value for a given parameter would boost overall rewards unless agents are trained to see how the hyperparameters impact performance. This means several, expensive training runs to find an efficient agent, as well as keeping track of the trials, data, and everything else related to training the models. The DQN agent is given the following hyperparameters:

The following hyperparameters are passed to the DQN agent:

•$training\_steps$ - The number of rounds the agent will play during the learning process; Once you have run a 10,000 training steps, you have essentially trained 10,000 different neural networks (each with just one training instance). These neural networks are certainly not isolated, as many of their weights are shared, yet they are all unique. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks; a unique neural network is generated at each training step.

•$gamma$ - Decay or discount rate, to consider that an action has more influence on the near future than on the distant future; for example, with a discount rate of 0.9, a reward of 100 that is received two time steps later is counted as only $0.9^2100 = 81$ when you are estimating the value of the action. The discount rate may be thought of as a measure of how much the future is valued in comparison to the present: if it is extremely near to 1, the future is almost as valuable as the present. If it's close to 0, only immediate benefits are important. If it is close to 0, then only immediate rewards matter. Of course, this has a significant influence on the best policy: if you value the future, you could be ready to put up with a lot of present penalty in exchange for the possibility of future gains, but if you don't value the future, you'll take whatever instant profit you can get and never invest in the future.

•$epsilon$ - We will use the ϵ-greedy policy so ϵ is the Exploration rate. This is the rate at which an agent randomly decides its action rather than prediction; ϵ gradually decrease from $init\_epsilon$ (1.0) to final_epsilon (e.g., 0.05) every $exploration\_steps$; so ϵ represents the decrease in the number of explorations as the model becomes better at guessing.

•$learning\_rate$ - Determines how much the neural network learns in each iteration, how fast it should adapt to changing states; If you choose a high learning rate, the model will adapt quickly to incoming data, but it will also forget previous informations faster. If you select a low learning rate, the networkmodel will have greater inertia, which means it will learn more slowly, but it will also be less sensitive to outliers in new states or non-representative data sequences.

•$replay\_size$ - Maximum number of experiences stored in replay memory. It's a collection of tuples $(S, A, R, S^{'})$ that prevents weights oscillation and harmful correlations: as the agent interacts with the environment, the tuples are gradually added to the buffer. The most basic solution is a fixed-size buffer with latest data appended to the end of the buffer, pushing the oldest experience out. Using a random selection of previous experiences to update the Q-network, rather than simply the single most recent occurrence, experience replay allows us to learn more from individual tuples several times, recall infrequent occurrences, and make better use of our experience in general.

$target\_update\_freq$ - The updates from the target network should be less frequent than training steps, because it is meant to balance the outputs, so that over or under estimations of Q-value functions do not cause inconsistent feedbacks.

The replay memory is optional, but highly recommended. Without it, you would train the DQN using consecutive experiences that may be very correlated. This would introduce a lot of bias and slow down the training algorithm's convergence [40]. We ensure that the memories provided to the training algorithm are somewhat uncorrelated by employing a replay memory.

The Adam [41] optimization algorithm is adopted to train a deep neural network, instead of the more simple SGD (Stochastic Gradient Descent).

$$\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3}$$

where

•m_t is a bias-corrected first moment estimate,

•v_t is a bias-corrected second raw moment estimate,

•α is the step size, ϵ is the correction parameter, and

•θ_t represents the weight of the connection between neurons of the deep neural network.

The original $Q(s, a)$ table will be replaced with a neural network fitting function whose update formula is much more efficient.

One essential assumption is that the agent has full network topology information: this implies that the agent is fully aware about the IP addresses and connections of each computer on the network. Most attack planning techniques use this premise, which is related to the availability of network structure informations from a customer requesting a security assessment; this will reconduct to a stationary state representations, which will provide a more stable set of results in comparison to applying RL to dynamic states problems where the state, so the number of hosts changes progressively.

4. TESTS

Hardware/Software setup

All the tests have been conducted on Anaconda (prompt), a distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment. We used a laptop equipped with an Intel Core i7-8550U CPU at 1.8 GHz, 4-core processors, running two threads per core, thus 8 logical processors, 8GB of RAM and NVIDIA GeForce MX130 Graphics Card. The GPU was used to speed up the performances of the ML model thanks to to the CUDA library [42]; Nvidia's Compute Unified Device Architecture library (CUDA) enables developers to leverage CUDA-enabled GPUs for a wide range of tasks (not just graphics acceleration). Nvidia's CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for DNNs. It provides optimized implementations of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling. It is part of Nvidia's Deep Learning SDK (note that it requires creating an Nvidia developer account in order to download it). TensorFlow controls GPUs and boosts calculations with CUDA and cuDNN.

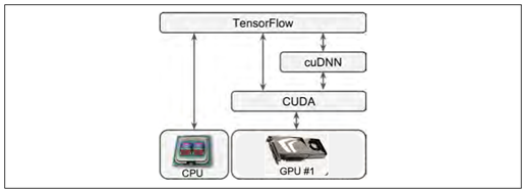


Figure 12. CUDA logic

Tests and parameters tuning

In this section we provide concrete instances of simple decision-making challenges, we model them in the form of RL problems using the formalism discussed in Section 4, and we solve them using Q-learning. The experiments were meant to meet a results-oriented approach; rather than comparing the benchmark ML algorithms, the tests are focused on the DQN technique, at first, manually tweaking the hyperparameters of the Neural Network, and then, taking advantage of tools and libraries for parameters automatic optimization. The typical experimental approach involved selecting the scenario which reproduce the ESA test environment (i.e., network size, topology, number of services, etc.) training the agent for a certain time period and then rating the final policy. Each of these steps is discussed in the subsequent sections. A simple test environment is shown in Figure 7 and described in the previous section. To better understand the influence of the variable of interest, we sought to maintain as many parameters consistent as possible for each case. Hyperparameters selection was done using a mixture of informal search on different size standard generated networks, selecting default choices from the machine learning library used and values seen in literature for DQN.

Case Study: ESA test environment

We simulated the ESA test network on which the PenBox machine actually carries out its tasks. The network comprised of 8 hosts, one networks. One SLES/SUSE [43] operating system was utilized for experimental simulation. We implemented four key services (SSH, FTP, HTTP, and TCP) that fit well with the kind of vulnerabilities detected by PenBox, and one vulnerable process with an associated exploit possible with Metasploit, which is the pivot process the will allow to put in a place the BACKDOOR SESSION on the MCS server (Host (1, 0)).

The keyboard_agent.py script mimics the action of a real-world pentester/attacker that executes the action step-by-step and choose manually what will be the next action. Unfortunately all these details (in this overview we can immediately spot that the sensitive machine is the Host(1, 0)) are not available in a realistic penetration testing; In fact, in this situation, the actor can easily take advantage of the vulnerable service performing bruteforcing action with Hydra against the sensitive machine, which costs 350, according to the cost Table 1 , until it will be successful, thus exploiting the SSH service and achieving USER permission. Once exploited the vulnerability, the actor can obtain ROOT access using Metasploit (action cost: 90). In the best case, we have reached success executing only two tool modes, gaining the highest reward possible(560, that is the result of the compromised host value minus the cumulative actions' cost). Now, we consider our DQN algorithm in order to see how an intelligent agent would behave when facing this kind of challenge, with the final aim of finding the optimal attack path with the minimum number of steps, assuming to only have information about the network topology and the number of hosts as prior-knowledge.

Test: DQN

Training— For each set of hyperparameters we trained the DQN until it generates the related policy file (a PyTorch [44] file that contains the outputs of the Neural Network, so the action value for each action for all the states, that could act as guideline for similar scenarios in the actions-selection task).

The test environment is shown in Figure 7 and described in detail in the previous section. This architecture has been used in previous studies, testing automated attack trees in virtualized environments, but it is still a very simple context, not representative of real world.

Policy Generation and execution— Here we focus on the execution of the DQN algorithm that will return the policy for that particular run; Then we can see the outcomes of the training launching the script run_dqn_policy.py; in this case, the system has reached the goal (compromising the sensitive machine) after 703 steps after a 35 seconds training.

Results

In the following Table 3 the 20 most valuable results are shown in order to try to understand how the outcomes, the training time and the number of actions change after tweaking the parameters in DQN. 150 test have been carried out. We recorded the proportion of the evaluation runs which were solved by the agent, where a run was considered solved when the agent successfully exploited the sensitive host on the network within the step limit (2000 steps). In the following table, the most important entries are listed, ignoring redundant and inconsistent values.

Discussion

The RL agent might be able to solve a difficult problem with a delicate and sharp structure, such as the one provided, but at a significant computational cost. The best results in terms of number of actions, is the one that shows how it can be possible to exploit the target machine with a sequence of 33 actions. It is not the ideal result, because the manual execution of the available tool_modes, has proven that the optimal attack sequence consists of 4 actions, but we are confident that the model can be still improved. Unexpectedly, we have deduced that with a low value of γ, the discount factor, thus making the agent "myopic" (or short-sighted) and only considering current rewards, we improve the agent's performance of relying on already performed action. The best results are achieved with an extremely large number of training steps and drastically reducing the exploration steps. This represents the exploration vs exploitation trade-off itself. This strategy enables the agent to achieve the correct balance between exploring new actions and focusing on those that have previously shown to be profitable. This reinforces the argument of trusting on past experience, but with attention to what is unknown, not yet explored.

5. CONCLUSION

We have investigated the application of AI/ML to automated penetration testing that helps reduce human interaction in penetration testing, utilizing domain-specific reward to represent intrinsic real-world settings of a pentesting exercise. The framework utilizes network configuration and vulnerability information to produce an estimate attack path with the aim of minimizing the steps and so the performed actions, completely ignoring the possibility to optimize the training time; nevertheless, in the future, we aim to investigate these possibilities. We remarked the RL's capacity to solve model-free problems with prior knowledge and deemed that a certain type of side-information (i.e. services present, vulnerabilities present, OSs) can be particularly effective to overcome a decision-making problem.

We think that RL could be combined with traditional artificial intelligence approaches and balance the model-free inference of RL with model-based and empirical biases. The results prove that the challenges we identified, as the identification of optimal attack paths with the minimum number of performed actions and the subsequent cost-saving in terms of time and assets, are significant and have demonstrated how RL techniques may be employed to meet these goals and obtain valuable outcomes that, if properly enriched with more complex conjectures(e.g.considering all the 52 penetration testing tools and the related 363 tool_modes, that include every single variation for each tool instead of the limited number we analysed in this work, can provide significant benefits in terms of performance for the penetration testing domain.

Table 3. Results overview

Test	lr	training_steps	batch_size	replay_size	exploration_steps	gamma	target_update_freq	Actions	Time	Success
1	0.01	20000	32	100000	10000	0.05	1000	703	35 s	Y
2	0.01	10000	32	100000	2000	0.05	1000	1868	1 m 33 s	Y
3	0.01	1000	32	100000	500	0.05	1000	2000	21 s	N
4	0.01	100000	32	100000	500	0.05	1000	601	8 m s	Y
5	0.01	100000	128	10000	500	0.05	1000	1410	15 m 18 s	Y
6	0.00001	100000	8	100000	500	0.05	1000	1247	31 s	Y
7	0.0001	100000	8	100000	500	0.05	1000	601	8 m s	Y
8	0.001	10000	8	100000	500	0.05	1000	1867	1 m 3 s	Y
9	0.01	100000	8	1000	500	0.05	1000	1936	7 m	Y
10	0.001	100000	32	1000	500	0.05	500	2000	9 m 10 s	N
11	0.001	100000	32	100000	500	0.05	500	2000	16 m 11 s	N
12	0.01	10000	32	10000	20000	0.6	400	2000	33 m 12 s	N
13	0.1	10000	32	10000	20000	0.7	300	2000	12 m 11 s	N
14	0.01	100000	32	10000	10000	0.8	300	2000	19 m 32 s	N
15	0.1	100000	32	1000	5000	0.3	200	1342	16 m 4 s	Y
16	0.01	100000	32	1000	5000	0.25	200	1549	19 m 5 s	Y
17	0.01	200000	16	500	1000	0.2	400	876	12 m 50 s	Y
18	0.01	200000	16	400	700	0.1	500	190	13 m	Y
19	0.01	200000	16	400	500	0.09	600	84	22 34 s m	Y
20	0.01	200000	16	400	500	0.08	700	33	15 m 12s	Y

Future work

The scenario addressed in this study was a highly simplified version of penetration testing simulations, closer to action-sequencing issues than real-world security assessment. Moving further would include both increasing the complexity of the specific domain and enhancing the way an RL agent handles structure and previous knowledge.

Regarding the scaling properties, a practical approach would aim to enhance the completeness of the problem by increasing the size of states and actions to match the reality. Additional objectives include the use of model-based approach to allow an agent to learn an approximate environment transition function, to enable the possibility to learn off-line, through simulations, how to manage rewards a priori.

One of the main further developments that could be carried in the next steps, should be creating an interface between the simulator and the PenBox, so that it will be possible to submit the best actions' path generated by the RL model; this will lead to a dramatic reduction of the computational effort and a lowering of the processing time needed to accomplish a full penetration test with the 52 tools contained in the framework used by the ESA.

The main limitation, so far is not the intelligent automation nor its effectiveness, but the standard IT oriented nature versus the legacy nature of the space systems under tests. On the one hand, the tools hackers and the security community create and maintain mainly target the most commonly used protocols (e.g., http) and systems (e.g., Microsoft Windows). Space systems on the other hand are different in most of these aspects. Thus, while PenBox is capable of carrying out a lot of attacks, many of the usual tools are quite ineffective on space systems, not necessarily because the latter are actually secure, but simply because the tool does not address many of these system's vulnerability. This indicates that space systems are actually not easy to attack with standard penetration testing tools, which could be perceived positively, however also introduces a risk of complacency, since security by obscurity is a very poor defence.

PenBox is just a prototype and will be evolved, but the proof of work explained in this paper has surely provided a contribution for considering ML/AI driven techniques as part of that evolution. That said, in the future ESA systems will become more and more generic and will be based on standard technologies. This is good for reducing costs but makes security all the more important.

APPENDICES

A. SOURCE CODE

This project's source code may be obtained, after ESA approval, asking to the authors.

The code is written entirely in Python 3 and is divided into several logical components. The Environment, Agents, and Scripts are the three main components.

Environment

The files in the "env" folder is listed in Table 4, along with an explanation of its function in the PenBox Attack Simulator.

Table 4. Main elements of the Environment

File	Features
action.py	Contains the tool_modes which were chosen from PenBox.
environment.py	The PenBox Attack Simulator's primary class. It manages the network attack model and is the primary interface for engaging with the simulator.
render.py	It has features for graphically rendering the environment and episodes.
state.py	This package contains the State class, which specifies a state in the PAS (PenBox Attack Simulator).
network.py	It outlines and manages the network's behaviour and configuration in the PAS.

ACKNOWLEDGMENTS

This work is the result of the co-operation of the Applications and Robotics Data Systems Section (OPS-GDA) of ESA/ESOC, Deutsche Forschungszentrum für Künstliche Intelligenz (DFKI) and the "Mediterranea" University of Reggio di Calabria.

REFERENCES

[1] Penetration testing. Y. Stefinko, A. Piskozub, and R. Banakh - "Manual and automated penetration testing. benefits and drawbacks. modern tendency".

[2] Kali. https://www.kali.org/.

[3] BackBox. https://www.backbox.org/.

[4] Pentest Box. https://pentestbox.org/.

[5] Metasploit Framework. https://www.metasploit.com/.

[6] Burp suite. https://portswigger.net/burp.

[7] PT Guidelines. https://www.vumetric.com/blog/top-penetration-testing-methodologies/.

[8] Attack trees. Schneier B. - Attack trees. Dr Dobb's J.

[9] Personalized Vulnerability analysis. Roberts M. et al. - "Personalized vulnerability analysis through automated planning".

[10] Attack planning. J. L. Obes, C. Sarraute, and G. Richarte - "Attack planning in the real world".

[11] Automated PT. Khan S, Parkinson S - "Towards automated vulnerability assessment. 27th Int Conf on Automated Planning and Scheduling".

[12] Core-Impact. Core Security, 2019. Core Impact Penetration System. https://www.secureauth.com/products/penetration-testing/core-impact .

[13] Nondeterministic scenarios. C. Sarraute, G. Richarte, and J. Lucángeli Obes - "An algorithm to find optimal attack paths in nondeterministic scenarios," in Proceedings of the 4th ACM workshop on Security and artificial intelligence.

[14] POMDP. Kurniawati H, Hsu D, Lee WS, 2008. SARSOP: efficient point-based POMDP planning by approx-

imating optimally reachable belief spaces. In: Brock O, Trinkle J, Ramos F (Eds.) Robotics: Science and Systems IV. MIT Press, Massachusetts, USA, Chapter 10.

[15] Classical planning. Shmaryahu D, Shani G, Hoffmann J, et al., 2017. Partially observable contingent planning for penetration testing. 1st Int Workshop on Artificial Intelligence in Security.".

[16] Uncertainty in POMDP. Sarraute C. Buffet O. Hoffmann J. - "POMDPs make better hackers: accounting for uncertainty in penetration testing".

[17] Protocol oriente analysis. Samant N. 2011 - Automated Penetration Testing. MS Thesis, San Jose State University, California, USA"

[18] Inference engine. Stefinko Y, Piskuzub A, 2017 - "Theory of modern penetration testing expert system. Inform Process Syst."

[19] Mitigation analysis. Steinmetz M, 2016 - "Critical constrained planning and an application to network penetration testing."

[20] Resource limitation. Backes M., Hoffmann J., Künnemann R. - "Simu lated penetration testing and mitigation analysis.".

[21] PenBox.https://gsaw.org/wp-content/uploads/2019/03/2019s04wallum.pdf.

[22] ML/AI manual. A. Géron - "Hands-on Machine Learning. 2nd ed. CA 95472: O'Reilly".

[23] RL. R. S. Sutton and A. G. Barto - "Reinforcement learning: An introduction".

[24] CTF. Zennaro et al. - "Modeling Penetration Testing with Reinforcement Learning Using Capture-the-Flag Challenges and Tabular Q-Learning".

[25] AlphaGo. D. Silver et al., "Mastering the game of go without human knowledge"

[26] NS3.https://www.nsnam.org/.

[27] Mininet.http://mininet.org/.

[28] NAS. J.Schwarz - "Autonomous Penetration Testing using Reinforcement Learning".

[29] OpenAI-gym. G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba - "Openai gym".

[30] ML cyberattack strategies. J. A. Bland, M. D. Petty, T. S. Whitaker, K. P. Maxwell, and W. A. Cantrell, "Machine learning cyberattack and defense strategies,".

[31] Nmap. G. Lyon - "Nmap security scanner".

[32] Nmap_light_scan. https://3os.org/penetration-testing/cheatsheets/nmap-cheatsheet/.

[33] Openvas_full_and_fast. https://hackertarget.com/openvas-scan/.

[34] Openvas_full_and_very_ultimate. https://community.greenbone.net/t/comparing-very-deep-and-very-deep-ultimate-scans/1756.

[35] Hydra_SSH. https://linuxconfig.org/ssh-password-testing-with-hydra-on-kali-linux.

[36] MSN. Chowdhary et al. - "Autonomous Security Analysis and Penetration Testing".

[37] CVSS. K. Scarfone, P. Mell - "An analysis of cvss version 2 vulnerability scoring," in 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE.

[38] Efficient Network. M. C. Ghanem and T. M. Chen, "Reinforcement learning for efficient network penetration testing".

[39] Human level RL. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," Nature.

[40] Bias. M. S. Boddy, J. Gohde, T. Haigh, and S. A. Harp - "Course of action generation for cyber security using classical planning." .

[41] NIG-AP. ZHOU et al.- "NIG-AP: a new method for automated penetration testing".

[42] CUDA. https://developer.nvidia.com/cuda-toolkit.

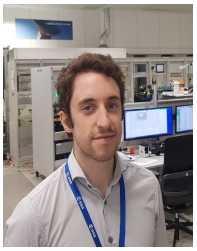
[43] SLES. https://www.suse.com/products/server/.

[44] PyTorch. https://pytorch.org/docs/stable/index.html.

BIOGRAPHY

Alessandro Confido *earned a Master's Degree in Informatics Engineering, at "Mediterranea" University, developing his thesis at the ESA-ESOC in Darmstadt, Germany. During his academic career he learnt the fundamentals of Computer Programming, Cybersecurity and Machine Learning; after the period spent within the edges of the university seeking to enhance the experience in a wider approach to Deep Learning and Data Science, he applied and won a scholarship to take part to the first italian PhD Programme in Artificial Intelligence announced in 2021.*

Evridiki V. Ntagiou *is a Research Fellow at the European Space Operations Centre of ESA, at the Department of Ground Systems Engineering and Innovation.She is an Electrical and Computer Engineer, and received her PhD on Automated Optimal Planning for Constellation missions from ESA and the University of Surrey.Her current research interests are in the area of applied Artificial Intelligence for Mission Operations and Space Safety applications and she is the domain lead for AI in the Mission Operations Data Systems Division.*



Marcus Wallum *is an Operations Data Systems Manager working at the European Space Operations Centre in Darmstadt, Germany. Marcus holds a Masters degree in Astrophysics from the University of St Andrews,professional certification in Information Systems Security from (ISC)2 and in Architecture and Systems Engineering from MIT. Marcus has worked in a variety of engineering and operational roles on ESA Earth Observation and Navigation missions and is now mainly focused on R&D where he leads the domains of Digital Engineering and Cybersecurity within the Mission Operations Data Systems Division at ESOC.*