# DAT620 Project Report

Automated Penetration Testing with Reinforcement Learning

Håvard Moe Jacobsen
University of Stavanger, Norway
ham.jacobsen@stud.uis.no

## ABSTRACT

As hacking tools have become more accessible and automated, the counterforce, *automated penetration testing* is also becoming increasingly important. The different tools and frameworks available were explored in order to create a pipeline with the possibility of automating the process of penetration testing. There is also performed some minor experiments with the Metasploit framework and simple penetration testing techniques on vulnerable web applications to familiarize myself with the subject area.

There is then combined a pipeline consisting of different tools and frameworks used to create a skeleton for the further development of agents based on reinforcement learning. After having stitched together this pipeline, a proposed baseline method using a random agent is introduced.

## KEYWORDS

Penetration Testing, Automated Penetration Testing, Reinforcement Learning, Deep Reinforcement Learning, Q Learning, Deep Q Learning, Metasploit Framework

## 1 INTRODUCTION

In this modern era, securing web servers and networks has become an increasingly important challenge. As more corporate and private data are getting stored in clouds and online databases, it is vital that we are able to secure important data from malignant hackers. Testing and securing online systems from the attacker's tools is often done by performing penetration testing (*pentesting*).

Securing systems was earlier typically solely done by focusing on the defensive aspect of a system's development. Even though this is an important start when it comes to creating a secure system, it is rarely enough as it is easy to miss important details that can have disastrous consequences. Using pentesting as a way to secure systems rose in popularity in the early 20s and quickly became important for creating secure systems. The central idea of penetration testing is to look for potential exploits and vulnerabilities the system developers may have neglected when building the infrastructure around the system.

Traditional pentesting is performed manually with tools such as Nmap, the Metasploit framework, Burpsuite, and more (we will go more into depth in section 2.1). Performing a thoroughly pentest for a large network is a time-consuming task that can prevent pentesters from spending more time on the more critical parts of the network. This is where reinforcement learning comes into play. By the use of reinforcement learning to automate the task of basic pentesting, one can spend more time going into detail on some of the cases acquired from the automated pentest's output or on other edge cases.

For this project, I will put together a pipeline to aid in performing pentesting and defense of standalone web applications and servers by the use of reinforcement learning.

Supervised by Ferhat Özgur Catak.

## 2 BACKGROUND

In this section, we will explain some relevant theory, technologies, and frameworks that would help for getting a better understanding of this project.

### 2.1 Penetration Testing

Penetration testing is the act of testing the security of a computer system. This includes scanning for attack vectors and using those to gain and maintain access to the system.

*2.1.1 Different Phases of in a Pentest.* Penetration testing is often separated into 5 different steps [10]:

(1) **Planning and Reconnaissance**
In this step, one decides which part of the system is to be tested, and how. This step also involves the gathering of information that could be used to better attack the target.
(2) **Scanning**
This step involves using technical tools and methods to further increase the knowledge of the target system. An often-used tool for this part is Nmap [17]. In this step, we usually find some attack vectors which potentially could be exploited. If we are given access to the application code, the scanning part could also include scanning of the code both static, and in running state [10].
(3) **Gaining Access**
In the previous phase, we identified the attack vectors, and potential methods to breach the system. It is time to start trying to breach the system. E.g. if the Scanning phase indicated that a web application may be vulnerable to a SQL injection, we could try performing an attack of this type in order to gain access to the system.
(4) **Maintaining Access**
In this step, we would want to stabilize the connection to the system and find a way to preserve communication system. Attackers could want to maintain the access in order to eavesdrop or continue to steal information in the future.
(5) **Analysis**
Summarizing the results in a report containing information about which vulnerabilities were exploited, the data that was obtained and the duration of which the pen tester was

able to stay in the system undetected, and other relevant information.

There are different ways to perform penetration testing. Some of them include **black**, **gray**, and **white**-box penetration testing [19].

*2.1.2 Black-box.* For black-box testing, the tester is not given any information. The tester has to perform planning and reconnaissance in order to collect the information needed to perform the following steps. Performing black-box tests requires a lot of time, but is also the most realistic recreation of how a real attack may come about.

The disadvantages of this type of penetration testing are that it is possible to overlook vulnerabilities that are present in the system. The vulnerabilities could remain undiscovered since the pentester does not have time to excessively iterate through all the possible attack vectors present in the system.

*2.1.3 Gray-box.* When performing gray-box we are given some access to the network/system as well as *some* prior knowledge of the network/system, e.g. login credentials for a normal user, application flow charts, or network infrastructure maps [19].

Using this initial information can help create a more optimized plan and approach for performing the penetration test. This will help save some time on the first step of the penetration test (step 1 above) which lets the pentesters spend more time testing the systems which seem to be of higher risk instead of looking for attack vectors.

*2.1.4 White-box.* White-box is sometimes referred to as *clear* or *glass-box* pentesting as it gives us full insight into the system and network details including access to application source code, network maps, and credentials to all levels of users.

This drastically decreases the time the pentesters have to use for the first steps of the pentest which at the same time increases the time for the last steps which includes attacking discovered attack vectors. A white-box pentest is therefore a good way of evaluating as many attack vectors as possible.

## 2.2 Markov Decision Process (MDP)

A Markov decision process is a mathematical way of simulating the environment around us into a state-space and an action-space. It is a discrete-time stochastic process and is used as a model for cases where we have some degrees of randomness.

A simple example of an environment that can be simulated as an MDP could be traveling to work. We start in state "home" and want to arrive at state "work". We have 3 different actions to choose from when we are in state "home", namely: "bike", "car" or "bus". Let's say we pick the action "bus"; we walk to the bus stop. Now there is a 0.8 chance the bus is ready to leave from the bus stop, and there is a 0.2 chance we have to wait for the bus to arrive. The bus is not here yet, so we have to wait for one time-step (hence discrete time), after this time step, the bus has either arrived or not arrived. The bus arrives after 3 *waits* and we arrive at work.

The goal of MDP is to find a policy that **optimizes** this process. This policy should give us the optimal action to take from a given state. This optimization is often based on **rewards**. Setting rewards for each state is a way of simulating what is the desired behavior inside the process. If we go back to the "commuting to work" example; we would have negative rewards proportional to the amount of time the different paths to the workplace would take. After optimizing the process, we would have ended up with an MDP policy choosing the right actions based on what state you are in, in order to get to work the quickest.

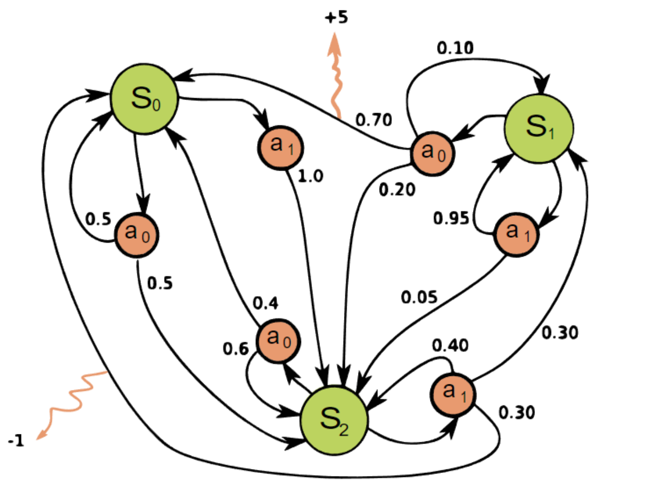


**Figure 2.2.1: Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows). Figure taken from [25]**

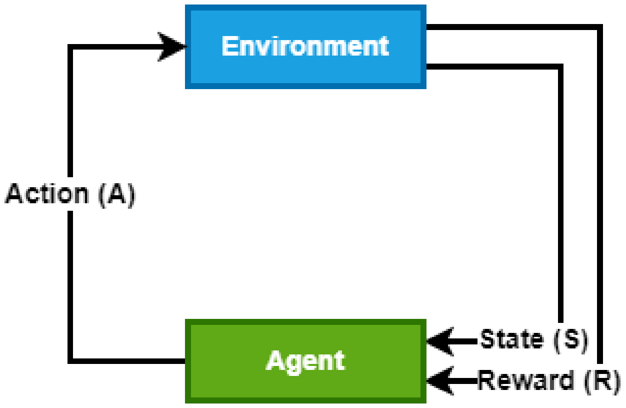
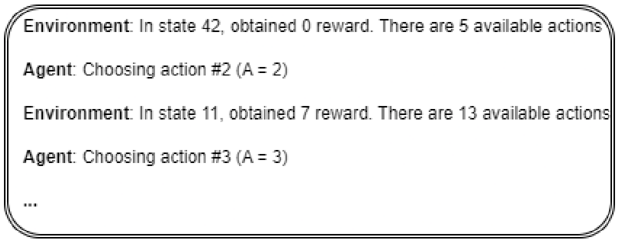


**Figure 2.3.1: Flowchart of RL algorithm**

## 2.3 Reinforcement Learning

Reinforcement learning is one of the main categories of machine learning. It consists of unknowing agents which are placed inside a virtual interactive environment [2]. The agents are playing a "turn-based game", where it has to pick an action from a predetermined list of possible actions each turn. The idea is that the agents will learn based on the feedback from their actions and experiences inside this environment. The agents will get rewards if the choice it makes leads to an environment state that is programmed to yield rewards (e.g. winning in a game).



**Environment**: In state 42, obtained 0 reward. There are 5 available actions

**Agent**: Choosing action #2 (A = 2)

**Environment**: In state 11, obtained 7 reward. There are 13 available actions

**Agent**: Choosing action #3 (A = 3)

...

**Figure 2.3.2: Dialog-box giving an example of what is happening in figure 2.3.1**

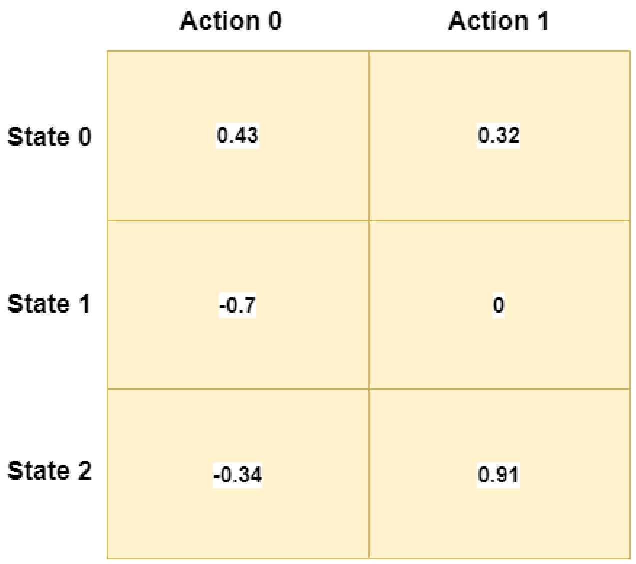
There are many problems where we previously believed humans to be superior, where now reinforcement algorithms have surpassed us. Examples of this are games like chess, Go, and shigo (Japanese chess) [22]. These results were achieved by agents playing multiple games against themselves, starting with only the rules of the game, and no previous human knowledge.

The goal is that the agents will learn a pattern of actions that maximize the reward given. It is very important to choose the right reward policies that will make the agents learn what we want them to learn [16].

Reinforcement learning models have some benefits compared to neural networks used for classification problems. One of them is that there is a more general problem formulation compared to the other branches of machine learning: supervised and unsupervised classification problems. As long as you can represent your environment (state space) in terms of a matrix and vectors, give the agents a set of actions represented as a vector (called action vector), and formulate a reward of some sort that learns the agent a correlation for which actions to pick from which environment states, you should be good to go.

Yet, you should be careful when formulating the reward policy and the environment, you could experience *reward hacking*. Reward hacking is when your agent interacts with the environment in such a way that it gets the most reward possible, but not in a way that solves your problem [9]. E.g. if you have an RL robot that you want to train to close windows in your apartment, and you give it (during training) rewards every time it does so. An example of reward hacking could be that it asks your neighbor to open your windows, just so that it can close them over and over again. This will yield huge amounts of reward, but it will not be the desired solution as your windows will be opened and closed continuously.

In order to properly train a reinforcement learning algorithm, one needs a lot of data. For many cases, just getting enough data may be the biggest problem, and may be one of the reasons that some of the areas where reinforcement learning has excelled the most, are games. Training an agent by making it play games against itself is a great way of generating enough data for the agent. Another great method for generating enough data is to create a simulation that imitates the real world, an example of this is the use of stochastic models to generate traffic flow on roads instead of using actual data. When using simulations instead of real data, it is important to make sure the statistical models you use are as realistic as possible in order for the agent to become as good as possible.



| | Action 0 | Action 1 |
|---|---|---|
| State 0 | 0.43 | 0.32 |
| State 1 | -0.7 | 0 |
| State 2 | -0.34 | 0.91 |

**Figure 2.3.3: Example simple Q-table (with random values) for figure 2.2.1**

*2.3.1 Q-Learning.* Q-learning is the most common RL algorithm. It is very similar to the MDP except for the fact that we Q-learning we are not able to precisely predict the reward and transition probability for each state. In MDPs, we assume we know all the transition probabilities, and rewards for each step; this lets us "solve" the MDP by the use of dynamic programming [3].

The goal behind Q-Learning is that we want to try out all the different actions from all the different states until we reach a convergence for each state, action pair and how much reward it would yield. Thus resulting in a so-called Q-Table (see figure 2.3.3). Working with Q-table is not feasible for bigger problems containing multiple states as the table size of such a Q-table can become extremely large.

Imagine creating a Q-table for chess; there are more than $10^{44}$ different legal positions on a chess board [23], and each of them can represent a state in a Q-table, we will quickly run out of memory trying to make a Q-table for this, let alone the different actions for each state. We have to find another way of solving this problem.

*2.3.2 Deep Q-Learning.* Deep Q-learning with Neural networks (DQN) solves said problem. Instead of using a table containing a Q-value for each state-action pair, it uses neural networks to provide an estimate for Q-values for each action possible from the state you are at.

## 2.4 Automated Penetration Testing

Until recent years, pentesting has solely been performed by proficient pentesters that have multiple years of experience in their domain. Not only is this process both money- and time-consuming, but professional penetration testers are also hard to come by [1].

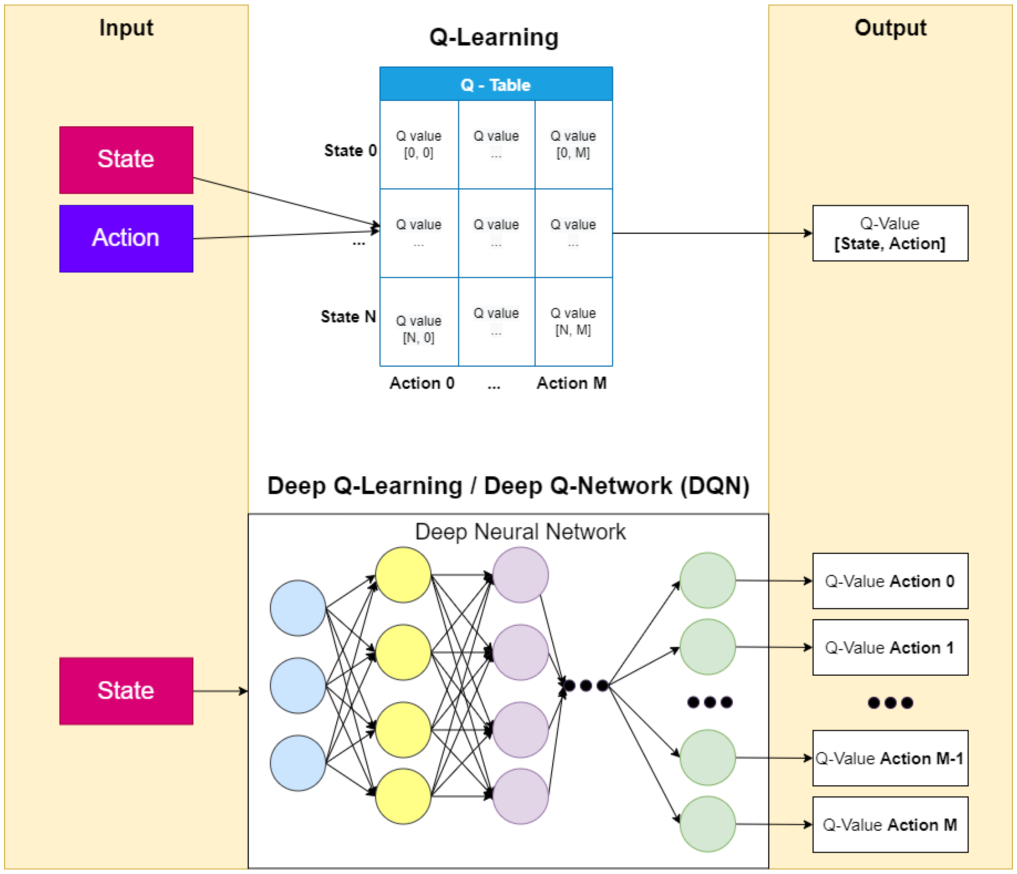


**Figure 2.3.4: Shows the difference between "normal Q" learning and DQN**

In short; penetration testing includes testing a lot of different tools and areas in a system, some of these can be automated, whereas others can not. An automated penetration test is usually conducted in a black-box manner where the first step is to scan the network/web server for potential attack vectors, i.e. open ports on a web server or a list of hosts that one could connect to inside the network. When performing the initial scan of the host(s) it is often used a scanning-tool such as Nmap [17]. Nmap can also give OS and system information letting us know which software and version numbers are run on different hosts/ports. How one continues the "attack" depends on the output of the scanning phase and on which type of automated pentest we are running. There are different methods of performing automated penetration testing; we can use a trained reinforcement learning agent that dictates the actions to perform based on the state it is currently in, or we can use a pure rule- or iteration-based approach which is more of a strict iteration through a number of predefined steps.

One of the greatest advantages to using automated penetration testing is the time benefit for the pentesters and the fact that you can run it in a routine manner to continuously check if there are any new vulnerabilities in the system [12]. They are performed as often as needed, making sure to identify security issues as soon as possible. As mentioned earlier; automated pentests are also usually executed in the initial step when security experts conduct a meticulous pentest in order to make an outline of the task at hand. Thus increasing their productivity, making them able to concentrate on "deeper" tasks that require human intervention [12].

## 2.5 Metasploit

Metasploit is an established open-source framework used for penetration testing which is commonly used [14]. It is a database containing modules for actions that can be used throughout the whole penetration test life cycle such as [7]. The most used types of modules are:

- **Exploits**
- **Payloads**
- **Auxiliaries**

**Exploits** are used to (as the name states) exploit a system often to a degree where one is able to run code on the attacked system (Remote Code Execution). A **payload** is often what you would want to run once you are able to exploit a system, a common type of payload is a reverse shell, which when executed lets the attacker maintain a 2-way communication method with the attacked system. An **auxiliary** module is a module that is not an exploit. This is a broad definition and includes a lot of interesting tools including vulnerability scanning, port scanning, fuzzing, and it gives you the opportunity to create your own auxiliary module.

Metasploit also keeps track of all the hosts you have discovered, as well as information about each host such as; state, exploit attempts, OS, service count, IP, and more [15]. This information about the hosts together with the exploits found in the Metasploit database is respectively what is used as a state space and action vector for the reinforcement learning *gym* we will use for the project [20].

## 2.6 MetasploitGym

The MetasploitGym is developed by Shane Caldwell which is a former pentester who transitioned into a career in machine learning. MetasploitGym [20] is used as a backbone of this project, we first want to give a brief overview of what it is before we continue to the next sections.

As stated in their github repository: "MetasploitGym is a gym environment designed to allow RL agents to interact with the Metasploit Framework's gRPC service to interact with networks and singular machines." [20]. gRPC is an open-source framework that allows for an easy-to-implement method of bidirectional communication across different languages and platforms [8] which is built into the Metasploit framework as default.

This framework lets you create agents that are able to interact with a state space via the Metasploit framework. There is a Random and Keyboard agents that can be used to experiment with the environment, keyboard agent in sense that you chose what the agent will do based on your keyboard input. There is also initialized a DQN agent that you can train in an environment consisting of either a stochastic simulation or a vulnerable machine(s).

*2.6.1 The State Space.* As mentioned in section 2.5, the Metasploit framework keeps a track of all the hosts discovered, as well as information about each host. This is used as a state space for this framework [4] and is represented as a matrix where each row is a **host vector**, and each column is a property of that host (i.e. IP-address, OS, ...). For each step, when we use our agent, it expects a state space matrix of the same size each time. This gives us a limitation for how many hosts one is able to keep track of in the state space at any given time. The state space matrix is created on a per-network basis, for this project, multiple state spaces will not be relevant.

The **host vector** represents the known information about a specific host. As the state space is initialized, all the values are set to 0. After the agent has scanned the network, values (firstly the IP addresses of the found hosts) are added to the host vectors accordingly. Most of the values are represented by a binary 0 and 1, whilst some of them can have other integer values i.e. the *Loot* value (see figure 2.6.2) representing the number of loot (basically files and information that act as proof of compromise) extracted.

The **feedback vector** contains feedback from the previous action. I.e. information about if the action chosen in the last time-step ran successfully or not and the amount of reward received [4]. There is also an **action vector** which represents an array of actions that one can choose from from a given state. The actions consist of exploits, privilege escalation methods, and scans.

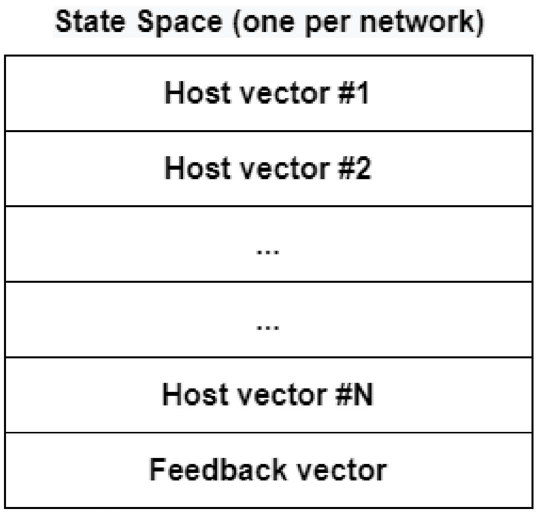
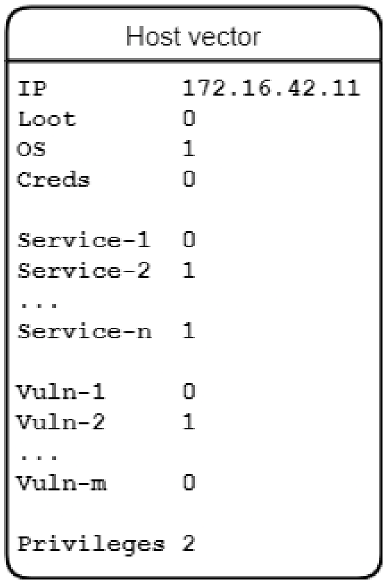


**Figure 2.6.1: MetasploitGym's state space**


Host vector

```
IP          172.16.42.11
Loot        0
OS          1
Creds       0

Service-1   0
Service-2   1
...
Service-n   1

Vuln-1      0
Vuln-2      1
...
Vuln-m      0

Privileges  2
```

**Figure 2.6.2: Host vector inside the state space**

## 3 METHOD

In this section, we present what was done in the project; The first phase where I familiarized myself with some of the basic concepts in penetration testing, and the second phase where I created a pipeline for the automation of this process.

## 3.1 Experimental phase

The goal of this first phase was to get more familiar with pentesting in general and which tools are available for performing one. A simple illustration of the environment can be seen in figure 3.1.1.

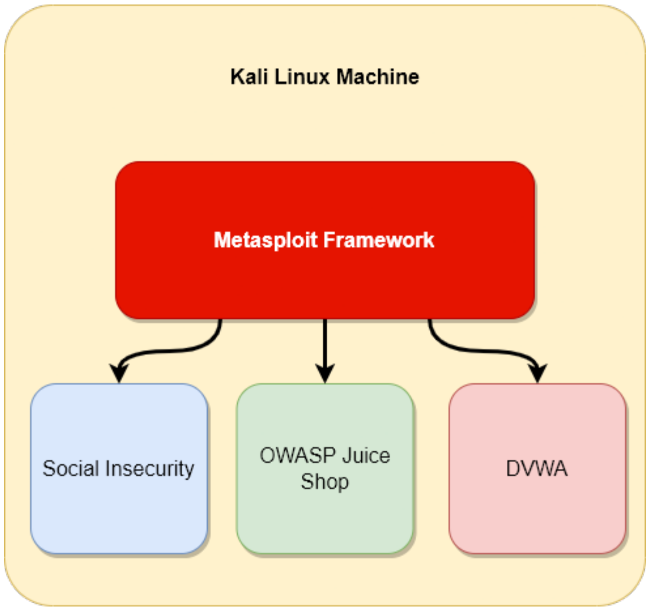


**Figure 3.1.1: Design of the "experimental phase"**

To set it all up, we used VirtualBox [24] and spun up a virtual machine with a Kali Linux Image to experiment with some of its built-in tools; such as Nmap for host- & port-scanning as well as Burpsuite which (among plenty of different use-cases) can be used for packet tempering or as a "fuzzing" tool used to check if endpoints were prone to SQL injection attacks.

The different insecure applications I used were DVWA [6], OWASP Juice Shop [18] and Social Insecurity [13]. Out of the three Vulnerable applications used, I ended up using DVWA the most. DVWA allows for a simple interface where you can choose which type of vulnerability you want to experiment with. I mostly experimented with **Command Injection**, **SQL Injection**, and **Brute Force**. Since DVWA is quite popular in the pentesting domain, there were a lot of easy-to-follow guides online which made the learning curve steeper.

## 3.2 Automating Pentesting by the use of Agents

After having learned and experimented with penetration testing and reinforcement learning, I better knew what was needed to build a complete pipeline for agent training and usage. I knew we needed an environment that our agent would be able to interact with (state space). We also needed a set of tools to give the agent possible ways of interacting with the environment. Inside the environment, we also needed machines/boxes that our agent could interact with using some of its tools.

## 3.3 Design

We ended up using MetasploitGym (section 2.6) as the RL environment. We pulled the GitHub repository [20] to the host computer and used this as the starting point of the whole pipeline. As for the action-space (the set of all possible actions to take at each time step); the MetasploitGym repository comes with some already-defined actions which assume you already have a gRPC connection to the Metasploit framework. The agent can choose to perform any of these actions, and there is also the possibility of adding more actions for the agent to take, either via the Metasploit framework, or specific personalized actions.

After having set up the "local" environment for the agent, we needed to create some virtual networks used for communication between the box running the Metasploit framework and the host, as well as an isolated network containing the vulnerable host(s). The reason we split the networks into two is to prevent the vulnerable hosts from being directly connected to the host and the internet, thus having them in the isolated network instead. By the use of VirtualBox and its possibility for creating personalized DHCP servers, we created a "Host-Only Network" having the address ‘192.168.56.0/24‘ and an "Isolated Network" that was given the address ‘10.38.1.0/24‘. The former is used as a connection between the host and the ubuntu box responsible for running the Metasploit framework whilst the latter is used to connect the vulnerable machines to the Metasploit framework box.

To create a box with the tools needed as an intermediate step between the agent's environment and the vulnerable machines, we needed to be able to run the Metasploit framework as well as being able to communicate with the agent. We found a docker image [21] (being maintained by Romain Dauby) that gives us the possibility of doing exactly this. It contains the Metasploit framework database, and it has the possibility of initializing a gRPC communication service that we can use to communicate with the agent's environment.

For the agent to have something to interact with, we needed some vulnerable boxes, or an endpoint simulating such boxes. SecGen [5] is an application using Ruby and Vagrant in order to spawn vulnerable boxes for VirtualBox in an on-demand manner. This application would have been perfect for this project as you could have continued to train your agent on new a box once it was able to penetrate (or fail n times) on the previous box. Even though I tried running this application from both Ubuntu and Windows, with tons of different configurations, I was unable to get it to work.

I instead opted for a simpler solution; downloading a virtual image of one of rapid7's vulnerable boxes; namely metasploitable-2, and initializing the box connected to the "Isolated Network". This virtual image has many different vulnerabilities and allows for the completion of the pipeline. This box may not be sufficient to properly train the agent, but if run long enough, it will learn to detect some of the vulnerabilities in the box.

Before running the agent, the agent has to be given the IP address and Port number as well as the credentials for the gRPC communication with the Metasploit framework. It also needs to know the IP address of the vulnerable host(s) on the isolated network. An illustration of the pipeline can be found in figure 3.3.1.

## 3.4 Implementation

To make sure all the parts of the pipeline were able to communicate with each other we had to create networks in VirtualBox. We earlier mentioned that the isolated network got an IP address range of 10.38.1.0/24 and that the host-only network got the range 192.168.56.0/24 and that we had to create a small DHCP server for

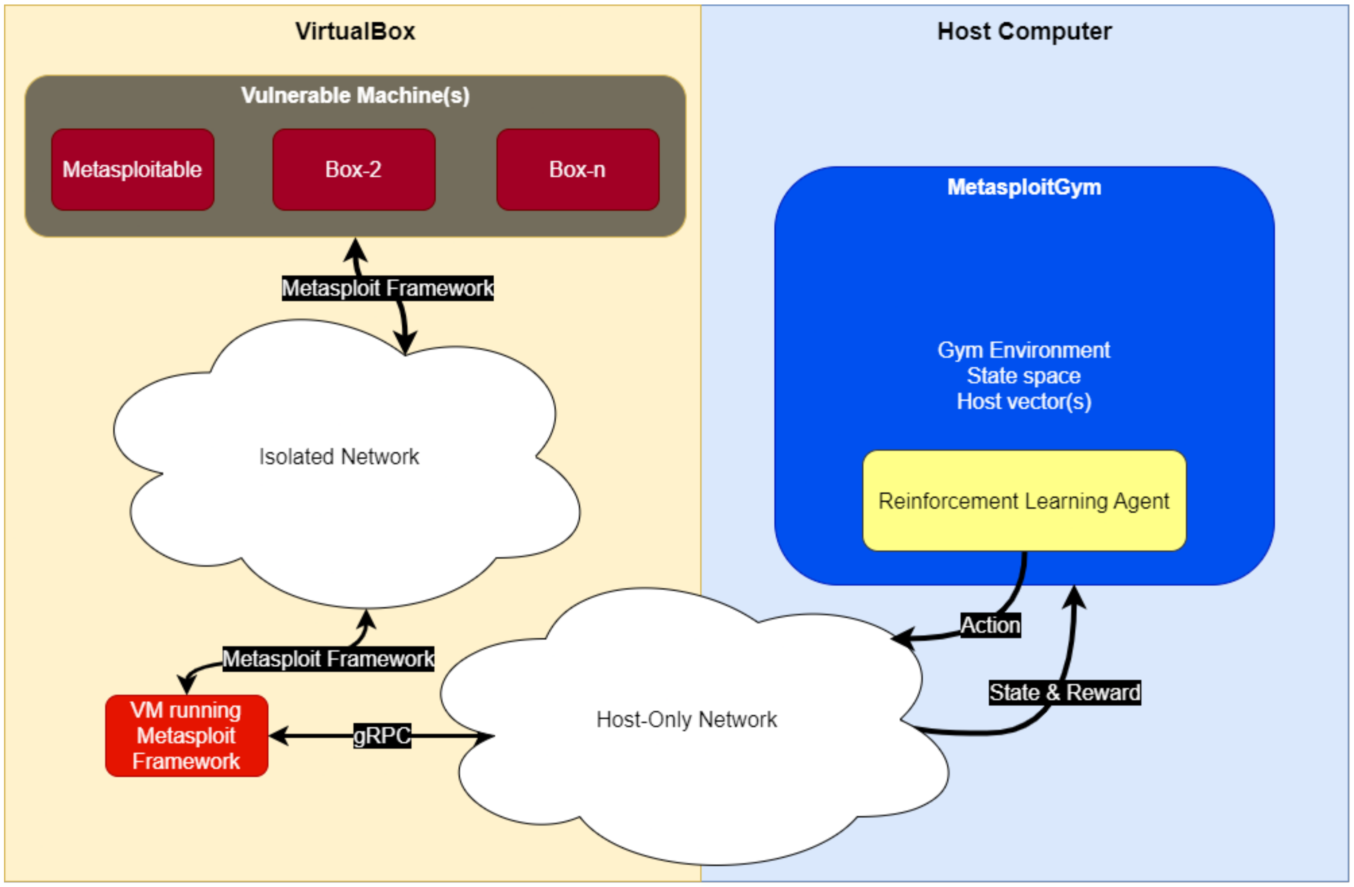


**Figure 3.3.1: Design of the Pipeline**

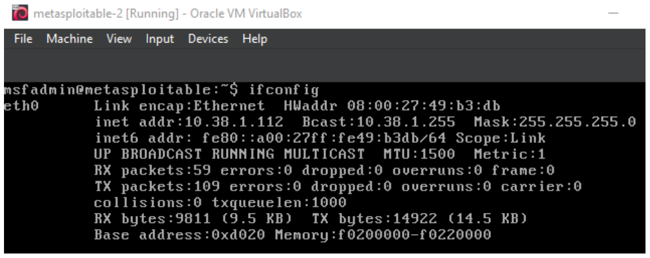
the networks. Having done this, we started the boxes from Virtual-Box and ran the command ‘ifconfig‘ to check which IP addresses they were given. These IP addresses were later added to a .env file inside the MetasploitGym environment for the agent to use when executing actions.

```
METASPLOIT_PASSWORD=skinnyhouse
METASPLOIT_PORT=9999
METASPLOIT_HOST=192.168.56.103
TARGET_HOST=10.38.1.112
```

**Listing 1: .env file to be read when creating the Metasploit-Gym environment**

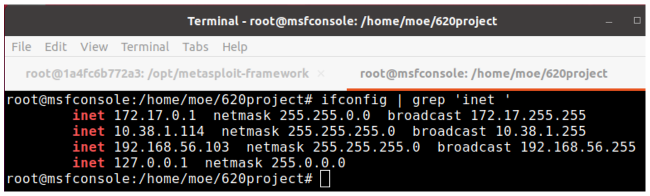
The MetasploitGym repository comes with some skeleton code for different types of agents; a **random agent**, a **keyboard agent**, and lastly a **DQN agent**. The random agent chooses actions at random, and can for future projects be used as a baseline. The keyboard agent lets the user choose which of the actions to take at each given step. Lastly, the DQN agent is an agent using Deep Q-Learning in order to learn and later predict which actions are clever choices at different positions/states.

```
INPUT num_runs
env <- generate_environment(file='.env')
```


metasploitable-2 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
msfadmin@metasploitable:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:49:b3:db
          inet addr:10.38.1.112  Bcast:10.38.1.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe49:b3db/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:59 errors:0 dropped:0 overruns:0 frame:0
          TX packets:109 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:9811 (9.5 KB)  TX bytes:14922 (14.5 KB)
          Base address:0xd020 Memory:f0200000-f0220000
```

**Figure 3.4.1: IP addresses of the vulnerable box**


Terminal - root@msfconsole: /home/moe/620project
File Edit View Terminal Tabs Help
root@1a4fc6b772a3: /opt/metasploit-framework
root@msfconsole: /home/moe/620project

```
root@msfconsole:/home/moe/620project# ifconfig | grep 'inet '
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet 10.38.1.114  netmask 255.255.255.0  broadcast 10.38.1.255
        inet 192.168.56.103  netmask 255.255.255.0  broadcast 192.168.56.255
        inet 127.0.0.1  netmask 255.0.0.0
root@msfconsole:/home/moe/620project# 
```

**Figure 3.4.2: IP addresses of the box running the docker image containing the MSF**

```
run_steps <- Array(Int)  # steps each run
run_rewards <- Array(Int)  # rewards each run
run_goals <- 0  # num times box penetrated
ms <- k: int  # max steps (actions) each run

FOR each run in num_runs
    run_output <- run_random_agent(env, ms)
    steps, reward, done <- run_output

    IF done
        run_goals = run_goals + 1

present_stats(run_steps, run_rewards,
              run_goals, num_runs)
```

**Listing 2: Pseudocode for the random agent. See listing 1 for .env file**

The listing (2) simply describes how the random agent works except from the **run_random_agent()** function. This function handles each of the agent runs. A run is defined as "either surpassing the max number of steps allowed" or "being able to penetrate the vulnerable machine". For each step in this function, an action is chosen; for the random agent, a random action is chosen, but for the DQN agent, an action is chosen based on the agent's internal neural network. After having performed the chosen action, it checks if **1**; it has gotten any new information about the box, and **2**; if it was successful in opening a console or meterpreter session with the target box. Both 1 and 2 give rewards (of course more for 2) but only 2 will be counted as having "penetrated" the box, and the run will stop.

After the agent has chosen an action to perform, the action details are sent to the IP:Port of the virtual machine hosting the docker image containing the Metasploit framework, and the agent starts waiting for a response. The docker image is accessible on the specified port after having activated the gRPC service from within the Metasploit framework (see my GitHub [11] or r0mdau's GitHub [21] for more details). After the docker image has received the action to perform, it finds the action in its database and performs the action directed at the given target-IP address received from the agent.

After the action is executed towards the target IP, the new (if changed, else the same) state is sent back to the gym environment where the rewards and "if penetrated" variables are calculated. Due to the limitations, I did not have time to implement the DQN agent in this project, however, this is part of the future work that will be discussed later.

## 4 EVALUATION

In this section, I want to present a trial of the baseline implementation using the random agent.

### 4.1 Experimental setup

For this experiment, I decided to use the box Metasploitable-2 running inside the isolated network (see fig 3.3.1) as the vulnerable box. The reason I chose this as the vulnerable machine is that it is

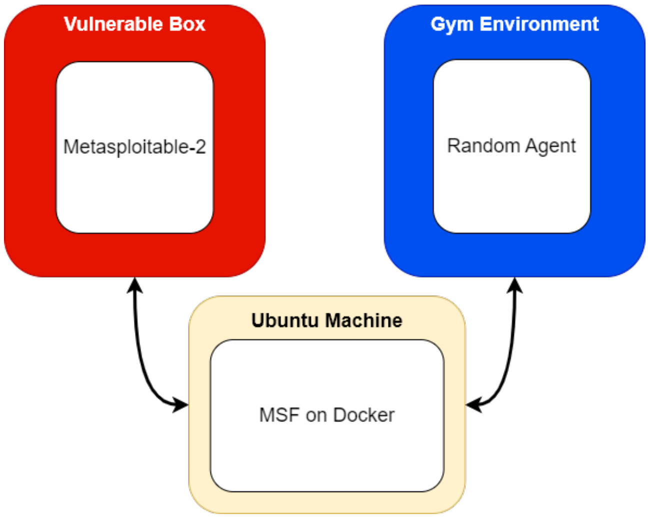
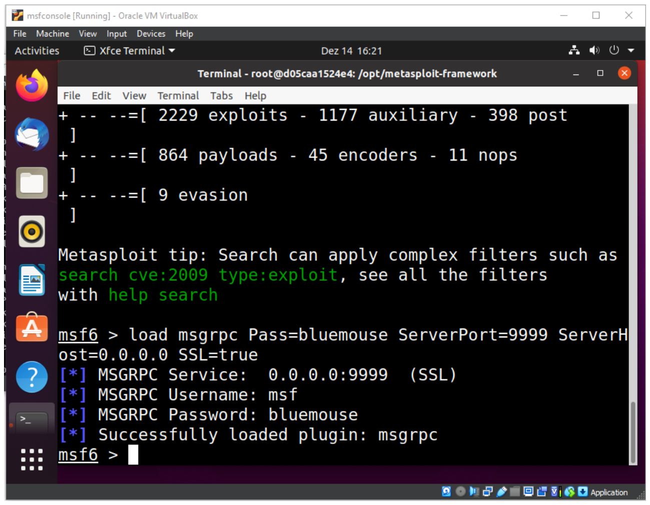


**Figure 4.1.1: Simple illustration of the "random agent" baseline experiment**

a highly vulnerable machine where multiple different attacks are able to penetrate it.

For the experiment, I use the parameters **num_runs** = **5** and **max_steps** = **10** (per run). Meaning the agent will try 5 times to penetrate the vulnerable machine, with a maximum of 10 actions each time before resetting the environment and resuming the next run.


msfconsole [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Xfce Terminal
Dez 14 16:21
Terminal - root@d05caa1524e4: /opt/metasploit-framework
File Edit View Terminal Tabs Help

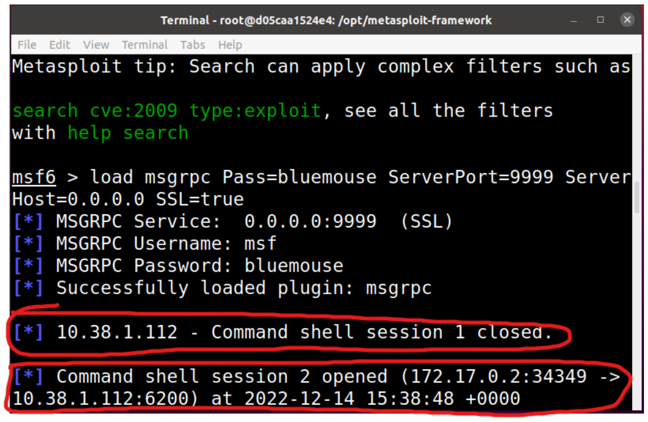
```
+ -- --=[ 2229 exploits - 1177 auxiliary - 398 post
 ]
+ -- --=[ 864 payloads - 45 encoders - 11 nops
 ]
+ -- --=[ 9 evasion
 ]

Metasploit tip: Search can apply complex filters such as
search cve:2009 type:exploit, see all the filters
with help search

msf6 > load msgrpc Pass=bluemouse ServerPort=9999 ServerH
ost=0.0.0.0 SSL=true
[*] MSGRPC Service:  0.0.0.0:9999  (SSL)
[*] MSGRPC Username: msf
[*] MSGRPC Password: bluemouse
[*] Successfully loaded plugin: msgrpc
msf6 >
```

**Figure 4.1.2: Starting the gRPC service on the Docker image running Metasploit framework on the virtual machine**

Starting the gRPC service with the IP address 0.0.0.0 and port 9999 will let all incoming traffic to the virtual machine addressed to port 9999 be routed to the docker image running the framework. When both the virtual machines are running, I run the **random_agent.py** script found in my repository [11] (which is a slight modification of the original [20]) and waited for the results to be printed out.

## 4.2 Results

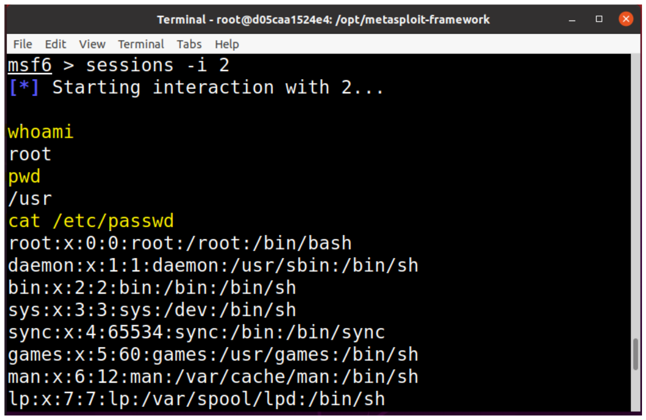
While running the experiment, the baseline agent managed to get root access in two of the five runs by choosing actions at random.



Terminal - root@d05caa1524e4: /opt/metasploit-framework
File Edit View Terminal Tabs Help

```
Metasploit tip: Search can apply complex filters such as

search cve:2009 type:exploit, see all the filters
with help search

msf6 > load msgrpc Pass=bluemouse ServerPort=9999 Server
Host=0.0.0.0 SSL=true
[*] MSGRPC Service:  0.0.0.0:9999  (SSL)
[*] MSGRPC Username: msf
[*] MSGRPC Password: bluemouse
[*] Successfully loaded plugin: msgrpc

[*] 10.38.1.112 - Command shell session 1 closed.

[*] Command shell session 2 opened (172.17.0.2:34349 ->
10.38.1.112:6200) at 2022-12-14 15:38:48 +0000
```

**Figure 4.2.1: You can see the output inside the Metasploit framework as the experiment is running, here you see a reverse shell achieved to the target machine**



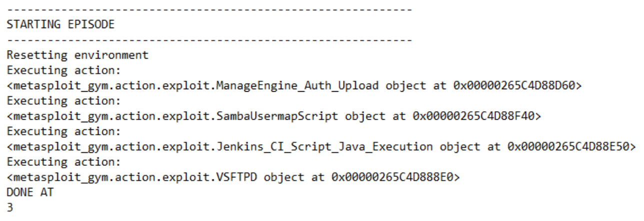
Terminal - root@d05caa1524e4: /opt/metasploit-framework
File Edit View Terminal Tabs Help

```
msf6 > sessions -i 2
[*] Starting interaction with 2...

whoami
root
pwd
/usr
cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
```

**Figure 4.2.2: Interacting with one of the reverse shells (session 2) achieved during the experiment**

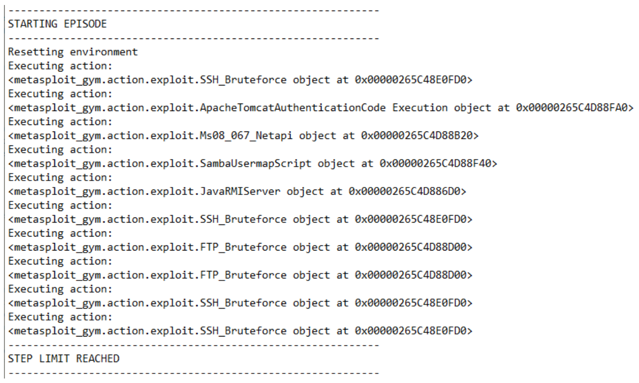
Based on the outputs from the successful runs (one of them in figure 4.2.3), we see that the VSFTPD exploit were the only exploit able to fully penetrate the machine. Had we exhaustively iterated through all the possible exploits/actions, there is a possibility that we could have found more vulnerabilities leading to getting a reverse shell on the machine.

Due to limitations for the project I have not been able to dive into the ranking and distribution of rewards for the pipeline and am therefore not able to properly interpret the metrics received as an output in figure 4.2.5.



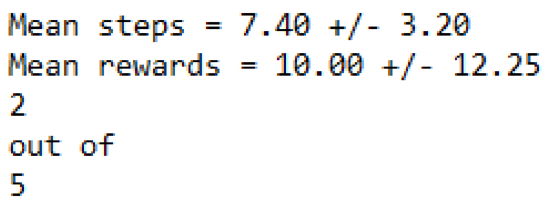
```
------------------------------------------------------------
STARTING EPISODE
------------------------------------------------------------
Resetting environment
Executing action:
<metasploit_gym.action.exploit.ManageEngine_Auth_Upload object at 0x00000265C4D88D60>
Executing action:
<metasploit_gym.action.exploit.SambaUsermapScript object at 0x00000265C4D88F40>
Executing action:
<metasploit_gym.action.exploit.Jenkins_CI_Script_Java_Execution object at 0x00000265C4D88E50>
Executing action:
<metasploit_gym.action.exploit.VSFTPD object at 0x00000265C4D888E0>
DONE AT
3
```

**Figure 4.2.3: Output of a successful run after 3 steps (yes, we count from 0 :))**



```
------------------------------------------------------------
STARTING EPISODE
------------------------------------------------------------
Resetting environment
Executing action:
<metasploit_gym.action.exploit.SSH_Bruteforce object at 0x00000265C48E0FD0>
Executing action:
<metasploit_gym.action.exploit.ApacheTomcatAuthenticationCode Execution object at 0x00000265C4D88FA0>
Executing action:
<metasploit_gym.action.exploit.Ms08_067_Netapi object at 0x00000265C4D88B20>
Executing action:
<metasploit_gym.action.exploit.SambaUsermapScript object at 0x00000265C4D88F40>
Executing action:
<metasploit_gym.action.exploit.JavaRMIServer object at 0x00000265C4D886D0>
Executing action:
<metasploit_gym.action.exploit.SSH_Bruteforce object at 0x00000265C48E0FD0>
Executing action:
<metasploit_gym.action.exploit.FTP_Bruteforce object at 0x00000265C4D88D00>
Executing action:
<metasploit_gym.action.exploit.FTP_Bruteforce object at 0x00000265C4D88D00>
Executing action:
<metasploit_gym.action.exploit.SSH_Bruteforce object at 0x00000265C48E0FD0>
Executing action:
<metasploit_gym.action.exploit.SSH_Bruteforce object at 0x00000265C48E0FD0>
------------------------------------------------------------
STEP LIMIT REACHED
------------------------------------------------------------
```

**Figure 4.2.4: output from one of the unsuccessful runs**



```
Mean steps = 7.40 +/- 3.20
Mean rewards = 10.00 +/- 12.25
2
out of
5
```

**Figure 4.2.5: Calculated metrics after performing the experiment, the number after '+/-' is the standard deviation across the different runs**

## 5 CONCLUSION

In this project, I have merely put together different tools in order to create a pipeline that can be polished and built upon to properly train a Deep Q-Learning agent for the use of penetration testing. I am able to run through the proposed pipeline using a simpler agent, i.e. the random agent or the keyboard agent. But there is still some work to do in order to get the RL agent up and running.

I would have loved to finish the whole potential of this project, but due to the limitations of this being a 10p course this semester, I was not able to allocate time enough to do so.

### 5.1 Future Work

For future work I would consider the following points:

- Finish the pipeline in order for the Deep Q-Learning agents to run and be able to learn from the environment.
- Build upon the possible actions which are already in the MetasploitGym. Increasing the selection of exploits, scans, and possible ways of doing privilege escalation.
- Find a way to generate vulnerable machines on-demand so that the agent has plenty of machines to train on.
- Or, as an alternative to the last point, create an array of simulated machines with an interface mimicking a real vulnerable machine prone to exploits from the Metasploit framework. I think this would be sufficient and a lot faster. Fine-tuning the agent on real machines would probably be beneficial after the initial training.

## 6 FINAL WORDS

I would like to thank my supervisor, Ferhat Ȯzgur Catak, for the possibility to work on this project. It gave me the opportunity to deeper study the subjects of both penetration testing and reinforcement learning, as well as how they can be used together.

## REFERENCES

[1] Farah Abu-Dabaseh and Esraa Alshammari. 2018. Automated penetration testing: An overview. In *The 4th International Conference on Natural Language Computing, Copenhagen, Denmark*. 121–129.
[2] Shweta Bhatt. 2018. Reinforcement Learning 101. https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292
[3] blackburn. 2020. Reinforcement Learning: Solving Markov Decision Process using Dynamic Programming. https://towardsdatascience.com/reinforcement-learning-solving-mdps-using-dynamic-programming-part-3-b53d32341540
[4] Shane Caldwell. [n. d.]. "Training an Autonomous Pentester with Deep RL" by Shane Caldwell. https://www.youtube.com/watch?v=EiI69BdWKPs
[5] cliffe. 2014. Security Scenario Generator (SecGen). https://github.com/cliffe/SecGen
[6] digininja. 2012. DAMN VULNERABLE WEB APPLICATION. https://github.com/digininja/DVWA
[7] EngEd. 2021. Getting Started with the Metasploit Framework. https://www.section.io/engineering-education/getting-started-with-metasploit-framework/
[8] gRPC. 2022. gRPC. https://grpc.io/
[9] Felix Hofstätter. 2022. How to stop your AI agents from hacking their reward function. https://towardsdatascience.com/how-to-stop-your-ai-agents-from-hacking-their-reward-function-5e26fc006e08
[10] Impreva. 2022. Impreva penetration testing. https://www.imperva.com/learn/application-security/penetration-testing/
[11] Håvard Moe Jacobsen. 2022. Reinforcement Learning for penetration testing. https://github.com/havardMoe/620project/
[12] Borislav Kiprin. 2021. What Is Penetration Testing Software? Best Tools to Use. https://crashtest-security.com/best-automated-penetration-testing/
[13] MagnusBook. 2019. social-insecurity. https://github.com/MagnusBook/social-insecurity/
[14] metasploit. 2022. metasploit. https://www.metasploit.com/
[15] metasploit. 2022. metasploit docs - hosts. https://docs.rapid7.com/metasploit/scanning-and-managing-hosts/
[16] Chris Nicholson. [n. d.]. A Beginner's Guide to Deep Reinforcement Learning. https://wiki.pathmind.com/deep-reinforcement-learning
[17] nmap. 2022. nmap. https://nmap.org/
[18] OWASP. 2016. OWASP Juice Shop. https://owasp.org/www-project-juice-shop/
[19] Packetlabs. 2022. Black-Box vs Grey-Box vs White-Box Penetration Testing. https://www.packetlabs.net/posts/types-of-penetration-testing/
[20] phreakAI. 2021. MetasploitGym. https://github.com/phreakAI/MetasploitGym
[21] r0mdau. 2015. Metasploit Framework dockerfile. https://github.com/r0mdau/dockerfile-msf
[22] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).
[23] John Tromp. 2021. Chess Position Ranking. https://github.com/tromp/ChessPositionRanking
[24] ViritualBox. 2022. ViritualBox. https://www.virtualbox.org/
[25] waldoalvarez. 2017. Markov Decision Process. https://en.wikipedia.org/wiki/Markov_decision_process#/media/File:Markov_Decision_Process.svg This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.