UCLouvain

École polytechnique de Louvain

# Automation of a cyber range

## Vagrant and CVE program for the creation of a vulnerable system

Author: **Amin ABDELKEFI**
Supervisor: **Ramin SADRE**
Readers: **Axel LEGAY, Ayham KASSAB**
Academic year 2020–2021
Master [120] in Cybersecurity

# Acknowledgements

First and foremost I am extremely grateful to my supervisor, Professor Ramin Sadre for his invaluable advice, continuous support, and patience during my master thesis. His guidance helped me in all the time of research and writing of this thesis. Since the first meeting where he proposed a subject different from the initial one in order to best fit my curriculum and during the various exchanges where he helped me by proposing multiple solutions or alternatives, I knew there was someone who constantly trusted and supported me.

I would like to acknowledge the two readers of my thesis Professor A. Legay and Professor A. Kassab for taking the time to read my work.

Finally, I would like to express my gratitude for the support I received from my family and my close friends throughout this time. It was not always easy for me to deal with all of the issues I faced throughout these months, and their guidance and support were invaluable in helping me overcome these obstacles.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Context and objectives

With the day by day digitalization of our world, the number of cyber-attacks increases. To counter these attacks, cybersecurity experts are needed. However, the current number of cybersecurity experts is not sufficiently large. It is therefore essential to train future cybersecurity experts properly and to place them in the best possible conditions, as close as possible to real cases. The situations in which students learn the most are when they are directly confronted with real-life cases. They gain practical experience in using security tools and techniques to target and protect vulnerable systems.

The most common practices to meet these needs are practical lab work, pre-configured hacking challenges, Capture the Flag (CTF) and pre-configured virtual machines. All of these practices provide good learning opportunities and allow students, but also experts, to gain experience. However, developing these solutions can quickly become time consuming.

Another popular solution is cyber ranges. Cyber ranges are dynamic, simulated networks of systems, software, and application [1]. They usually provide a secure environment for product development and security testing, as well as a secured, legal environment to gain cyber skills. Cyber range can help promote and encourage cybersecurity education, training, and certification. Actual hardware and software, as well as a mix of real and virtual components, can be used in these critical tools.

The objective of this thesis is to implement a tool for the automatic generation of a cyber range, represented here by a virtual machine.

The goal of this tool is firstly, to provide an educational vulnerable system for students to use to gain real case experience and secondly, to create a tool that

is easy to use and that can generate a personalized, scalable system based on determined conditions.

The vulnerable system is represented by the generation of a virtual machine with a number of vulnerabilities defined with the help of the CVE (Common Vulnerabilities and Exposures) Program and tools using CVEs such as CVE-Search and Debsecan.

The use of a vulnerable system is necessary for learning. Using a virtual machine to represent this system will allow the student not to worry about the consequences of the actions performed on the machine. Moreover, the integration of vulnerability will allow the student to develop his problem solving skill as well as his vulnerability research method.

As mentioned above, the integration of these vulnerabilities will be done with the CVE program.

CVE is the abbreviation of Common Vulnerabilities and Exposures. It is a dictionary of names for publicly known information system vulnerabilities [2].

The CVE program is used to allow the recovery of the list of vulnerabilities present in a system as well as to determine vulnerabilities to be injected into the virtual machine.

## 1.2 Plan

The thesis is structured as follows:

Chapter 2, *Background knowledge*, outlines the primary concepts needed to understand the rest of the thesis correctly.

Chapter 3, *State of the art*, mainly consists of the results of scientific papers read. It presents and highlights the different existing methods of generating a cyber range or a cybersecurity problem.

Chapter 4, *Concept*, proposes and explores some concepts and tools used to achieve the objective of this thesis.

Chapter 5, *Our approach*, provides a detailed description of the problem and the final objective to be reached. It also describes the solution found and the algorithms implemented or used to meet the objective.

Chapter 6, *Experimental results*, describes how the tool was tested, the interesting results observed and the problems encountered during the implementation.

Chapter 7, *Future work*, proposes a list of potential ideas and amelioration for the tool.

Chapter 8, *Conclusion*, resumes and concludes this master thesis.

## 1.3 Contribution

The main contribution of our work is the implementation of a tool allowing the automation of a vulnerable system based on the catalogue of publicly disclosed cybersecurity vulnerabilities that can be found using the CVE program (Common Vulnerability and Exposures). The contribution is the concept presented in the chapter 4 and the algorithms implemented in chapter 5.

## 1.4 Notations

**CAPEC** Common Attack Pattern Enumeration and Classification

**CNA** CVE Numbering Authority

**CPE** Common Platform Enumeration

**CR** Cyber Range

**CTF** Capture the Flag

**CVE** Common Vulnerabilities and Exposures

**CVSS** Common Vulnerability Scoring System

**CWE** Common Weakness Enumeration

**NIST** National Institute of Standards and Technology

**NVD** National Vulnerability Database

**VIA4CVE** Vulnerability Information Aggregator for CVEs

**VM** Virtual Machine

# Chapter 2

# Background knowledge

This chapter aims to define the key concepts used in some chapters of this master thesis. A quick description of the Capture the Flag competitions is given followed by a definition of a cyber range, its usage and its pros and cons.

Another concept presented in this chapter is CVE. Again, some definitions are given to help understand this concept followed by the main idea and an illustration of the way it works.

## 2.1 Capture the flag

Cybersecurity competitions are interesting events that are growing in popularity among cybersecurity professionals. A "Capture the Flag" (also known as CTF) competition is a popular event type that may be found both online and in-person in many areas [3]. It challenges the players to complete a number of tasks ranging from a Wikipedia scavenger hunt to simple programming exercises to hacking into a server to steal data. The competitor is usually requested to find a certain piece of text that is concealed on the server or behind a web page. The goal is mainly known as the flag, therefore the name of the competition. These competitions are open to all, both professionals and students.

### 2.1.1 Types of CTFs

There are 2 types of CTFs that are very prevalent [4]:

- **Jeopardy-style CTFs**
  This style gives a list of challenges and reward points to individuals or teams who complete them. The points given depends on the complexity of the task and only once a team solves the previous task in chain can the next

task be opened. At the end of the time, the team with the most points wins. **Jeopardy-style** CTFs challenges are typically divided into categories such as *Cryptography* (decrypting/encrypting a piece of data), *Steganography* (find information hidden in files or images), *Binary* (reverse engineering or exploiting a binary file), *Web* (exploiting web pages to find the flag) and *Pwn* (exploiting a server to find the flag).

- **Attack-defence**
  Another interesting type of competition is **Attack-defence**. Each team has its own network (or just a host) with vulnerable services. In most cases, a certain amount of time is given to the teams to fix some vulnerable services and to develop exploits. Then all participants are connected to each other and the wargame begins. Defence points are awarded for protecting services and attack points for attacks on other participants' networks/hosts.

### 2.1.2 Benefits

These competitions are a good way to gain experience and enrich a resume. For an employer, having won or participated in a CTF will show that the person is interested in this field and wishes to develop his knowledge.

These competitions are also an opportunity to exchange with other competitors, coaches or even sponsors to learn more about the possibilities that cybersecurity offers in different companies.

In addition, since many CTFs are held by cyber companies, they may take advantage of these competitions to do some talent scouting. These events are also a great way to meet and make new contacts in the cybersecurity industry, find possible employment opportunities, and learn about what is new in the industry from cybersecurity experts.

## 2.2 Cyber range

With the increase in the number of cyber-attacks, security experts need to be well-prepared and equipped with the right security tools. A popular solution is the use of cyber range.

### 2.2.1 Definition

A cyber range is a virtual environment that is used by companies for cyberwarfare training and software development [5]. These complex IT environments are necessary for countering modern cybercrime because they allow firms to practice

handling specific real-world scenarios, train staff and consumers on the latest threats. Military and government agencies, private enterprises (software development and cybersecurity), and other privates entities with a focus on cybersecurity are actively using cyber ranges.

### 2.2.2 Usage

There are numerous factors that influence the development of cyber ranges. For one thing, it is impossible to learn everything there is to know about cyber defence in a classroom. On the other hand, the real world is not suitable for this type of education. Allowing students to develop cyber skills on production systems with real data is far too dangerous. Furthermore, the chances of a teachable incident occurring on a schedule that coincides with a training program are exceedingly slim. It is possible to wait months for a significant cyber-attack to occur; but it is critical to be ready when it does. Hence, the range.

As said [6], there are many reasons that will lead to the creation of a cyber range such as:

- The growth of highly advanced, ever-evolving attack vectors
- A need for training that replicates different kinds of attacks
- A need for readiness
- Testing incident response plans
- Training cybersecurity professionals

### 2.2.3 Benefits

With cybercrime on the rise, more training is needed to prepare businesses and government agencies to deal with new threats as they emerge [7].

Virtual cyber ranges have a number of benefits that might assist a company to deal with potential threats. These benefits include:

- **Experiencing real-world threats in a safe environment**
  Cyber ranges give staff and consumers a chance to experience real-world threats in a virtual environment.
- **Learn how to recognize and respond to threats**
  Actively train the staff and customers against modern threats in order for them to learn how to identify and deal with potential threats.

- **Validate proof of concepts (POC)**
  Cyber range applications can be used by software engineers to validate virtual proofs of concept. Cyber ranges are ideal for confirming new ideas, proving market viability, and more due to their unlimited testing possibilities.
- **Save time and money**
  Cybersecurity labs are a cost-effective method of ensuring cybersecurity. They are easy to set up, involve minimal capital, and only need to be done once.
- **Virtual environments are constantly updated**
  Updates, fixes, and exploits are constantly being released. A virtual cyber range is always up to date and ready to assist the company in testing against any threats they desire.

## 2.3 Common Vulnerabilities and Exposures

Common vulnerabilities and Exposures (CVE) is a free to use list of publicly known information security vulnerabilities and exposures [8]. The MITRE corporation introduced CVE in 1999 to define and categorize software and firmware vulnerabilities. For companies looking to enhance their cybersecurity, CVE offers a free dictionary. MITRE is a nonprofit organization in the United States that manages federally funded research and development centres.

### 2.3.1 Definition

As stated above, CVE provides a list of known vulnerabilities and exposures. A vulnerability can be described as a weakness in a computer system that can be used in a cyber-attack to obtain unauthorized access or to perform unauthorized actions on it [9]. Attackers can use vulnerabilities to run code, gain access to system memory, install various types of malware and steal, destroy or change sensitive data.

An exposure occurs when a user makes a mistake allowing an attacker to gain access to a device or network. Exposures can result in data breaches, data leaks and the sale of personally identifiable information (PII) on the dark web. In fact, rather than sophisticated cyber-attacks, some of the largest data breaches were triggered by unintentional disclosure.

### 2.3.2 Objective

CVE's mission is to make it easier for organizations to exchange knowledge about known vulnerabilities. CVE accomplishes this by assigning a unique identifier to

each vulnerability or exposure. CVE identifiers, also known as CVE names, enable security professionals to search for information about specific cyber threats through various sources using a single name.

### 2.3.3 Determination of a CVE

CVE IDs are assigned to flaws that follow a set of criteria [10]. They have to be:

- **Independently fixable**
  The flaw can be patched regardless of whether or not there are any other bugs.
- **The vendor has acknowledged it and it has been documented**
  The bug is acknowledged by the software or hardware manufacturer, and it has a negative effect on security. Alternatively, the reporter would have shared a vulnerability report demonstrating the bug's negative effect as well as the fact that it violates the affected system's security policies.
- **Having an effect on a single codebase**
  Flaws that affect more than one product get separate CVEs. In case of shared libraries, protocols or standards, only if there is no way to use the shared code without begin vulnerable does the bug receive a single CVE. Otherwise, a unique CVE is assigned to each affected codebase or product.

### 2.3.4 Benefits

The benefits of CVE are [9]:

- All organizations benefit from sharing CVE information because it helps them establish a baseline for assessing the coverage of their security tools. CVE numbers allow organizations to see what each of their security tool covers and if it is suitable for their needs.
- Organizations may easily and reliably collect information about a specific vulnerability or exposure by using the CVE ID, and organize their efforts to prioritize and fix these vulnerabilities to make their organization more secure.
- CVE details may be used by security advisories to look for known attack signatures and detect specific vulnerability exploits.

### 2.3.5 CVE's structure

A CVE Numbering Authority (CNA) is in charge of assigning CVE identifiers [11]. There are approximately 100 CNAs, representing major IT manufacturers, security firms, and research organizations. CVEs may also be issued directly by MITRE.

CNAs are given block of CVEs which are held in reserve in case new vulnerabilities are found. Every year, tens of thousand of CVE IDs are released. Hundreds of CVEs may accumulate in a single complex product, such as an operating system.

CVE reports can originate from any source. A vendor, a researcher, or even an astute consumer may find a mistake and report it to the appropriate authorities. Knowledge about the flaw eventually finds its way to a CNA. The CNA assigns a CVE ID to the content, as well as a brief description and references. The CVE is then published on the CVE website in the format shown in figure 2.3.1 and 2.3.2.

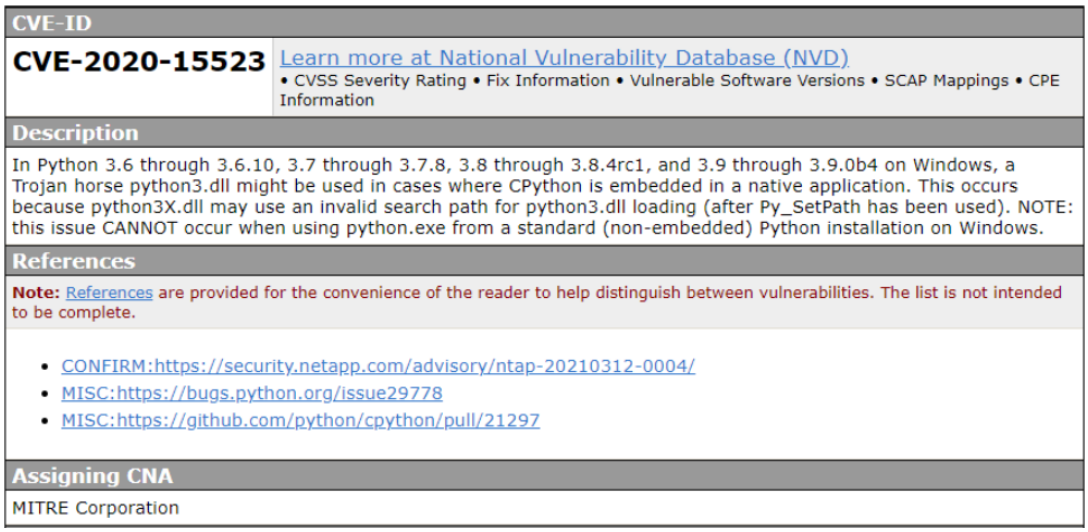


Figure 2.3.1: An example of the structure of a CVE (part 1).

A CVE entry contains the CVE ID, in the format "CVE-[Year]-[Number]" with year representing the year in which the flaw was reported and number, a sequential number assigned by the CNA. A brief overview of the security vulnerability or exposure, and references which can include links to vulnerability reports and advisories are also included.

Additional information on each CVE can be found on vendor websites as well as in the National Vulnerability Database (NVD) maintained by the National Institute

| Date Record Created | |
|---|---|
| **20200704** | Disclaimer: The record creation date may reflect when the CVE ID was allocated or reserved, and does not necessarily indicate when this vulnerability was discovered, shared with the affected vendor, publicly disclosed, or updated in CVE. |
| **Phase (Legacy)** | |
| Assigned (20200704) | |
| **Votes (Legacy)** | |
| | |
| **Comments (Legacy)** | |
| | |
| **Proposed (Legacy)** | |
| N/A | |
| This is a record on the CVE List, which provides common identifiers for publicly known cybersecurity vulnerabilities. | |
| **SEARCH CVE USING KEYWORDS:** Submit<br>You can also search by reference using the CVE Reference Maps. | |
| **For More Information:** CVE Request Web Form (select "Other" from dropdown) | |

Figure 2.3.2: An example of the structure of a CVE (part 2).

of Standards and Technology (NIST). The NVD offers CVSS Based Scores, patch information, and other critical data that information security teams often need when attempting to mitigate a vulnerability or determining its overall priority.

### 2.3.6 Common Vulnerability Scoring System

The Common Vulnerability Scoring System (CVSS) is a collection of open standards used to assign a number to a vulnerability in order to determine its severity. The NVD, CERT and others use CVSS scores to measure the impact of vulnerabilities [12]. There are three types of metrics in CVSS: Base, Temporal, and Environmental (shown in figure 2.3.3)


Figure 2.3.3: Metric Groups

The Base metrics generate a score between 0 and 10, which can then be tweaked by ranking the Temporal and Environmental metrics. For industries, organisations, and governments that need accurate and consistent vulnerability severity ratings,

CVSS is an excellent option. The severity of vulnerabilities found on one's systems can be calculated using CVSS, and it can also be used to prioritize vulnerability remediation activities. CVSS ratings for almost all known vulnerabilities are provided by the National Vulnerability Database (NVD).

For base score ranges, NVD offers qualitative severity rankings of "Low", "Medium", and "High".

There are also cases where not all the information about a vulnerability is available, making it difficult to create a CVSS score. This can happen when the vendor announces the vulnerability but refuses to provide further details. In such cases, NVD analysts grant CVSS scores based on the worst-case scenario. As a result, if a vendor does not provide any information about a vulnerability, NVD will assign a score of 10.0 to that vulnerability.

# Chapter 3

# State of the art

This chapter aims to present the existing research and solutions presented in several articles regarding the implementation of cyber range and the automation of cybersecurity problems.

Much research has been done on the automatic generation of cyber range and cybersecurity problems in Capture-the-Flag competitions.

Several methods exist to generate these problems and automating cyber ranges.

Both [13] and [14] propose a framework for the automatic generation of cybersecurity problems.

Indeed, in the article "Automating the Generation of Cyber Range Virtual Scenarios with VDSL" [13] a framework to automate the definition and deployment of arbitrary complex cyber range scenarios is proposed. In this paper, a scenario can be described as the working environment, including networks, hardware, software and their behaviour during the session. The objective of the framework was to overcome the main limitation of virtual scenarios: the relatively short lifetime.
In general, a cyber range should not be overused because it may become monotonous and quickly lose interest. As a result, a cyber range should have a system for rapidly creating new scenarios while ensuring that they include the desired features. In order to do this, the framework relies on a virtual scenario description language (VSDL) which will provides the constraints of the virtual scenario. These constraints are then processed through a satisfiability modulo theories (SMT) solver to be validated into a model. The model allocates values to constants and functions and automatically converts them into a virtual scenario.

On the other hand, in the paper "Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events"[14], the framework does not generate a virtual scenario based on desired features. The Security Scenario Generator

(SecGen) randomly generates vulnerable rich-scenario VMs. To do so, it uses pre-existing scenarios that cover a wide range of use-cases.
SecGen includes a modular architecture that allows it to dynamically produce difficulties by nesting modules, as well as a hint generation system that helps beginner security students move through complex challenges. [14]'s aims were to overcome the limitation of the existing static pre-configured hacking challenges, to provide an easy way for student to educate themselves and to find a better solution instead of manually configuring the scenarios.

Within the article "Automatic Problem Generation for Capture-the-Flag Competitions" [15], the authors explain the problematic of having fixed challenges in a capture-the-flag competition and provide a solution to it. In this paper, they focused on *Jeopardy-style* contests where each team were presented the exact same set of challenges, where each challenge had the same flag. This type of competition had one significant disadvantage which was that teams could share and copy flags. To resolve this problem, they implemented an automatic problem generation (APG) which creates problem instances to ensure that each team has a different flag.

The objective of the authors was to explore automatic problem generation in the context of capture-the-flag competitions, where issues like fairness and scalability are particularly critical.

Another interesting research paper was "An Offline Capture The Flag-Style Virtual Machine and an Assessment of its Value for Cybersecurity Education" [16] in which a framework is provided that enables academic cybersecurity courses to feature Jeopardy-style CTF challenges. This framework consists of an offline virtual machine under Linux with a web server, database server, and daemons that run purpose-built insecure protocols, as well as several user accounts and intricate, layered access control configurations.

Flags are generated when the VM is first started. Depending on where they are written (e.g. in the source code of a service that is then compiled), these flags will allow different types of exercises (basic encryption, access control, protocol analysis, web security and reverse-engineering).

In the article "Simulating Cyber-Attacks for Fun and Profit" [17], the authors introduce a new simulation platform called *Insight*. It was built with the goal of designing and simulating cyber-attacks against big arbitrary targets. It allows the simulation of vulnerabilities (including 0-days) and exploits, enabling an attacker to compromise machines and utilize them as pivoting stones in their attack.

The authors make a quick comparison between Insight and the previous methods

used to attract malicious network activities and to collect data. These previous methods were primarily honeypots and honey-nets which can be described as an information system resource whose value lies in unauthorized or illicit use of that resource[18]. They were therefore used to attract attackers and to obtain information on the different types of cyber-attacks.

The authors of articles [19, 20, 21] discussed the current and future trends of cyber ranges and test-beds and their usage for education, training and research. These papers' goals were:

- to study the concept of a cyber range, conduct a thorough literature assessment on unclassified cyber ranges and security test-beds
- to learn more about key components, particularly the tools used to design, build, install and run a cyber range platform
- and to evaluate the documented cyber ranges and test-beds.

In the same idea as cyber ranges, test-beds can be described as a controlled environment in which to carry out all kinds of actions in a secure and isolated manner without prejudicing the process being controlled.

In these articles, the focus is on the use of cyber ranges in the academic sector, but cyber ranges are not limited to this sector. They are also highly used by companies for learning and for security testing purposes. It also shows that the content of security challenges of cyber ranges varies and relies on the type of cybersecurity competition (web, cryptography, forensics, exploitation, steganography, DDoS,...). Other interesting results were the infrastructure platform that was mainly used for virtualization. Two types of virtualization technology were observed, conventional and cloud virtualization. Based on the results obtain in article [21], Openstack is the main tool used to deploy cloud environment and Virtualbox for the traditional technology.

Another question that was interesting for the rest of the thesis was the tool used to set up VMs. Infrastructure as code, IaC tools, particularly Ansible (with 40%), Vagrant, and Packer, dominate the technology used to deploy cyber ranges.

# Chapter 4

# Concept

In this chapter, the different concepts and tools used for the realization of this thesis are discussed and presented.

## 4.1 Vagrant

Vagrant is an open-source tool for building and managing virtual machine environments in a single workflow[22]. Mitchel Hashimoto began working on Vagrant as a personal side-project in January 2010. He then founded the HashiCorp corporation in November 2012 to fund the full-time development of Vagrant. The latter was originally only compatible with VirtualBox, but later versions included support for VMWare and KVM as well. Written in Ruby, it can also be used in projects written in other programming languages.

Vagrant works using a configuration file called *Vagrantfile* and uses what are called Boxes for the operating system of the VMs.

As shown below, the Vagranfile contains the different information about the VM we want to create from the name of the VM to the number of CPUs.

```
Vagrant.configure(2) do |config|
    config.vm.define 'nameVM' do |node|
        node.vm.box = 'generic/centos8'
        node.vm.provider 'virtualbox' do |vbNode|
            vbNode.name = 'VMCentos'
            vbNode.memory = 2048
            vbNode.cpus = 2
        end

        node.vm.provision 'shell', path: 'path/to/scripts.sh'
    end
```

```
end
```

The Vagrantfile shown here contains the configuration of a VM with CentOS 8 as operating system. It uses VirtualBox as a provider to create the VM and assigns it 2 CPUs with a memory of 2048 MB. It is also possible to add one or more script that will be executed at the first execution of the VM.

The example shown creates only a basic VM but it is possible to create several different VMs using one Vagrantfile and to modify individually the parameters of each VM.

Vagrant environment use the box package format that are used for operating systems. Anyone can use a box to create an identical working environment on any platform that Vagrant supports. The box uses in the example is a box available in the public Vagrant box catalog[1].

## 4.2 CVE-Search

CVE-Search is a tool that helps to search and process CVEs by importing CVE and CPE (Common Platform Enumeration) into a MongoDB (database) [23].

CVE-Search's main goal is to stop searching the public CVE databases directly and publicly. Local searches are typically easier, and help to limit the sensitive queries via the Internet.

### 4.2.1 CVE-Search database

MongoDB is a cross-platform document-oriented database software that is open source. MongoDB is a NoSQL database software that works with JSON-like documents[24].

By default, the MongoDB database generated by CVE-Search contains the following collections (with their source):

- CVEs (NVD NIST)
- CPE (NVD NIST)
- CWE (NVD NIST)
- CAPEC (NVD NIST)
- MITRE Reference Key/Maps
- VIA4CVE (aggregator of the known vendor vulnerabilities database to support the expansion of information with CVEs)

[1] `https://app.vagrantup.com/boxes/search`

The database is populated by downloading all current JSON files from the CVE feed as well as the CPE feed.

### 4.2.2 Usage

CVE-Search comes with a back-end for storing vulnerabilities and related data, a user-friendly web API interface for searching and handling vulnerabilities, and a set of tools for querying the device.

It can be used in console mode but also has a web interface (figure 4.2.1).

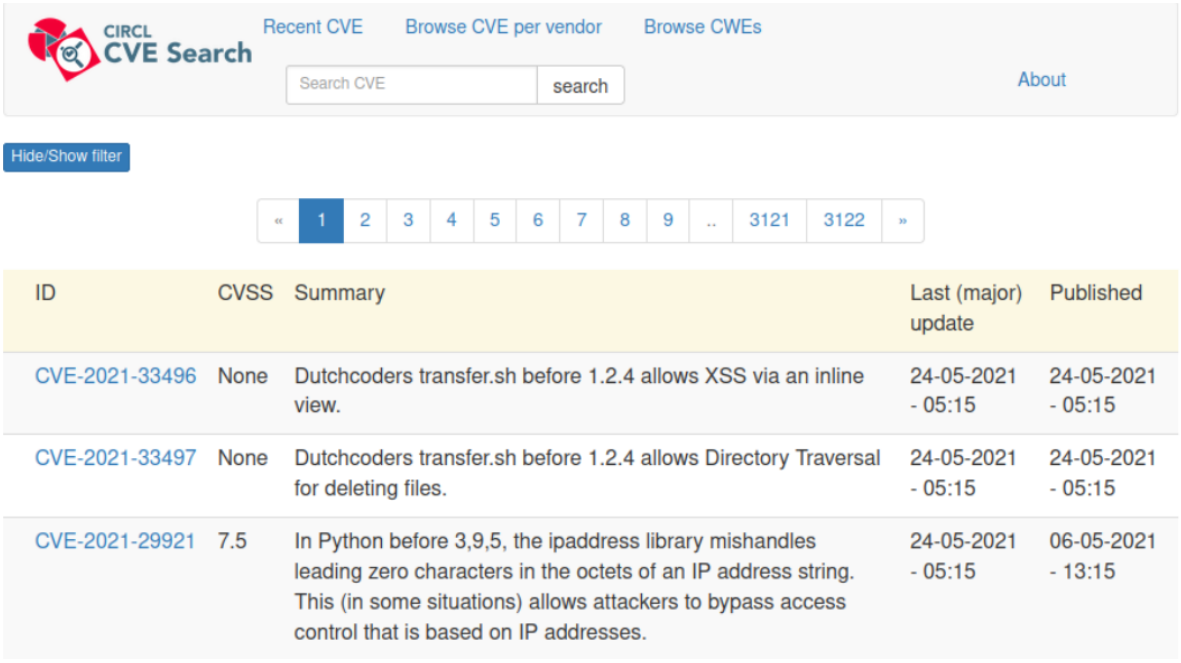


Figure 4.2.1: CVE-Search's web interface

The console mode offers a set of commands to search for CVE by providing:

- A name/version combination of the software

```
python ./bin/search.py -p cisco:12.4
```

- A particular word to search for in CVEs (including in title and description)

```
python ./bin/search.py -f python -n
```

- The ID name of a CVE

```
python ./bin/search.py -c CVE-2020-13873
```

It is also possible to define the format in which the result is displayed using the - *o* parameter (json, html, xml, csv or cveid).

The web interface proposes the same information available on the MITRE website[2] but also other information that will be useful in our implementation. As shown in figures 4.2.2 and 4.2.3, new data are added, such as a severity index, the CVSS (explains in chapter 2) and a complexity factor.

| | |
|---|---|
| ID | CVE-2020-13873 |
| Summary | A SQL Injection vulnerability in get_topic_info() in sys/CODOF/Forum/Topic.php in Codoforum before 4.9 allows remote attackers (pre-authentication) to bypass the admin page via a leaked password-reset token of the admin. (As an admin, an attacker can upload a PHP shell and execute remote code on the operating system.) |
| References | • https://twitter.com/sonarsource/status/1300818196090384384<br>• https://community.sonarsource.com/c/announce/stories/23<br>• http://codologic.com/forum/<br>• https://community.sonarsource.com/t/codoforum-4-8-7-critical-code-vulnerabilities-explained/28297 |
| Vulnerable Configurations | • cpe:2.3:a:codologic:codoforum:-:*:*:*:*:*:*:*<br>• cpe:2.3:a:codologic:codoforum:2.5.1:*:*:*:*:*:*:*<br>• cpe:2.3:a:codologic:codoforum:4.8.3:*:*:*:*:*:*:*<br>• cpe:2.3:a:codologic:codoforum:4.8.4:*:*:*:*:*:*:*<br>• cpe:2.3:a:codologic:codoforum:4.8.7:*:*:*:*:*:*:* |
| CVSS | **Base:** 10.0 (as of 20-05-2021 - 19:18)<br>**Impact:**<br>**Exploitability:** |
| CWE | CWE-89 |
| CAPEC | • Blind SQL Injection<br>• Object Relational Mapping Injection<br>• SQL Injection through SOAP Parameter Tampering<br>• SQL Injection<br>• Expanding Control over the Operating System from the Database |

Figure 4.2.2: Information display on the Web Interface of CVE-Search

[2] `https://cve.mitre.org`

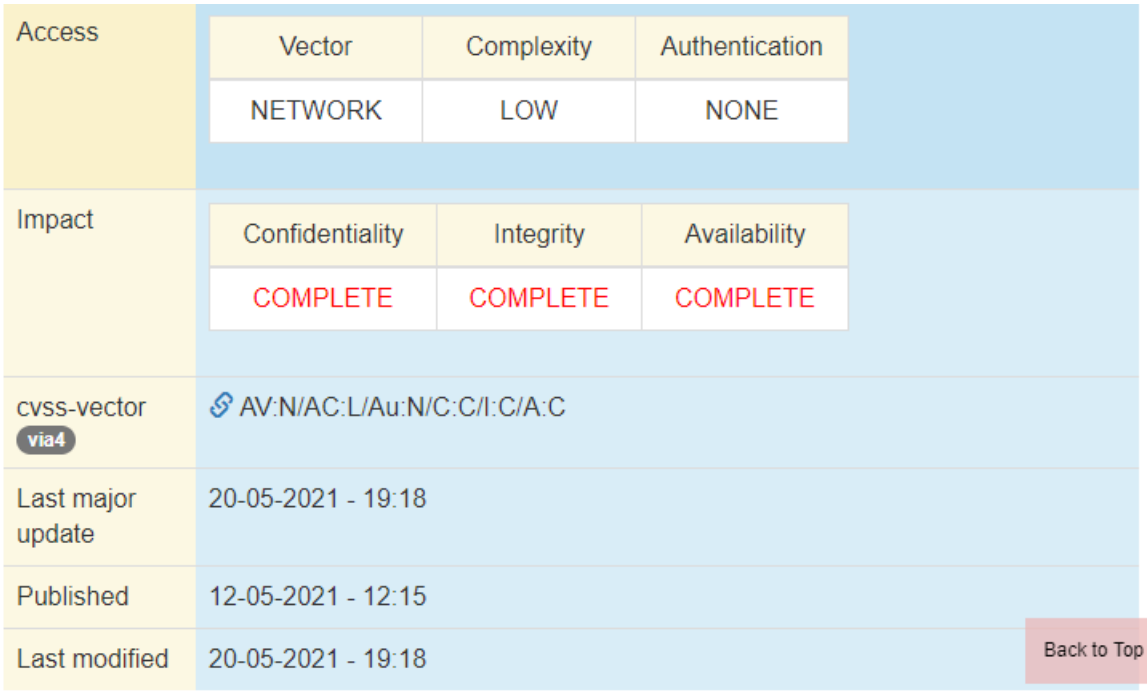


Figure 4.2.3: Information display on the Web Interface of CVE-Search

### 4.2.3 Complexity

As said, each CVE has multiple complexity indexes: the access complexity and the exploitability complexity (which is not shown on the web interface). These complexities can have three different values: Low, Medium and High.

Access complexity is determined by several variables. These variables are the rights privileged required by the attacker to access the vulnerability, the need for authentication, and the access to the vulnerability (network, adjacent network, local or physical).

The other complexity is the exploitability complexity which depends on the complexity of the attack, the privileges needed for this exploitation as well as the user interaction needed.

## 4.3 Debsecan

Debian Security Analyzer (Debsecan) is a tool that generates a list of vulnerabilities (CVEs) that affect a specific Debian installation. Debsecan is a program that runs on the target host and downloads vulnerability information from the Internet. It will notify interested parties via email when new vulnerabilities are found or

security updates are released.

The vulnerability database is maintained by the Debian testing security team[3].

Debsecan can be use with the following command:

```
python ./src/debsecan
```

This command analyzes the local system and outputs a list of CVEs that may be applicable in the analysed system. Indeed, this list of CVEs will be generated taking into account only the installed libraries and software but not their versions (fig. 4.3.1).

```
CVE-2017-14160 libvorbisfile3
CVE-2018-10392 libvorbisfile3
CVE-2018-10393 libvorbisfile3
CVE-2019-15795 python-apt-common
CVE-2019-15796 python-apt-common
CVE-2020-27351 python-apt-common
CVE-2019-10216 libgs9-common
CVE-2019-14811 libgs9-common
CVE-2019-14812 libgs9-common
CVE-2019-14813 libgs9-common
CVE-2019-14817 libgs9-common
```

Figure 4.3.1: Example of a debsecan execution

[3] https://security-team.debian.org/security_tracker.html

# Chapter 5

# Our approach

This chapter will describe the work done during this thesis and the means used to achieve the defined objective.

Firstly, this chapter provides a detailed description of the problem and the final objective to be reached.

Then, a description of the solution found is given by detailing the different steps of the resolution and the problems encountered.

Finally, this chapter ends with a list of the different algorithms implemented and/or used to meet our needs.

## 5.1 Description of the problem

As shown in Chapter 3, cyber ranges are quite common in the field of cybersecurity.

These cyber ranges allow the generation of many different environments that can perfectly represent company architectures. They are therefore very useful for learning and testing (web application, firewall rules, etc.).

However, the creation of these cyber ranges is not easy and can quickly become time consuming. Moreover, once created, these cyber ranges represent a fixed case and their modification can be difficult and also time consuming.

The main problem encountered here is the static aspect of cyber ranges. Once a solution has been found or tests have been performed, the cyber range is no longer useful.

As stated in some articles presented in Chapter 3, frameworks proposing some solutions to this problem have been implemented such as SecGen.

However, these frameworks use scenarios to generate cyber ranges. Although this solutions allows creating cyber ranges with different scenarios or creating new scenarios by associating several cases, the number of possibilities is limited and a

new cyber range must be generated at each modification.

The objective of this thesis is to propose an alternative to the existing framework using CVEs that allows the automation of a vulnerable system as well as its modification according to the needs.

## 5.2 Solution proposed

In order to achieve this goal, a tool has been implemented to retrieve the desired parameters and to automatically generate a vulnerable system with these vulnerabilities.

This tool also allows adapting and modifying at will an existing system in order to not have to generate a new one at each modification.

As said, the implemented tool allows the automation of a vulnerable system. In order to overcome the time consuming problem, this tool will allow the automatic generation of a vulnerable system from the conditions stated by the user. No other interaction with the user will be necessary once the parameters are entered.

The first approach was to implement a script that would directly interact with the provider, in our case VirtualBox.

This script retrieves the parameters of the virtual machine using the information provided via a CLI interface available to the user. This interface allows the user to enter information about the virtual machine such as the number of CPUs, the hostname, the allocated RAM or the number of vulnerabilities to integrate and the complexity of these vulnerabilities (fig 5.2.1).

```
What parameter do you want to change ?
          1 - Name of the VM (default: test)
          2 - OS (default: ubuntu)
          3 - Memory (default: 2048)
          4 - Hard disk size
          5 - Number of cpu (default: 2)
          6 - Add random vulnerability (default: None)
          7 - Exit
Choice : 
```

Figure 5.2.1: List of parameters of a VM

Once this information was retrieved, the script would create the VM and then it would connect to it via SSH to add the vulnerabilities stated by the user.

Unfortunately, the generation of the virtual machine in this way was not conclusive and creating multiple environments was not efficient.

Following this, another solution was found allowing the generation of one (or more) vulnerable(s) system(s). Instead of first creating the VM and then connecting to it via SSH, we used a tool called *Vagrant* to do all this at once.

As described in chapter 4, Vagrant is an open-source tool allowing to create and manage virtual machine environments in a single workflow.

It allows establishing beforehand the configuration of one or several virtual machine with the help of its configuration file, *Vagrantfile* (4.1). Vagrant also allows including scripts that will be executed at the creation of the system(s). It is also possible to create a sharing directory between the virtual machine and our machine, which is very useful for the rest of the implementation.

Regarding the static aspect of cyber range, the solution to this problem is to use the list of publicly known information security vulnerabilities and exposures (CVE).

These CVEs are used in order to select the vulnerabilities that will be integrated or added into the system. Indeed, the user can choose:

- the software for which he wants to have a vulnerability,
- the severity of the vulnerability,
- the CVSS score of the vulnerability,
- the complexity of the vulnerability,
- and the number of vulnerabilities desired for the selected software.

This can be seen in the following figure:

```
What parameter do you want to change ?
         1 - Name of the VM (default: test)
         2 - OS (default: ubuntu)
         3 - Memory (default: 2048)
         4 - Hard disk size
         5 - Number of cpu (default: 2)
         6 - Add random vulnerability (default: None)
         7 - Exit
Choice : 6
        1 - New software
        2 - Exit
Choice: 1
Name of the software: python
Minimum score for the vulnerability (0-10): 6
Vulnerability complexity (1 - Low, 2 - Medium, 3 - High): 3
Number of wanted vulnerability for this software ? 2
```

Figure 5.2.2: Addition of a vulnerability

To modify an existing virtual machine, the procedure is the same as for the selection of vulnerabilities shows above except that these vulnerabilities will need to be added by connecting to the existing machine.

```
Welcome !
What do you want to do ?
         1 - Create one (or more) Virtual Machine(s)
         2 - Launch the Virtual Machine(s)
         3 - List the vulnerabilities of a Virtual Machine
         4 - Modify existing Virtual Machine
         5 - Exit
Choice : 4

2 VM(s) was(were) found :
        1 - testCentos
        2 - test(vm must be launched before)
        3 - Exit
Choice: 1
```

Figure 5.2.3: Modifying the vulnerabilities of an existing VM

This is facilitated by Vagrant which already provides a method of connecting over SSH using the following command:

```
vagrant ssh [name|id]
```

### 5.2.1 Algorithms

This section describes the different algorithms implemented and used to fulfil the above requirements. The algorithms are represented in the form of pseudo-codes.

#### Creation of a virtual machine

Before we can "install" the vulnerabilities in the virtual machine, we must first create the virtual machine.

To create the first virtual machine, we use one of the tools outlined in Chapter 4, Vagrant. It allows creating, from a configuration file (Vagrantfile), one or more virtual machines. The generation of the configuration file of the first VM is done in the following way:

- The data is retrieved via the console interface, as shown in figure 5.2.1.
- The configuration file is created using the function *create_vagrantfile* which consists in creating the file and writing the initial structure of the file.

Once the file is created, the function *write_in_vagrantfile* intervenes which consists in recovering the contents of the configuration file and adding the information concerning the machine to be created.

This is how it works when you want to create the first machine but if the user wants to create different machines at the same time, then only the function *write_in_vagrantfile* will intervene for the next iterations.

Here is the pseudocode of the process used to create the Vagrantfile:

```
def create_vm:
    while user does not exit:
        user choose which parameter he wants to add/change

    if user has finish:
        launch create_vagrantfile
        launch write_in_vagrantfile
            (with a dictionary containing the parameter)

def create_vagrantfile:
    if Vagrantfile does not exist:
        create the file and opens it
        write the line for allowing a share directory between each vm

def write_in_vagrantfile:
    open the Vagrantfile
    extract the content

    if parameter_list is not empty:
        write in the file the parameters
            retrieved via the console (create_vm)
    add the scripts that will be launched at the start of the vm
    close the file
```

### Adding of vulnerabilities

Now that the creation of the virtual machine has been explained as well as the algorithms used, we can address the main topic of this thesis: the integration of vulnerabilities.

When the user wants to integrate vulnerabilities to a machine, he can decide for which software he wants to search vulnerabilities as well as the number of vulnerabilities, the complexity and the minimum score of CVEs. As quickly explained in the introduction, we use the CVE list to select which vulnerabilities will constitute the vulnerable system. Indeed, as shown in the following pseudo-code, we retrieve the information entered by the user and we use the CVE-Search tool to find the vulnerabilities that we will integrate.

```
def search_vulnerability:
    while user has not finish:
        user choose for which software he wants vulnerabilities

    for each software:
        execute CVE-Search tool to look for CVEs
        compute complexity
        compare complexity & score with requirements

        find the optimal version to install
            (with the required number of vulnerabilities)
        store optimal version in dictionary

    creation of the script with the command line for the installation
        (depending of the os)

    if vm does not exist:
        write_in_vagrantfile(installation_script)

    return script
```

For each software, we perform a search by name in the database generated by CVE-Search. The obtained CVEs are stored in a temporary file under a JSON format. Once all the CVEs are obtained for a software, the content of the file is extracted and each CVE is analyzed. Since the CVEs have been stored in JSON format, the analysis of the CVEs is done through their keys. We first calculate the complexity of the CVE to check if it matches the user's request. The complexity is calculated by retrieving the access and exploitability complexities and following the table below:

| Exploitability \ Access | LOW | MEDIUM | HIGH |
|---|---|---|---|
| LOW | LOW | MEDIUM | HIGH |
| MEDIUM | MEDIUM | MEDIUM | HIGH |
| HIGH | HIGH | HIGH | HIGH |

Table 5.1: Determination of the complexity of a CVE

In addition to the complexity, we also check if the CVSS score is above the desired threshold. Once all the CVEs have been analyzed, the optimal version is selected. This version is determined using the list of vulnerable "products" (our

software and others related) affected by the vulnerability. This process is performed for each software selected by the user. Once the number of predefined vulnerabilities is reached for all the software, a script allowing the installation of these softwares (with their optimal version) is created and added to the configuration file.

If the vulnerabilities are decided at the creation of the VM, it will simply be added as a script as explain in the pseudo-code of the creation of a virtual machine. If the vulnerabilities are added/modified after the VM has been created, then the following algorithm applies:

```
def modify_vm:

    recuperation of the vulnerable software already installed
    uninstallation_list = parse_vagrantfile()

    creation of the script with the command line
        for the reset (depending of the os)

    installation_script = search_vulnerability()

    connection to the virtual machine through SSH

    execution of the scripts
```

When modifying an existing VM, not only must new vulnerable software be installed but any vulnerable software that is already present must be uninstalled first.

The uninstallation of vulnerable software on the VM is easily achieved because the script that installed the specific versions of these softwares was always accessible from the configuration file. Obtaining these softwares (and their respective versions) is done in the following way:

```
def parse_vagrantfile:
    open Vagrantfile

    obtain value for corresponding VM
    find vulnerability installation script
    open script file
    extract list of software and version

    return list of software and version
```

### Listing of vulnerabilities

Another feature of the tool is the possibility of obtaining the list of vulnerabilities of a machine.

To implement this feature, two different methods are used depending on the virtual machine's OS.

If the machine is running on Ubuntu, a connection is established with the virtual machine and the Debsecan tool is executed. Debsecan allows to retrieve a list of CVEs based on the list of installed softwares and libraries (fig 4.3.1). Debsecan is directly executed on the concerned machine with the following command:

```
python3 /vagrant/debsecan/src/debsecan
```

The resulting list from the execution is written to a temporary file in the share folder between the host machine and the virtual machine (defined in the configuration file).

In addition to this file, another file is generated using the following command:

```
apt list --manual-installed | grep -F [\installed]
```

This last one allows to get the list of softwares and libraries installed manually (and not the one "integrated" with the OS) and to store it in a second file (fig 5.2.4).

```
accountsservice/bionic-updates,bionic-security,now 0.6.45-1ubuntu1.3 amd64 [installed]
acl/bionic,now 2.2.52-3build1 amd64 [installed]
acpid/bionic,now 1:2.0.28-1ubuntu1 amd64 [installed]
adduser/bionic,bionic,now 3.116ubuntu1 all [installed]
apparmor/bionic-updates,bionic-security,now 2.12-4ubuntu5.1 amd64 [installed]
apport/bionic-updates,bionic-updates,bionic-security,bionic-security,now 2.20.9-0ubuntu7.23 all [installed]
apport-symptoms/bionic,bionic,now 0.20 all [installed]
apt/bionic-updates,now 1.6.13 amd64 [installed]
apt-utils/bionic-updates,now 1.6.13 amd64 [installed]
at/bionic,now 3.1.20-3.1ubuntu2 amd64 [installed]
base-files/bionic-updates,now 10.1ubuntu2.10 amd64 [installed]
bash/bionic-updates,now 4.4.18-2ubuntu1.2 amd64 [installed]
bash-completion/bionic,bionic,now 1:2.8-1ubuntu1 all [installed]
bc/bionic,now 1.07.1-2 amd64 [installed]
bcache-tools/bionic-updates,now 1.0.8-2ubuntu0.18.04.1 amd64 [installed]
bsdmainutils/bionic,now 11.1.2ubuntu1 amd64 [installed]
bsdutils/bionic-updates,bionic-security,now 1:2.31.1-0.4ubuntu3.7 amd64 [installed]
```

Figure 5.2.4: List of some softwares extracted from Ubuntu system

Once these two files are obtained, we first extract the list of softwares with their corresponding versions. Then, we use the CVE-Search tool to check if the

listed CVEs are actually applicable in our machine. Indeed, Debsecan generates the list taking into account only the name of the software and libraries but not their version. The CVE-Search tool allows us to obtain information about the CVEs and to compare if the version of the concerned product for each CVE correspond to the version installed on the virtual machine.

```
def print_vulnerabilities:
    connect to the vm
    extract software_list & execute Debsecan (if OS is Ubuntu)

    if OS is Ubuntu:
        cve_list = parse_CVE_list(software_list)
    elif OS is CentOS:
        cve_list = parse_software_list(software_list)
    create new file
    for each cve in cve_list:
        print id, description, score, complexity
        add to file id, description, score, complexity

def parse_CVE_list:
    open file containing the CVEs
    extract each CVE

    open file containgin the list of software
    parse it and store in a dictionary the software and version

    cve_list
    for each CVE:
        execute CVE-Search (result in file)

        open result file
        compare product & version with corresponding software
        if match:
            store in cve_list

    return dictionary
```

The result of this analysis is then displayed in the console and written in a file. The information displayed is:

- the CVE identifier,
- the CVSS score,

- a description of the vulnerability,
- and the complexity.

In case the machine is under CentOS, Debsecan cannot be used. The method applied in this case is the following (as well as the print_vulnerabilities function explained above):

```
def parse_software_list:
    open file containing the list of software

    parse it and store in a dictionary the software and version

    cve_list
    for each software-version combination
        execute CVE-Search (result in temp file)

        open temp file
        for each CVE
            compare product & version
            if match
                store in cve_list

    return cve_list
```

The

```
dnf history userinstalled
```

command also allows to retrieve a list of softwares in a CentOS environment. This list is then analysed and the software/version combinations are recovered (fig 5.2.5).

Once the combinations are retrieved, the CVE-Search tool allows us to search for CVEs corresponding to one of the combinations. For each CVE found, the vulnerable products and versions are analysed and each match is displayed in the same way as for Ubuntu.

```
Packages installed by user
authselect-compat-1.2.1-2.el8.x86_64
autoconf-2.69-27.el8.noarch
automake-1.16.1-6.el8.noarch
bash-completion-1:2.7-5.el8.noarch
bind-utils-32:9.11.20-5.el8_3.1.x86_64
bzip2-1.0.6-26.el8.x86_64
chrony-3.5-1.el8.x86_64
cmake-3.11.4-7.el8.x86_64
cpp-8.3.1-5.1.el8.x86_64
dos2unix-7.4.0-3.el8.x86_64
elfutils-libelf-devel-0.180-1.el8.x86_64
epel-release-8-10.el8.noarch
gcc-8.3.1-5.1.el8.x86_64
gcc-c++-8.3.1-5.1.el8.x86_64
glibc-devel-2.28-127.el8.x86_64
glibc-headers-2.28-127.el8.x86_64
grub2-pc-1:2.02-90.el8_3.1.x86_64
grub2-tools-1:2.02-90.el8_3.1.x86_64
haveged-1.9.14-1.el8.x86_64
```

Figure 5.2.5: List of some softwares extracted from CentOS system

# Chapter 6

# Experimental results

This chapter describes how the tool was tested, the interesting results observed, the problems encountered during the different steps and the solution found to overcome them.

Firstly, here is short a timeline showing the 3 main steps of the implementation:

- Step 1 • Automation of a virtual machine
- Step 2 • Integration of vulnerabilities in new VM
- Step 3 • Modifying an existing VM

During the first step, several options were considered. The first idea was to generate the machine using CLI commands which would directly interact with VirtualBox. Unfortunately, during the tests, we could see that the generated machines did not have the parameters defined by the user and that this method was not the most optimal in terms of generation time.

Following these inconclusive results, a second method was tested. This method was based on the use of the Vagrant tool which allowed the building and maintenance of portable virtual software development environments. This tool allowed the automatic generation of several vulnerable systems with different parameters for each (OS, CPU, RAM, etc.). Indeed, this was confirmed during the tests which also allowed to detect a problem with the use of Vagrant. In order to generate machines with different OS, Vagrant uses Vagrant Boxes (4). These boxes are mainly created by developers or organizations in order to provide an environment for developers. The problem with these boxes is that they can have pre-installed software or tools and they usually don't have a GUI. Different boxes had to be tested to determine which ones were "empty" (without pre-installed software or tools) and scripts were written for each OS allowing the installation of a GUI at the generation of the VMs.

Step 2 was the step in which several tools and methods were tested to determine which were more interesting. Since the use of CVEs in the integration of vulnerabilities was already planned initially, the question that arose was, *"How to select these vulnerabilities ?"* The first method considered was to use Mitre's CVE list available online to retrieve vulnerabilities that met the user's criteria. Following some unsuccessful tests, we observed the following downsides:

- Requires a good and constant Internet connection (not always the case for students)
- One access was needed per vulnerability (many vulnerabilities → many connections)
- Time consuming (online access takes longer than local access)

Based on these observations, a local solution seemed more interesting and efficient. During the search for a solution, a tool offering this local solution was found, CVE-Search. As explained in chapter 4, CVE-Search creates a local database containing CVEs extracted from several databases. This solution allows to have continuous access with the database and reduces the search time for each vulnerability. As explained in chapter 5, the integration of these vulnerabilities is done with the help of scripts that are executed at the creation of the machine.

Regarding the tests carried out to determine if the correct version of the software had been installed, two tools were tested: **Vulmap** and **Debsecan**.

Vulmap is an online local vulnerability scanner that was executed on the virtual machine (scan the local system and look for vulnerability on an online API) and allowed to obtain a list of vulnerabilities by providing a file containing the information on the CVEs. Although this tool produced good results, it was unfortunately limited in the number of uses per month and the analysis time was long as it searches on online APIs. Vulmap was therefore not used.

The second tool tested unfortunately only worked on Ubuntu machines and as explained in chapter 5, it only provided the list of CVEs based on the name of the software but not its version. The tests were therefore carried out following the method defined in the previous chapter, depending on the OS of the machine.

When analysing the results of the scans, we could observe some interesting but also problematic results.

During the tests, we tried to get a VM with 2 python-related vulnerabilities with a Low complexity and a minimum score of 5. As shown in the figure 6.0.1, we

```
Software: python 3.5.0
--------------------

        CVE: CVE-2016-2183
        Score: 5.0
        Complexity: LOW
        Summary: The DES and Triple DES ciphers, as used in the TLS, SSH, and IP
Sec protocols and other protocols and products, have a birthday bound of approxi
mately four billion blocks, which makes it easier for remote attackers to obtain
 cleartext data via a birthday attack against a long-duration encrypted session,
 as demonstrated by an HTTPS session using Triple DES in CBC mode, aka a "Sweet3
2" attack.
        --------------------

        CVE: CVE-2021-23336
        Score: 4.0
        Complexity: HIGH
        Summary: The package python/cpython from 0 and before 3.6.13, from 3.7.0
 and before 3.7.10, from 3.8.0 and before 3.8.8, from 3.9.0 and before 3.9.2 are
 vulnerable to Web Cache Poisoning via urllib.parse.parse_qsl and urllib.parse.p
arse_qs by using a vector called parameter cloaking. When the attacker can separ
ate query parameters using a semicolon (;), they can cause a difference in the i
nterpretation of the request between the proxy (running with default configurati
on) and the server. This can result in malicious requests being cached as comple
tely safe ones, as the proxy would usually not see the semicolon as a separator,
 and therefore would not include it in a cache key of an unkeyed parameter.
        --------------------

        CVE: CVE-2019-20907
        Score: 5.0
        Complexity: LOW
        Summary: In Lib/tarfile.py in Python through 3.8.3, an attacker is able
to craft a TAR archive leading to an infinite loop when opened by tarfile.open,
because _proc_pax lacks header validation.
        --------------------
```

Figure 6.0.1: Vulnerabilities found for python 3.5.0

can observe that the VM does include the requested vulnerabilities but also has a High complexity vulnerability with a score of 4.

Although the latter does not respect the criteria imposed by the user, it is still present in the VM. This can be explained by the fact that even if the desired vulnerabilities had to be of Low complexity, the optimal version of python found (3.5.0) for this request also included a vulnerability of High complexity.

Regarding the last major part of the implementation, very few problems occurred or interesting results observed (different from the one stated above).

Indeed, the particularity of this part was based on the connection to an existing VM and on the reset of the latter (uninstallation of vulnerable software). The connection to the VM was facilitated using the method provided by Vagrant which allowed to connect to a particular machine by providing the name of the latter. As for the installation and the search for vulnerabilities or the tests performed, the method remained the same as before.

As explained in the previous chapter, resetting the VM was easily done because all the information needed were still available (name and version of vulnerable software).

# Chapter 7

# Future work

Throughout the thesis, the tests were carried out on a laptop environment in order to obtain results reflecting the situations of the majority of students which only possess a laptop. Of course, using this tool on a more powerful machine would give better results in terms of execution time or even access to existing machines. It would also allow the use of the tool to be extended and new features to be added such as the one described here.

The current version of the tool fulfils the original purpose, but there is always room for improvement.

An interesting point would be to make it accessible to a wider range of people. Indeed, the tool currently offers only 2 operating systems for virtual machines: Ubuntu and CentOS.

In the professional environment, most machines run on Windows 10, so it might be interesting to include this operating system in the list to increase the number of possible cases. This could be done by adding a Vagrant box of Windows 10. The difficult part would certainly be about the installation of vulnerable software and how to do it via the Windows terminal.

Another solution that could greatly facilitate the use and access to the tool would be the implementation of a graphical interface (web or local) in addition to the command-line interface. Indeed, the latter can limit the functions that can be implemented. For example, the display of elements can quickly become saturated with information and confuse the user when he wants to obtain the list of vulnerabilities of a system.

This graphical interface would therefore allow more information to be displayed than the console version and in a more orderly fashion. For example, instead of choosing each parameter of a new VM individually, the user could do it in one go with a few clicks.

Another improvement also related to the graphical interface would be to allow a more permissive selection of CVEs. Currently, CVEs are selected randomly based on the parameters entered by the user and by choosing the optimal version that satisfies the user's criteria. With this improvement, the user will be able to directly select from a list of CVEs the one he wants to add and will also be able to filter the CVEs more easily according to his preferences (score, complexity, date, etc.).

The last improvement idea would be something more complex but very useful and interesting to implement. This would be to have an automatic upgrade of the environment once a set of vulnerabilities have been exploited or explored.

This could be done by placing flags at certain vulnerable locations and once a certain number (defined by the user at creation) of flags is reached, the tool would automatically update the machine and integrate a new set of vulnerabilities by removing the previous ones. In this idea, the flags are not to be retrieved, like in CTF challenges, but they would represent a location, a command or a solution that would be found. The user would then be warned by a pop-up or a message displayed on the tool.

# Chapter 8

# Conclusion

The aim of this master thesis was to provide students with a tool to automatically generate a vulnerable and modifiable learning environment in the field of cybersecurity.

As mentioned in Chapter 1, there are many resources available for learning in the field of cybersecurity. These include Capture the Flag competitions, pre-configured virtual machines, test-beds and cyber ranges, which is the resource used for the rest of this master thesis.

Before devoting ourselves to the implementation of the tool allowing us to fulfil the initial objective, a *state of the art* was first carried out as can be seen in Chapter 3.

In this chapter, many works were presented such as frameworks created to generate real-case scenarios. These frameworks worked by using defined scenarios and by automatically creating the environment according to the desired scenario. Other articles were also interesting as they covered the analysis part of these solutions (CTF, cyber ranges, test-beds, etc.)

Based on this chapter, several problems were often mentioned such as the very high time consuming cost of producing the environments when it is done manually or required manual intervention, the limitation in the number of different environments that can be created (depending on the number of scenarios available) or the static aspect of the environment, once the objective is reached or the solution is found, the virtual environment becomes obsolete.

Something that surprised me when looking at the state of the art is that there are very few, if any, scientific papers that use the method visited in this thesis, CVEs, in the generation of a vulnerable environment. CVEs are often used to determine the vulnerabilities of a system or to search for information about a particular vulnerability but rarely to create a vulnerable environment for learning purposes.

In order to automatically create a vulnerable and dynamic environment, a tool

has been designed. Several tools were used and defined in chapter 4 to search and integrate vulnerabilities within a system (Vagrant, CVE-Search and Debsecan). The solution developed and the algorithms implemented to realize this tool were explained in chapter 5 and the interesting results following the tests were reviewed in chapter 6. The implemented tool has allowed to meet the stated needs and to generate automatically a vulnerable environment that can be easily modified when desired.

As explained in chapter 7, *future work*, the next steps are the following. The first step would be to develop a graphical user interface to allow a simplified and optimal use. The next improvements would allow for a more advanced selection of the cyber range parameters (displayed list of CVEs for selection, display of the different VMs and their parameters). The last improvement idea explained in the chapter, would be something more complex to implement and would be to have an automatic upgrade of the environment once a goal is reached.

# Bibliography


[1] National Initiative for Cybersecurity Education (NICE) Cyber Range Project Team. The cyber range: A guide.

[2] Nist cve - glossary. https://csrc.nist.gov/glossary/term/CVE. Accessed: 2021-05-23.

[3] What is a cybersecurity capture the flag? https://startacybercareer.com/what-is-a-cybersecurity-capture-the-flag/. Accessed: 2021-06-05.

[4] What is capture the flag? https://ctftime.org/ctf-wtf/. Accessed: 2021-06-05.

[5] What is a cyber range? https://www.techopedia.com/definition/28613/cyber-range.

[6] What is a cyber range? https://cybersecurityguide.org/resources/cyber-ranges/.

[7] About Ecso. Understanding cyber ranges: From hype to reality. 2020.

[8] Common vulnerabilities and exposures. https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures. Accessed: 2020-12-20.

[9] What is cve? common vulnerabilities and exposures explained. https://www.upguard.com/blog/cve.

[10] Cve numbering authority (cna) rules. https://cve.mitre.org/cve/cna/rules.html\#Section_4_1_qualifications. Accessed: 2021-05-07.

[11] What is cve, its definition and purpose ? https://www.csoonline.com/article/3204884/what-is-cve-its-definition-and-purpose.html. Accessed: 2021-05-07.

[12] Nist vulnerability metrics. https://nvd.nist.gov/vuln-metrics/cvss. Accessed: 2021-05-23.

[13] Gabriele Costa, Enrico Russo, and Alessandro Armando. Automating the generation of cyber range virtual scenarios with vsdl, 2020.

[14] Z. Cliffe Schreuders, Thomas Shaw, Mohammad Shan-A-Khuda, Gajendra Ravichandran, Jason Keighley, and Mihai Ordean. Security scenario generator (secgen): A framework for generating randomly vulnerable rich-scenario vms for learning computer security and hosting CTF events. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, Vancouver, BC, August 2017. USENIX Association.

[15] Jonathan Burket, Peter Chapman, Tim Becker, Christopher Ganas, and David Brumley. Automatic problem generation for capture-the-flag competitions. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*, Washington, D.C., August 2015. USENIX Association.

[16] Tom Chothia and Chris Novakovic. An offline capture the flag-style virtual machine and an assessment of its value for cybersecurity education. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*, Washington, D.C., August 2015. USENIX Association.

[17] Ariel Futoransky, Fernando Miranda, Jose Orlicki, and Carlos Sarraute. Simulating cyber-attacks for fun and profit, 2010.

[18] Honeynet Project. *Know Your Enemy: Learning about Security Threats*. Addison-Wesley, 2004.

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

[20] Elochukwu Ukwandu, Mohamed Ben Farah, Hanan Hindy, David Brosset, Dimitrios Kavallieros, Robert Atkinson, Christos Tachtatzis, Miroslav Bures, I. Andonovic, and Xavier Bellekens. A review of cyber-ranges and test-beds: Current and future trends. *Sensors*, 20:7148, 12 2020.

[21] Nestoras Chouliaras, George Kittes, Ioanna Kantzavelou, Leandros Maglaras, Grammati Pantziou, and Mohamed Amine Ferrag. Cyber ranges and testbeds for education, training, and research. *Applied sciences*, 11(4):1809–, 2021.

[22] Introduction to vagrant. `https://www.vagrantup.com/intro`. Accessed: 2020-12-04.

[23] Cve-search. `https://cve-search.github.io/cve-search/`. Accessed: 2021-05-04.

[24] Mongodb. https://en.wikipedia.org/wiki/MongoDB. Accessed: 2021-05-04.

[25] Enrico Russo, Gabriele Costa, and Alessandro Armando. Building next generation cyber ranges with crack. *Computers security*, 95:101837–, 2020.

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

[27] Nestoras Chouliaras, George Kittes, Ioanna Kantzavelou, Leandros Maglaras, Grammati Pantziou, and Mohamed Amine Ferrag. Cyber ranges and testbeds for education, training, and research. *Applied Sciences*, 11(4), 2021.

[28] Elochukwu Ukwandu, Mohamed Amine Ben Farah, Hanan Hindy, David Brosset, Dimitris Kavallieros, Robert Atkinson, Christos Tachtatzis, Miroslav Bures, Ivan Andonovic, and Xavier Bellekens. A review of cyber-ranges and test-beds: Current and future trends. *Sensors*, 20(24), 2020.

[29] Seokcheol Lee, Seokjun Lee, Hyunguk Yoo, SungMoon Kwon, and Taeshik Shon. Design and implementation of cybersecurity testbed for industrial iot systems. *J. Supercomput.*, 74(9):4506–4520, 2018.

[30] Elochukwu Ukwandu, Mohamed Amine Ben Farah, Hanan Hindy, David Brosset, Dimitris Kavallieros, Robert C. Atkinson, Christos Tachtatzis, Miroslav Bures, Ivan Andonovic, and Xavier J. A. Bellekens. A review of cyber-ranges and test-beds: Current and future trends. *Sensors*, 20(24):7148, 2020.

[31] Debsecan. https://gitlab.com/fweimer/debsecan. Accessed: 2021-05-04.

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**