

Bachelor thesis

in the degree program
Informatik

A Framework to Simulate Computer Security Contests

Submitted by

Jannik Leander Hyun-Ho Novak

Matr. Nr.: 392210

on January 13, 2024

at the Technische Universität Berlin


University supervisor: Prof. Jean-Pierre Seifert

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Berlin, den 13. Januar 2024



Jannik Leander Hyun-Ho Novak

Abstract

Cybersecurity competitions, in which teams compete against each other with the goal of discovering and exploiting vulnerabilities in the opposing teams systems while also defending their own systems by patching the found vulnerabilities, have become a popular means for applying acquired knowledge in a supervised and legal setting. Being able to perform a realistic simulation of such a competition while receiving detailed performance feedback enables organizers of cybersecurity competitions to identify potential issues in their existing infrastructure ahead of the competition date. For this purpose, this work investigates the methods and procedures involved in simulating a cybersecurity competition and proposes a monitoring system for measuring the performance of every component in real-time.

To facilitate a realistic simulation of the participating teams, a probabilistic model was derived from a set of scoreboards obtained from past cybersecurity competitions. This model was integrated into a simulation program that performs actions for each team, which include launching an attack against or defending against an opposing team. The scoreboards obtained from competitions simulated in this manner showed a significant similarity with their original counterparts. Furthermore, the monitoring data indicated sufficiently well whether the components were performing appropriately throughout the simulated competition. This leads to the conclusion that the presented simulation and monitoring systems may be utilized in testing cybersecurity competition infrastructure ahead of the official date, increasing the chances of a successful competition.

Keywords: cybersecurity, simulation, ctf competition, attack-defense.

Zusammenfassung

Cybersecurity Wettbewerbe in denen Teams gegeneinander antreten mit dem Ziel Schwachstellen in den Systemen gegnerischer Teams zu finden und diese gleichzeitig in den eigenen Systemen zu beheben gewinnen zunehmend an Beliebtheit, da auf diese Weise das erworbene Wissen in einer legalen und beaufsichtigten Umgebung angewendet werden kann. Eine realistische Simulation eines Cybersecurity Wettbewerbs durchzuführen und zeitgleich detaillierte Informationen zu der Leistung einzelner Komponenten zu erhalten versetzt die Organisatoren solcher Wettbewerbe in die Lage, potentielle Probleme in ihrer existierenden Infrastruktur zu erkennen. Um dies zu ermöglichen, werden in dieser Arbeit die Methoden und Verfahren vorgestellt, die in einem Cybersecurity Wettbewerb involviert sind, und ein Monitoringsystem wird vorgestellt, welches in der Lage ist, die Leistungsstatistiken aller Komponenten in Echtzeit zu messen.

Um eine realistische Simulation der teilnehmenden Teams zu ermöglichen, wurde ein Modell basierend auf Wahrscheinlichkeiten entwickelt, welches als Ausgangspunkt Scoreboards von vergangenen Cybersecurity Wettbewerben verwendet. Dieses Modell wurde in ein Simulationsprogramm integriert, welches für jedes Team bestimmte Handlungen durchführt, beispielsweise das Angreifen des Systems eines gegnerischen Teams oder das Verteidigen des eigenen Systems. Die Scoreboards, die nach Ausgang dieser Simulationen erhalten wurden, wiesen signifikante Ähnlichkeiten mit den originalen Scoreboards auf. Weiterhin gab das Monitoringsystem ausreichende Hinweise über die Leistung einzelner Komponenten, um mögliche Schwachstellen identifizieren zu können. Abschließend führt dies zu der Konklusion, dass die vorgeschlagenen Methoden für das Simulieren und Überwachen eines Cybersecurity Wettbewerbs für das frühzeitige Testen der Infrastruktur solcher Wettbewerbe geeignet sind.

Keywords: cybersecurity, simulation, ctf competition, attack-defense.

Table of Contents

Acronyms	VI
1. Introduction	1
1.1. Motivation	2
1.2. Thesis Structure	2
2. Background	3
2.1. Understanding CTF Competitions	3
2.2. What Components Are Involved	5
2.2.1. The Engine Component	5
2.2.2. The Checker Component	5
2.2.3. The Vulnbox Component	6
3. Setup	7
3.1. The CTF Infrastructure	7
3.2. Provisioning a CTF Infrastructure	9
3.2.1. Generating Terraform Files	10
3.2.2. Invoking the Build Process	11
3.3. Configuring Specific Components	12
3.3.1. Configuration Methods	12
3.3.2. Configuring the Engine	13
3.3.3. Configuring the Checker	13
3.3.4. Configuring the Vulnbox	13
3.3.5. Invoking the Configuration Process	14
3.3.6. A More Efficient Configuration	15
3.4. Introducing the Orchestrator	15

4. Methods	17
4.1. An Overview of the Simulation Process	17
4.1.1. Determining Exploits and Patches	18
4.1.2. Orchestrating the Exploit Procedure	20
4.1.3. Flag Submissions	22
4.1.4. Monitoring the Simulation	22
4.1.5. Performance Considerations	25
4.2. A Probabilistic Model for CTF Simulations	26
4.2.1. Data Sources for Scoreboards	26
4.2.2. Scores as a Normal Distribution	26
4.2.3. Finding an Experience Level Distribution	29
4.2.4. Deriving Success Probabilities	29
4.2.5. Integration Into the Simulation Program	32
4.3. Different Modes of Simulation	33
4.3.1. Basic Stress Testing	33
4.3.2. Regular Stress Testing	34
4.3.3. Combined Stress Testing	35
5. Results	36
5.1. Scoreboard Comparison	36
5.2. Monitoring Information	38
6. Conclusions	40
6.1. Interpreting Simulation Data	40
6.2. Considerations for More Realistic Simulations	41
6.3. Interpreting Monitoring Data	41
6.4. Suggestions for Future Work	42
List of Figures	43
Listings	44
Bibliography	45
A. Appendix	49

Acronyms

API Application Programming Interface.

CPU Central Processing Unit.

CTF Capture The Flag.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

IO Input Output.

IP Internet Protocol.

JSON Javascript Object Notation.

RAM Random Access Memory.

SCP Secure Copy Protocol.

SSH Secure Shell Protocol.

TCP Transmission Control Protocol.

1. Introduction

Cybersecurity presents a field comprising many vast and rapidly changing areas of knowledge. To be able to remain on top of the newest developments in each area requires constantly learning about and adapting to these developments. Much of this learning, however, remains purely theoretical if a student does not aim to break the law or undergo the laborious task of building a home lab in which they can apply their acquired knowledge. It is for these reasons that CTF (Capture The Flag) challenges and competitions have quickly risen in popularity as a way of applying the acquired knowledge in a supervised and legal setting. The objective of such a challenge is generally to analyze a given piece of software for hidden vulnerabilities and then find a way to exploit these in order to acquire a hidden secret as a means of solving the challenge. [38, 37]

These challenges and competitions can come in many different forms, including *attack-defense*, which refers to a competition in which teams compete against each other with the aim of finding and exploiting vulnerabilities in the other team's systems while also patching the found vulnerabilities in their own systems. Such a competition requires meticulous planning to be able to provide and arrange all of the necessary infrastructure and advertise the competition to potential participants. [36] For this purpose, a group of developers and organizers collaborate to develop and test the necessary software, employing different methods available to them. However, current testing procedures are often limited to unit tests and basic stress testing environments, which are not sufficient for testing the software under conditions that it would be exposed to in the context of a realistic competition. [9] The goal of this work is to establish a framework for simulating the entire procedure of such a competition while also providing a monitoring system that can be used to test the CTF software and identify possible flaws occurring under realistic conditions. [33]

1.1. Motivation

Ahead of the CTF competition, so-called *vulnerable services* are developed so that each offers a unique application that will be strewn with a small number of vulnerabilities. [4] During the development of such a service, particular bugs can come in the form of race conditions or a lack of scaling to accommodate for the sort of load that the service will be under in realistic conditions, which can be quite difficult to find without at least a basic stress testing setup in place. However, existing stress testing configurations often do not offer an accurate emulation of the particular network traffic and resource utilization a service may encounter.

Having participated in the development of such services, we encountered various issues in our service implementation that were not readily apparent under normal conditions and involved certain race conditions that only appeared when the service was exposed to a significant load. These particular issues were quite difficult to identify with the existing test suite and have caused a lot of frustration, which prompted us to create a testing environment that would provide service developers with more in-depth feedback on the performance of their service implementations and further enable them to identify issues that may appear in their services under realistic conditions.

1.2. Thesis Structure

Beginning with chapter 2 the knowledge that serves as a prerequisite to understanding this work will be laid out. This includes information regarding the procedure of a CTF competition and the components that it involves. Following this, in chapter 3 the general infrastructure that has to be deployed and configured for a CTF competition will be explained, and the particular methods for setting up this infrastructure in a simulated context will be discussed. In chapter 4 the specific implementation details and procedures taking place in the simulation will be revealed. Furthermore, the methods that have been employed in creating a probabilistic model based on real CTF competitions will be presented. Lastly, in chapters 5 and 6, the results of this work will be discussed, and suggestions for future work will be laid out.

2. Background

This chapter aims to introduce the relevant terms that will be used in subsequent chapters and provide the prerequisite information to be able to follow along with this work. For this purpose, the overall concept and the procedures surrounding the organization and preparation of an attack-defense CTF competition will be described, and following this, the specific components comprising the infrastructure facilitating such a competition will be examined.

2.1. Understanding CTF Competitions

The usual procedure for a CTF competition involves an initial development phase where a number of services will be created for the competition. These services will usually attempt to provide a realistic-seeming application that includes a small number of hidden vulnerabilities, which the teams will try to find and take advantage of during the course of the competition. [2] There are several requirements that a service has to fulfill regarding its functionality and the mechanisms it employs to store a piece of information, also referred to as a *flag*, that can be obtained by exploiting a vulnerability. These requirements include, for example, that a service must be able to endure the load expected during a competition. [28] After this initial stage has been completed, the services are tested with the help of a standardized test suite, test runs where the service developers compete against each other, and a basic stress testing environment. [9]

Following the development stage, the upcoming CTF competition will be announced on a website where teams can sign up for it; typically, this will be on ctftime.org. This allows the organizers of the competition to recognize the demand for participation and plan out the competition in advance. Each competition has its own weight rating, and some competitions may contribute a greater number of points to a participating team's general rating than others. The *Enowars* competition, for instance, has a comparatively high weight rating, which is another reason that the services for it should be tested thoroughly. [41]

On the day of the competition, each team is assigned a virtual machine, from here on referred to as *vulnbox*, which is running every service developed for the competition. They will receive one hour to examine the source code or binary of each service until the competition officially begins. From then on, the teams will try to find and exploit as many vulnerabilities in the services on the other team's vulnboxes as they can, while also doing their best to patch the found vulnerabilities on their own vulnboxes. To facilitate this, the competition utilizes a round-based concept where a set number of rounds will be played, with each round lasting approximately one minute. With the start of a new round, new flags and noise are deposited into the services, with the flags being valid for the following ten rounds. Exploiting other teams and submitting the flags that have been acquired this way will increase the attack score of a team. The team's final score, however, also includes the points they lost from being exploited and the time their services have remained online throughout the competition. The team that has achieved the largest number of points wins the competition. [26]

2.2. What Components Are Involved

A CTF infrastructure typically comprises numerous components, each taking on a particular responsibility to facilitate the competition procedure. In the following, the three central components consisting of the *engine*, *checker*, and *vulnbox* will be examined, and their specific functionalities will be described.

2.2.1. The Engine Component

The engine functions, as the name suggests, as the central game engine responsible for maintaining information about every participating team, the services that are being played, keeping track of the scores for every team, and employing the methods provided by the so-called checkers in periodically depositing different kinds of data into the services. The data that the engine deposits consists, firstly, of the flags, which are stored in their respective locations, referred to as *flagstores*, and present the main target of acquisition for the teams. It also comprises so-called *noise* and *havoc*, which are primarily intended to test a service's functionality, but they also have the effect of generating misleading network traffic, which obscures the network traffic that is generated during flag deposits and makes it less evident where flags are located in the services. [27, 26]

2.2.2. The Checker Component

The checkers are a piece of software that is developed alongside each vulnerable service. Checkers are mainly used by the engine for interacting with services, utilizing a standardized set of methods. The primarily used methods include a method for depositing flags and a related method for retrieving flags, referred to as *putflag* and *getflag*. This goal is most often achieved by checkers by registering an account on the service and then uploading or otherwise storing the flag, which is a simple string in a specified format, at the particular flagstore location. The checkers further provide methods for storing and retrieving noise, which have been mentioned in the engine section and are referred to as *putnoise* and *getnoise*.

The last kinds of methods checker implementations provide are *havoc* and *exploit* methods, where the exploit methods will become more relevant in a later chapter. The havoc methods will be initiated by the engine alongside flag and noise deposits, and they generally emulate particular user behaviors and service interactions with the goal of generating unrelated traffic on the service. Typically, there is one exploit method defined for each vulnerability in the service, which can be employed to run an exploit against a particular flagstore and retrieve the previously deposited flag. To achieve this, the checker utilizes the method for exploiting a vulnerability that was intended by the service creator. [23, 24]

2.2.3. The Vulnbox Component

The so-called vulnboxes are a set of virtual machines that will be distributed among the participating teams. The vulnboxes typically contain the source code or binary for every service that is being played in the competition, while the services are also running on the vulnboxes. This gives teams the opportunity to analyze the source code of the services to find hidden vulnerabilities and to modify the source code to patch out any discovered vulnerabilities. The main goal of the teams is to gain points by running exploits for the vulnerabilities they have discovered against the services running on the other team's vulnboxes and to patch the vulnerabilities present on their own vulnbox. Furthermore, the checkers will directly interact with the services on each vulnbox when they are prompted to do so by the game engine. This happens, for instance, at the beginning of every round when the engine contacts the checkers to deposit new flags and noise into the services. [26]

3. Setup

In order to simulate an attack-defense CTF competition, first and foremost, a cloud infrastructure consisting of a game engine, a virtual machine for each team, and lastly, a checker node that the engine interacts with to deposit flags and noise into the services and to launch havoc methods has to be deployed.

Furthermore, it should be ensured that the simulation setup will be configurable to a granular degree and automate the entire workflow of creating and configuring a cloud infrastructure for the purpose of the simulation and destroying it after the simulation has finished running.

3.1. The CTF Infrastructure

The initial step towards simulating an attack-defense cybersecurity competition lies in building and configuring the CTF infrastructure. To accomplish this, a cloud provider that will end up hosting the infrastructure has to be chosen. There is a large selection of cloud providers offering IaaS (Infrastructure as a Service), which can be leveraged to build the infrastructure. [30]

To accommodate for this, the simulation program presents the users with various cloud providers to choose from depending on their specific requirements; this includes, for instance, *Hetzner Cloud* and *Microsoft Azure*. [12, 29] Throughout the course of this work, the infrastructure has been deployed on Hetzner Cloud since the fees were relatively low. Furthermore, the configuration of the underlying CTF infrastructure was inspired by the configuration available on the Enowars GitHub page, which was developed with Hetzner Cloud as the primary cloud provider. [35]

The next step lies in automating the process of deploying the infrastructure in the cloud environment, which is a task that can be accomplished with a tool called *Terraform*. [15] Using Terraform, the required virtual machines and associated network configuration parameters can be specified, and the deployment and destruction procedures of the simulation can be initiated reliably.

The infrastructure that serves as the basis for a CTF competition with N participating teams is depicted in Figure 3.1. It consists of a game engine that periodically instructs the checkers on the checker virtual machine to deposit flags and noise into the services running on each vulnbox and retrieve them to confirm they have been deposited successfully.

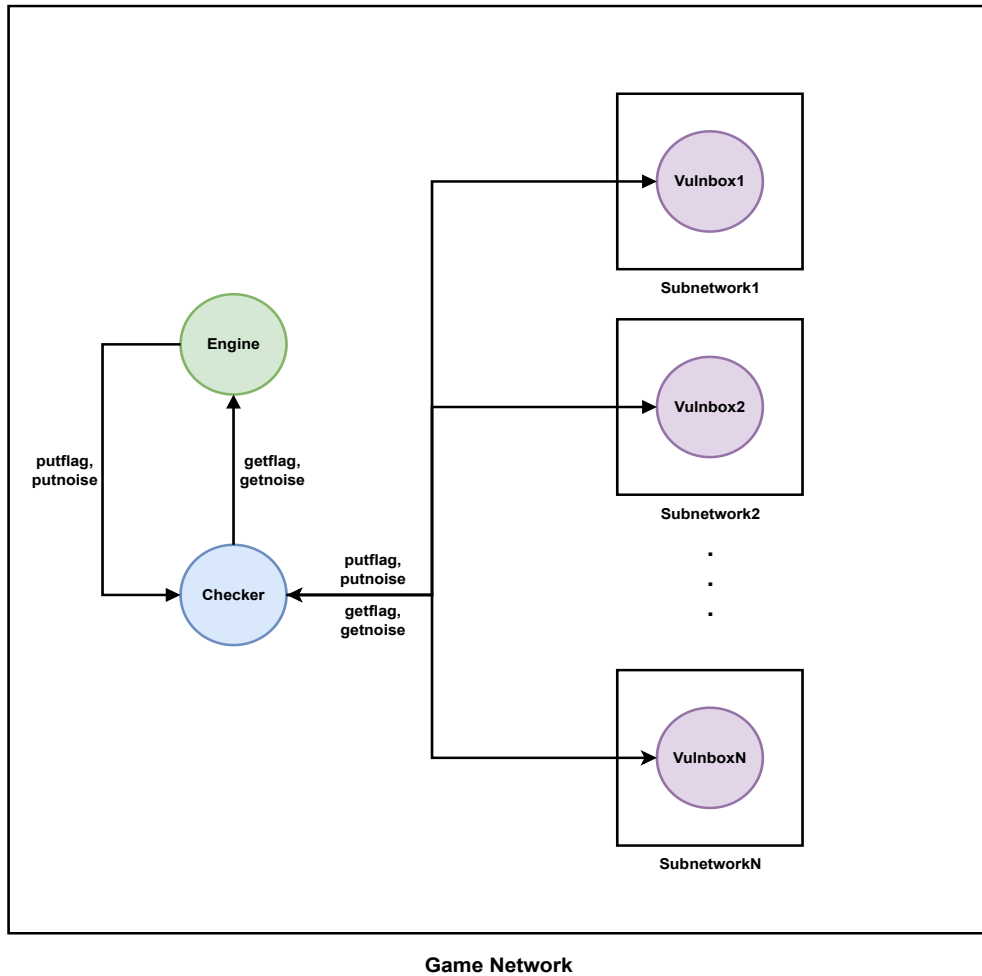


Figure 3.1.: CTF Infrastructure

3.2. Provisioning a CTF Infrastructure

The simulation program should further integrate a configuration section, which allows users to provide details regarding the infrastructure and the simulation at a high level of abstraction. This means, for instance, that users should be able to specify the number of simulated teams and the particular services that should be played.

To ensure a separation of concerns and the distinction between sensitive data and public data, the simulation program will split up the configuration section into two files in JSON (Javascript Object Notation) format that the user is responsible for supplying. [13] The first configuration file provides parameters such as the number of participating teams or the specific types of virtual machines to be deployed for each component in the simulation. The other file then contains restricted information, such as the location of the SSH (Secure Shell Protocol) private key to be used for authenticating connections to virtual machines. [6] The following two listings present excerpts from the configuration file containing publicly available configuration parameters and the configuration file storing sensitive parameters. How exactly the simulation program integrates these parameters will be explained in the following sections.

```
1 {  
2   [...]  
3   "settings": {  
4     "duration-in-minutes": 300,  
5     "teams": 3,  
6     "services": ["enowars7-service-CVExchange"],  
7     "checker-ports": [7331],  
8     "simulation-type": "realistic",  
9     "scoreboard-file": "/path/to/scoreboard.json"  
10  }  
11  [...]  
12 }
```

Listing 3.1: Configuration


```
1 {  
2   [...]  
3   "cloud-secrets": {  
4     "azure-service-principal": {  
5       "subscription-id": "[redacted]",  
6       "client-id": "[redacted]",  
7       "client-secret": "[redacted]",  
8       "tenant-id": "[redacted]"  
9     },  
10    "hetzner-api-token": "[redacted]"  
11  [...]  
12 }
```

Listing 3.2: Secrets

3.2.1. Generating Terraform Files

In the initial stage of the simulation procedure, the program has to start a configuration phase that includes converting the user-supplied configuration parameters into working Terraform provisioning scripts. The parameters should further be used to generate other files, including, for instance, a configuration file for the engine application. [25] These configuration files will be more closely examined in the next section. To facilitate the integration of user-defined parameters into functioning Terraform scripts and configuration files, cloud-provider-specific directories, which include template files for each Terraform script and configuration file, need to exist.

For this purpose, the simulation software includes a directory for each available cloud provider, containing the aforementioned template files. An exemplary procedure for converting one such template file according to the user-supplied parameters would begin by parsing the provided configuration files and then inserting the parameters into the correct placeholder sections inside the template files. Listing A.1 depicts an exemplary Terraform script template including placeholder variables, which can then be converted into a working provisioning script with the help of the Python code in Listing A.2. In this particular excerpt, the file will also be complemented with virtual machine specifications.

3.2.2. Invoking the Build Process

Following the initial configuration phase, the Terraform provisioning scripts have been created, and the next step the program should take consists of building the infrastructure with the help of a subprocess that launches the Terraform scripts. [11] More specifically, the program may execute a shell script generated from a template file that fulfills the responsibility of invoking the previously written Terraform scripts and creating an SSH configuration file in the user-specified location to enable authenticated connections to the created virtual machines.

Listing 3.3 depicts an excerpt from the template file that the main program converts into a functional shell script and invokes to create the CTF infrastructure. The template conversion procedure follows the same methods as described in the previous section. The specific steps that the script performs consist of executing the Terraform scripts via the *terraform apply* command, parsing the IP (Internet Protocol) addresses generated during the build stage via regular expressions, and lastly, generating an SSH configuration file in the user-defined location. [6, 15]

```
1  #!/usr/bin/env bash
2
3  terraform init
4  terraform apply -auto-approve
5  terraform output >./logs/ip_addresses.log
6
7  CHECKER_IP=$(grep -oP "checker\s*=\s*\K[^\s]+" ./logs/ip_addresses.
   log | sed 's/"//g')
8  ENGINE_IP=$(grep -oP "engine\s*=\s*\K[^\s]+" ./logs/ip_addresses.
   log | sed 's/"//g')
9
10 rm -f ${SSH_CONFIG}
11 echo -e "Host checker\nUser root\nHostName ${CHECKER_IP}\
   nIdentityFile ${SSH_PRIVATE_KEY_PATH}\nStrictHostKeyChecking no\
   n" >>${SSH_CONFIG}
12 echo -e "Host engine\nUser root\nHostName ${ENGINE_IP}\
   nIdentityFile ${SSH_PRIVATE_KEY_PATH}\nStrictHostKeyChecking no\
   n" >>${SSH_CONFIG}
```

Listing 3.3: Build Script Template

3.3. Configuring Specific Components

In addition to establishing a CTF infrastructure, the simulation program also needs to automate the process of configuring the individual virtual machines so they will be able to fulfill their role in the competition. For instance, the virtual machine that will serve as the engine has to install and start the game engine software, and the vulnboxes need to launch the services to be played in the competition. [25, 32] This section will describe in detail the procedures for configuring each component of the CTF infrastructure and how this configuration process can be automated.

3.3.1. Configuration Methods

To carry out the configuration of a virtual machine, shell script template files can be converted according to the same approach elaborated on in the previous sections. As an example, the configuration script for the vulnboxes may be written to include a *git clone* command for every service that will be played. [5] The generated configuration scripts can then be uploaded to the virtual machines using SCP (Secure Copy Protocol) and later executed via an SSH connection. [6] The responsibilities of these shell scripts include installing all of the necessary dependencies for the particular applications that need to run on each virtual machine and modifying certain system settings. One example of a system setting that should be configured is adding users to the *Docker* group, which later enables the execution of Docker commands from a remote host via an SSH connection. [20] The specific configuration details of each component will be elaborated on in the following sections.

Another tool that can be used to automate the configuration of virtual machines is called *Ansible*. This is a tool that focuses on configuring remote systems via so-called *playbooks* that define a sequence of steps to configure particular aspects of a virtual machine. [16] However, the configuration procedure has been implemented utilizing shell scripts because they were effortless to integrate into the main application through the use of subprocesses. A further benefit was the granular level of control over every parameter in a system that shell scripts allowed for.

3.3.2. Configuring the Engine

There are three applications that need to be launched and configured on the system operating as the game engine. The first one is the engine application, which is responsible for depositing flags and noise into the services and tracking the scores of every team. The second application is a flag submission endpoint, which the teams will connect to in order to submit the flags that they have acquired by exploiting other teams. Lastly, a separate frontend for displaying the competition scoreboard on a browser interface has to be launched. This application also provides so-called attack information to players over an HTTP (Hypertext Transfer Protocol) endpoint, which a checker optionally generates after flag deposits and contains additional details about the location of specific flagstores. [25, 34]

3.3.3. Configuring the Checker

The software that needs to run on the checker virtual machine includes the checker application for each of the services played in the competition. The checkers will primarily be used by the game engine to interact with the services on each vulnbox at the start of a round. They are typically included in a service repository and can therefore be downloaded alongside the service via a simple *git clone* command. [23]

3.3.4. Configuring the Vulnbox

The configuration of the vulnboxes follows a very similar series of steps as the checker configuration. The service repositories are downloaded onto the vulnboxes using the *git clone* command, and each service is started. [26] For the purpose of the simulation, the checkers need to run not only on the checker virtual machine, but each vulnbox also needs to run every checker alongside the related service. Because there are no real teams playing, the checkers will be utilized to emulate particular actions that a team operating a vulnbox may perform. An example of this could be the initiation of a havoc method on the checker running on vulnbox A, which has as its target vulnbox B. This would then result in the checker of vulnbox A interacting with the related service on vulnbox B. [8]

3.3.5. Invoking the Configuration Process

Having created the configuration files for every virtual machine, the simulation program has to copy the files onto the correct virtual machines and launch the configuration scripts. It may accomplish this by starting a new subprocess that launches a shell script responsible for uploading the configuration files onto the remote hosts via SCP and then executing the configuration scripts through an SSH connection. This shell script can be generated from a template file alongside the other configuration files, and it should serve as the main entrypoint for virtual machine configurations.

This was accomplished in the simulation program by adding a section for each virtual machine to the script that executes the commands for configuring the specific virtual machine. To enable the use of SCP and SSH in this process, the script utilizes the SSH configuration file that has been generated during the initial build phase of the infrastructure to authenticate the SSH connections. [6] Listing 3.4 depicts an excerpt from the template file used to generate this shell script. The completed shell script will further include configuration commands for every vulnbox and the correct file path to the SSH configuration file.

```
1 [...]
2 echo -e "\n\033[32m[+] Configuring checker ...\033[0m"
3 scp -F ${SSH_CONFIG} ./data/checker.sh checker:/root/checker.sh
4 scp -F ${SSH_CONFIG} ./config/services.txt checker:/root/services.
   txt
5 ssh -F ${SSH_CONFIG} checker "chmod +x checker.sh && ./checker.sh"
   >./logs/checker_config.log 2>&1 &
6
7 echo -e "\n\033[32m[+] Configuring engine ...\033[0m"
8 scp -F ${SSH_CONFIG} ./data/engine.sh engine:/root/engine.sh
9 scp -F ${SSH_CONFIG} ./config/ctf.json engine:/root/ctf.json
10 ssh -F ${SSH_CONFIG} engine "chmod +x engine.sh && ./engine.sh" |
   tee ./logs/engine_config.log 2>&1
11 [...]
```

Listing 3.4: Configuration Script Template

3.3.6. A More Efficient Configuration

An issue that became apparent when running the simulation program was that it required a long time to configure the individual virtual machines because the installation of dependencies and other time-consuming tasks had to be repeated for each of them. This was not a feasible option considering that users of the simulation program may have to wait an unreasonably long time for the infrastructure to be configured before the simulation would start running.

A solution to this issue presented itself in the form of *Packer*, which is a tool that can be used to generate file system images of virtual machines and facilitates reusing them in their deployment to be able to avoid the more time-consuming parts of the configuration process. For this reason, the configuration file includes a section to enable users to optionally provide a Packer-generated image for each component of the CTF infrastructure. Utilizing these images resulted in a considerable speedup in the overall time it took until the entire CTF infrastructure was available. [14, 31]

3.4. Introducing the Orchestrator

The last essential component needed to simulate a cybersecurity competition is called the *orchestrator*. This will represent an entity residing on a separate network that is running the simulation software. The orchestrator takes on the responsibility of instructing teams to run exploits against other teams, patch their services, and submit their flags. This is essentially mimicking real teams operating their vulnboxes and playing in the CTF competition. The orchestrator achieves this by communicating with the checkers running on each vulnbox via HTTP requests, and it also connects to components via SSH. The intricate details of these procedures will be elaborated on in chapter 4.

The orchestrator further takes on the job of monitoring the system statistics and Docker statistics on each virtual machine and persisting these statistics in a database for the purpose of reviewing the service and checker implementations. This allows service creators to receive in-depth feedback on potential performance issues with their service or checker implementations and enables them to discover issues that would usually be difficult to find with the help of a simple test suite. Catching these issues in a simulated version of the CTF competition is crucial to avoid unpleasant surprises and being required to patch services during the real competition.

Figure 3.2 reveals the functions of the orchestrator and how it integrates into the entire simulation infrastructure. It depicts the orchestrator's responsibilities of controlling the actions taken by each simulated team, storing system- and service-specific analytical data in a database, and lastly, integrating a separate frontend that will display this data in a graphical user interface, making it simple to monitor the simulation in real time.

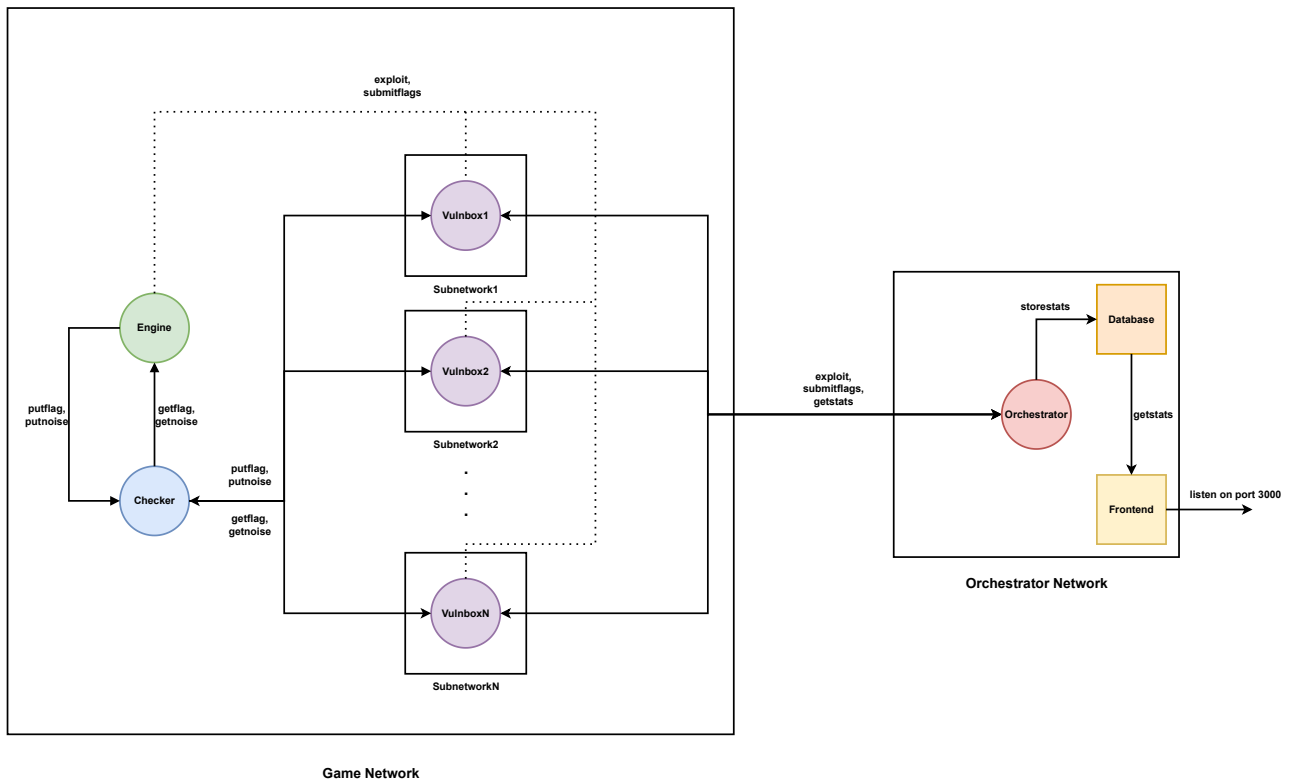


Figure 3.2.: CTF Simulation Infrastructure

4. Methods

At this stage, the groundwork for simulating a cybersecurity competition has been laid out. To build on this foundation, this chapter will examine more closely the intricate details of this simulation, including the sequence of actions taken by the orchestrator in every round and how they affect the remaining components, and mimic the dataflow of a real CTF competition.

This chapter will further elaborate on deriving parameters that will determine whether a team starts exploiting or patching a service at the start of any given round using a probabilistic model. Furthermore, the process of deriving a distribution of experience levels across the teams from a collection of scoreboards that have been obtained from previous CTF competitions will be described.

4.1. An Overview of the Simulation Process

To introduce the simulation procedure, this section will present a high-level overview of the sequence of steps that will be taken inside the orchestrator at the beginning of every round and how these actions will affect each component of the simulation. These steps are further illustrated in Figure 4.1.

As the first action in every round, the orchestrator parses the current round identifier from the scoreboard and acquires the latest attack information from the engine virtual machine. The orchestrator then utilizes these pieces of information to construct the so-called *checker task messages* for instructing the checkers on each vulnbox to run exploits against the services running on the other vulnboxes. [24] The intricate details of this procedure will be discussed in Section 4.1.2.

The next stage consists of performing a random test for each team to determine whether it starts exploiting or patching a new service that is chosen at random. The state of the specific services that a team has patched or is exploiting is saved in the orchestrator and updated with the random test in every round. Further details of this test will be elaborated on in the next section.

In the following, the orchestrator instructs each team to exploit the services that have been determined via the random test if and only if the targeted teams have not patched the specific service. Simultaneously, it establishes an SSH connection to each virtual machine to measure particular system and Docker statistics.

As the last action in every round, the orchestrator will submit the flags that were collected as a result of the exploit methods executed in the previous step. Section 4.1.3 will examine the exact steps that the orchestrator takes to achieve this.

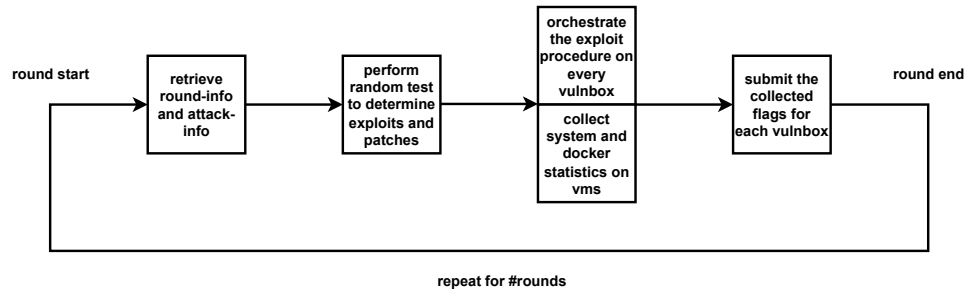


Figure 4.1.: Simulation Workflow

4.1.1. Determining Exploits and Patches

One of the first actions the orchestrator takes in every round is to determine, through a random test, whether a team will start exploiting or patching a new service.

To keep track of the states of the individual teams, the orchestrator saves a dictionary that stores for each team the exact services and flagstores it is exploiting and the services and flagstores it has patched. In a realistic simulation setting, each team will start with a state of false for both the state of exploiting and patching for every flagstore. This will gradually change throughout the simulation at the beginning of each round when the random test is performed.

To enable performing the random test, each team is assigned an experience level, which was derived through the analysis of scoreboards from past competitions. An experience level stores two pieces of information that are relevant to the simulation. The first is the team's likelihood of discovering a new exploit or patch, which is equivalent to the team passing the random test. The second is the prevalence of teams with specific expertise levels in the competition. The methods for deriving experience levels and their related probabilities will be elaborated on in Section 4.2.

For the random test, a random number between zero and one gets generated, and if the team's probability value exceeds the randomly generated value, they will pass the random test. Passing the random test will then cause a random flagstore to be selected either for exploitation or patching, and the game state will be adjusted to include the randomly selected flagstore in the team's exploiting or patching states. The following listings display the code that was used to perform the random test and the immediate steps following the random test.

```
1 def _random_test(self, team: Team) -> bool:
2     probability = team.experience.value[0]
3     random_value = random.random()
4     return random_value < probability
```

Listing 4.1: Random Test

```
1 async def _update_exploiting_and_patched(self) -> List[str]:
2     info_messages = []
3     if self.setup.config.settings.simulation_type == "realistic":
4         async with async_lock(self.locks["team"]):
5             for team_name, team in self.setup.teams.items():
6                 if self._random_test(team):
7                     random_choice = self._choose_random(team)
8                     variant, service, flagstore = random_choice
9                     info_message = self._update_team(
10                         team_name, variant, service, flagstore
11                     )
12                     info_messages.append(info_message)
13     return info_messages
```

Listing 4.2: Updating a Random Flagstore

4.1.2. Orchestrating the Exploit Procedure

Knowing the specific flagstores that each team is exploiting or has patched, the orchestrator will continue in the following phase by instructing each checker running on the vulnboxes to exploit the services that the teams are currently exploiting. This section seeks to describe in detail how the orchestrator achieves this and which components it involves in this process.

As pointed out in chapter 2, the engine utilizes the checker implementation for each service to interact with the services and to deposit flags and noise at the beginning of each round. [24] These checker applications further define exploit methods for each flagstore of the related service that are generally used for testing purposes to ensure that the flagstore can be accessed via the intended vulnerability. The execution of these methods can be initiated by any system able to contact the checker via the HTTP endpoint it exposes for this purpose. [23]

This means that the orchestrator may contact the checker running on a vulnbox through this HTTP endpoint to specify, via a so-called checker task message, which method the checker should execute and which system it should target with this method. [8] For this reason, every vulnbox needs to be running the checker for each service so the orchestrator can leverage the exploit methods of the checkers. For instance, the orchestrator is then able to instruct a checker running on vulnbox A to exploit a flagstore on vulnbox B, which effectively mimics the team operating vulnbox A running an exploit against this flagstore on vulnbox B.

The checker API (Application Programming Interface) is implemented in such a way that when the checker has finished running the exploit, it will return an HTTP response containing either the flag it has retrieved by successfully exploiting the targeted flagstore or, if the exploit was unsuccessful, a meaningful message that contains information about why the exploit failed. [8] This will become more relevant in the process of submitting the collected flags, which will be examined in the following section.

Figure 4.2 illustrates the vulnbox configuration, which entails running each checker implementation alongside its service. These checkers can then be contacted by the orchestrator to initiate an exploit task against another team specified in the checker task message.

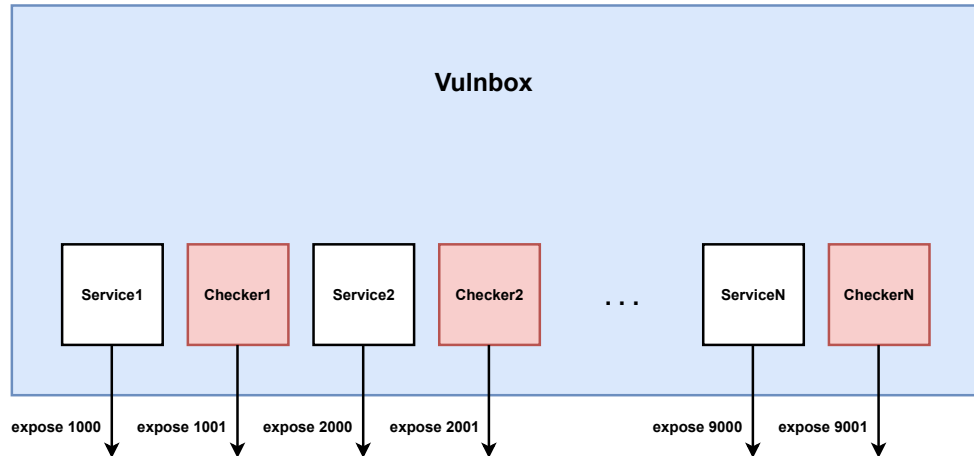


Figure 4.2.: Vulnbox Configuration

A typical exploit workflow is further depicted in Figure 4.3. The orchestrator generates a checker task message based on the source and target teams and fills in the relevant details, such as, for example, the IP address that the targeted team's vulnbox resides at. Sending the request to the checker then commences the exploit process against the service on the targeted vulnbox.

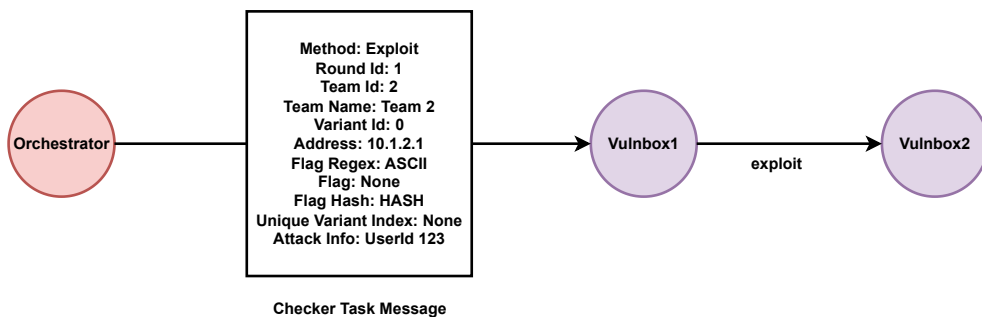


Figure 4.3.: Exploit Workflow

4.1.3. Flag Submissions

After the checkers have completed exploiting and the orchestrator has received responses to all of the sent-out checker task requests, the orchestrator has to gather the flags that are contained in the respective checker responses and submit them at the flag submission endpoint running on the engine virtual machine. [26]

It achieves this by establishing an SSH tunnel from the orchestrator via the specific vulnbox to the engine virtual machine. With this approach, the orchestrator is then able to create a new TCP (Transmission Control Protocol) channel between the particular vulnbox and the engine and send the collected flags over this connection. [6] The engine will register this as the team submitting its flags through their vulnbox and recalculate the team's score accordingly.

4.1.4. Monitoring the Simulation

To facilitate users of the simulation program receiving an overview of the simulation without being forced to have the technical knowledge required to connect to specific virtual machines via SSH to gather certain system statistics, the simulation program should offer a graphical interface that neatly aggregates every relevant statistic and displays them in an intuitive manner. For this, it would be important to display not only all of the configuration details of the simulation but also details regarding the infrastructure that was built for the simulation. Furthermore, a system for monitoring the performance of each component in real-time should be included.

The first implementation details for this system present themselves in the methods used for gathering the relevant system statistics. These could be collected by establishing an SSH connection to each virtual machine and executing a set of system commands. The data that will be accumulated with these commands consists of multiple parts. The first part includes details about the hardware comprising a virtual machine and statistics that are unlikely to change throughout the simulation, such as each virtual machine's public IP address. Furthermore, dynamically varying statistics, including RAM (Random Access Memory) usage, CPU (Central Processing Unit) usage, and network traffic, will be measured with these commands in every round. [42]

Another performance metric that should be measured concerns the Docker statistics on each virtual machine, which provide insight into the specific usage of resources for each service and checker implementation and allow the organizers as well as the service creators to recognize potential performance issues in specific services. [1] An example could be a service that utilizes a separate database and forgets to integrate indexing into their tables, which could potentially result in large CPU spikes in a realistic environment while being difficult to identify if the service is being tested with the help of simple unit tests.

For the measurement of all the aforementioned statistics, it is also crucial to choose an appropriate time of measurement. Measuring the statistics at the beginning of a round when no exploit traffic has been initiated between any virtual machines yet would lead to an inaccurate portrayal of the resource usage of each component. This is because the services and checkers are exposed to the greatest amount of load when they are being exploited or executing exploit methods. For this reason, the measurements will be performed in parallel with the execution of exploits in the middle of every round with the use of multithreading. Listing 4.3 displays an excerpt of the code that was used to collect such statistics.

```

1 with paramiko.SSHClient() as client:
2     [...]
3     _, stdout, _ = client.exec_command(
4         "free -m | grep Mem | awk '{print ($3/$2)*100}' &&"
5         + "free -m | grep Mem | awk '{print $2}' &&"
6         + "free -m | grep Mem | awk '{print $3}' &&"
7         + "sar 1 2 | grep 'Average' | sed 's/^.* //' | awk '{print
100 - $1}' &&"
8         + "nproc &&"
9         + "df -h / | awk 'NR == 2 {print $2}'"
10    )
11    system_stats = stdout.read().decode("utf-8")

```

Listing 4.3: Collecting System Statistics

The next feature that should be implemented for a monitoring system is an intuitive graphical user interface that offers an overview of every statistic associated with the simulation in a well-condensed manner. To this end, a simple web interface was developed using the React Framework called *Next.js*. [21] This web interface includes a dashboard for the simulation, which is split into three distinct sections, each providing relevant sets of information about the state of the simulation.

The first page displays an overview of the progress of the simulation, meaning how many rounds are still left to emulate, and an overview of each virtual machine in the simulation infrastructure, which includes both static information as well as graphical representations of the system statistics of each virtual machine over the course of the past rounds. The second page then displays service- and checker-specific statistics in a similar manner, where the graphical representations inform about the resource usage of each Docker container running on the vulnboxes, which includes the service and checker containers. The last page offers a condensed overview of the scores that each team has attained. Figure 4.4 displays the graphical user interface created for the simulation.

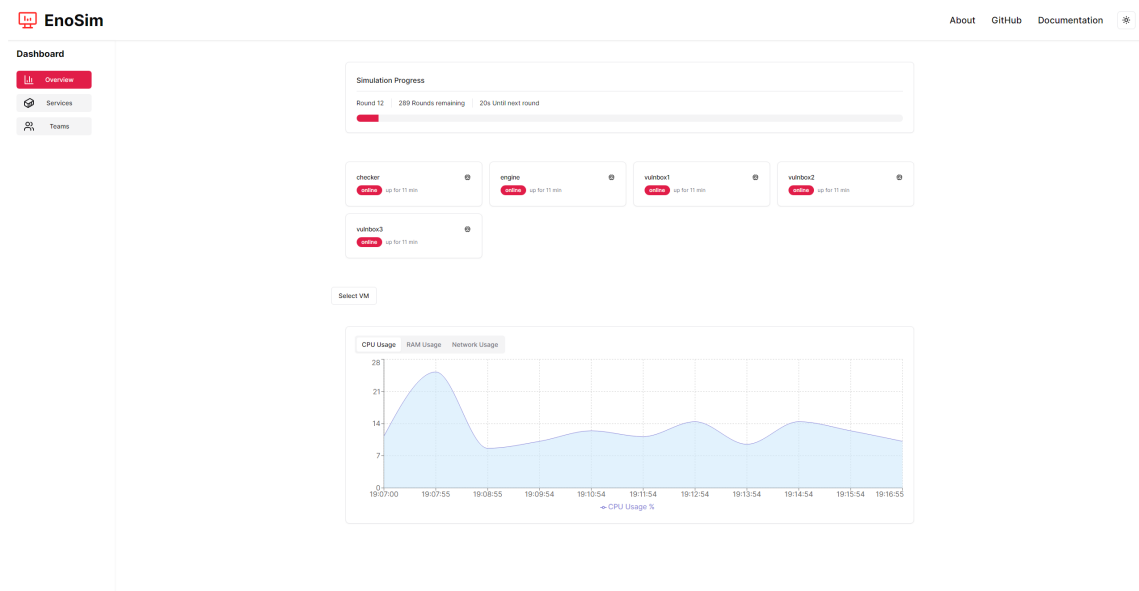


Figure 4.4.: Monitoring User Interface

4.1.5. Performance Considerations

Taking into account that the orchestrator has been implemented using Python, which is less performant in many ways than a system-level language like, for example, Rust, some considerations have to be made in regards to performance, speed, and efficient resource usage on the system acting as the orchestrator. [10] This is because the number of checker task requests that it has to process in each round can become significantly large.

For instance, if there are 100 teams exploiting 10 flagstores, the orchestrator will have to send out $10 \cdot (100 - 1)$ checker requests for each team, which makes a total of 99000 requests it has to send in rapid succession. This also does not yet take into account the traffic that the orchestrator generates to monitor the system and Docker statistics on each virtual machine. Due to the tasks that the orchestrator performs being especially IO-oriented (Input Output), asynchronous programming presents a viable option to improve the performance of these tasks. [17]

More specifically, with so-called *asynchronous task groups* that manage the asynchronous execution of functions, the exploit process could be sped up considerably. The speedup became especially significant as the number of requests that had to be processed in quick succession increased to a large number. The code extract in Listing 4.4 illustrates the implementation details of the exploit procedure.

```
1 async with asyncio.TaskGroup() as task_group:
2     tasks = [
3         task_group.create_task(
4             self.orchestrator.exploit(
5                 self.round_id, team, self.setup.teams.values()
6             )
7         )
8         for team in self.setup.teams.values()
9     ]
10 for task_index, task in enumerate(tasks):
11     team_flags[task_index].append(task.result())
```

Listing 4.4: Asynchronous Exploits

4.2. A Probabilistic Model for CTF Simulations

An important aspect of simulating a realistic CTF competition lies in the creation of a model that relies on probabilities to imitate the process of teams coming up with a new patch or exploit for a service and flagstore in any given round, which is closely related to their level of expertise.

For this, both the distribution of experience levels that should be mapped to the teams as well as the average probability of success for each experience level, which represents a team's likelihood to discover a new exploit or patch in any given round, need to be determined.

4.2.1. Data Sources for Scoreboards

The initial stage in developing this model lies in evaluating data that was generated in previous CTF competitions. For this purpose, the scoreboards of past CTF competitions had to be acquired, and there existed two reliable sources for such scoreboards that will be analyzed in the subsequent steps to determine an appropriate distribution of experience levels and success probabilities.

The first source came in the form of the numerous scoreboards of past CTF competitions that are publicly available on ctftime.org. [39] The second source were scoreboard files in JSON format that were made publicly available on the GitHub page of the organizers of the past iterations of the Enowars attack-defense competition. [35]

4.2.2. Scores as a Normal Distribution

Since many comparisons of human expertise follow a normal distribution, it may be assumed that the scoreboards generated in past CTF competitions would similarly follow a normal distribution. [22, 19] Following this premise, it would then be possible to derive a set number of experience levels based on standard deviations from the mean score, which could be used to categorize teams and generate a point average for each experience level.

A more concrete implementation of this idea discriminates between five distinct levels of expertise, ranging from complete beginners who achieved a score within minus three to minus two standard deviations to professionals who achieved a score within plus two to plus three standard deviations from the mean. This is further illustrated in Figure 4.5.

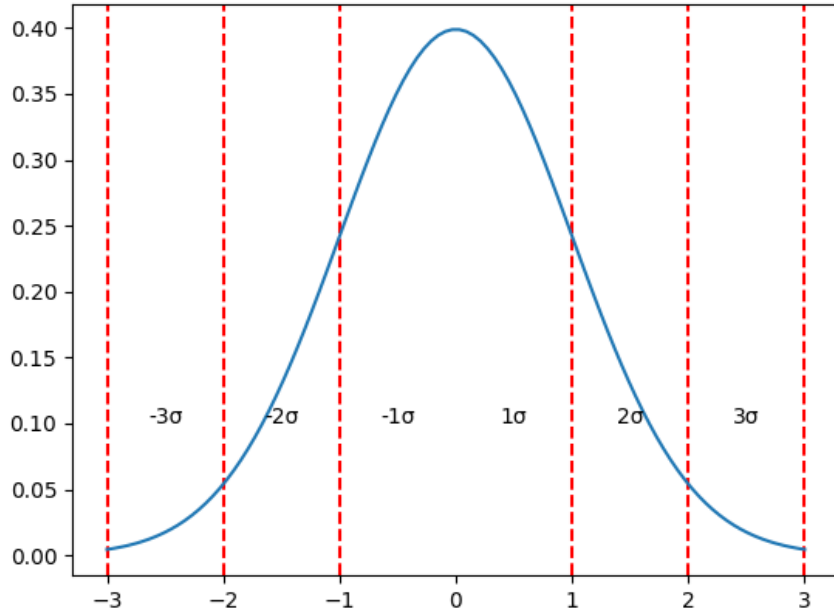


Figure 4.5.: Normal Distribution Concept

Following this approach, a Python script was developed that aggregates every score on a given scoreboard and then calculates the mean μ and the standard deviation σ of these scores using the following formulas for a competition with N teams, where x_i represents the score achieved by team i . [3]

$$\mu = \frac{\sum_{i=1}^n x_i}{N}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \mu)^2}{N}}$$

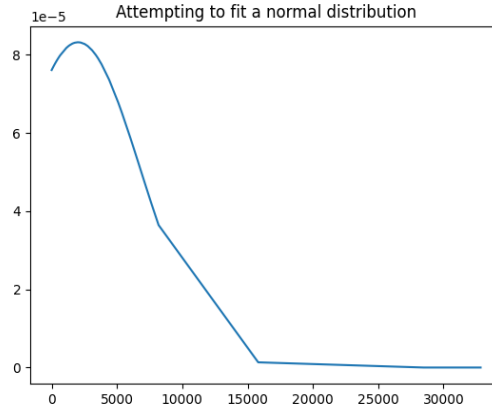


Figure 4.6.: Attempt at a Normal Distribution

Utilizing the standard deviation and mean in combination with the *scipy* datascience library facilitated mapping the dataset to a normal distribution and displaying the outcome. [7] Figure 4.6 illustrates the graph that was created in this manner using a scoreboard obtained from the Enowars 7 CTF competition. [40]

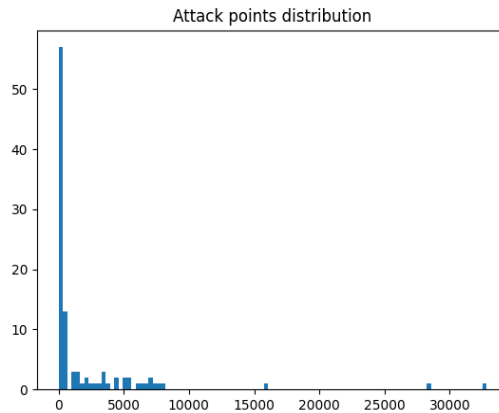


Figure 4.7.: Attack Score Distribution

It quickly became apparent that the dataset does not in fact follow a normal distribution, which is further emphasized when examining the very uneven distribution of scores that are presented in Figure 4.7. It can be seen that the vast majority of teams have not been able to exploit a single flagstore, which indicates that they ended the competition with an attack score of zero. For this reason, an alternative method will be proposed in the following section.

4.2.3. Finding an Experience Level Distribution

Another method for categorizing teams into different levels of experience could be found quite simply in calculating the percentage of points that the team attained when compared to the highest score in the competition. Applying this method to distribute teams into five distinct levels of expertise, the teams would have to be divided in increasing steps of 20 percent. This means that a team that has reached anywhere from 0 to 20 percent of the high score would be classified as a complete beginner, while teams that achieved between 80 and 100 percent of the high score would be classified as professionals.

A problem that appears with this approach, at least in the scoreboards that have been analyzed, is that the vast majority of teams will be classified as complete beginners since they have not managed to exploit a single flagstore. This may not be a very accurate representation of the real skill levels these teams possess. However, these scores presented the only way for deriving levels of expertise, and for the purpose of the simulation, these expertise levels sufficed to mimic the exploit traffic of a real CTF competition.

4.2.4. Deriving Success Probabilities

Having developed a method for dividing teams into levels of expertise based on the number of points they attained in a competition, it becomes possible to calculate a point average for each expertise level and use this average to determine a success probability. The point average will simply be derived as the median between the highest score and the lowest score that fall under the same level of expertise. In the following, the specific team with a score closest to this point average will be selected for further analysis to derive a success probability for the experience level. This section seeks to describe in detail the necessary steps to determine this probability through the analysis of the data obtained from a competition's scoreboard.

For this, it needs to be understood that, in terms of probability, the event of a team finding an exploit or a patch for a service follows a geometric distribution. Each round of the competition is representative of one Bernoulli trial, where the act of a team starting to exploit a service counts as a success. [18] Knowing that, it becomes possible to utilize the existing formulas for the geometric distribution to derive the likelihood of a team finding a new exploit for any given service at the beginning of a round. The formula for the expected value of a geometrically distributed random variable X is defined as follows:

$$E(X) = \frac{1}{p}$$

The expected value $E(X)$ in this case will be the number of rounds that a team requires until their first successful exploit of a particular service. The probability p stands for the team's likelihood of success in any given round, which is exactly what has to be discovered. Since the expected value can be calculated from the scoreboard, the above equation should be rearranged as follows to discover p :

$$p = \frac{1}{E(X)}$$

This now allows us to derive the probability of finding an exploit a team possesses for a specific service. However, it still remains to calculate the expected value, $E(X)$, which can be achieved by analyzing the attack score that a team has achieved on the particular service. For this, there are two calculations that need to be made. The initial calculation includes finding out what percentage of the maximum score that could be attained for a service the team managed to attain. This can be found by dividing the team's service-specific attack score by the total number of rounds multiplied by the average points achievable in each round on the given service:

$$\text{percent}_{\max} = \frac{\text{SCORE}_{\text{team/service}}}{\text{rounds}_{\text{total}} \cdot \text{SCORE}_{\text{service/round}}}$$

This percentage can then be used to derive the number of rounds a team needed until they began exploiting the particular service, which represents the expected value $E(X)$. To achieve this, we multiply the percentage of the maximum score achievable on the service with the total number of rounds, which will result in the number of rounds that the team spent exploiting. Subtracting this number from the total number of rounds yields the number of rounds needed until the first success:

$$E(X) = \text{rounds}_{\text{total}} - \text{percent}_{\text{max}} \cdot \text{rounds}_{\text{total}}$$

Using this sequence of steps enables us to calculate the success probability a team has on a specific service; however, their total success probability needs to take into account all of their service-specific success probabilities, so it can be derived as the sum of all service-specific success probabilities, assuming that there are n services played and p_i is the success probability for service i .

$$p'_{\text{total}} = \sum_{i=1}^n p_i$$

Lastly, there are some slight adjustments that have to be made to the total success probability to take into account that less experienced teams may find an exploit but fail to take advantage of the exploit by not having the technical knowledge required to automate the exploit process against other teams. For this purpose, we introduce a scalar α that readjusts the success probability based on a team's level of expertise. An additional factor that should be pointed out is that the success probability serves for both discovering new exploits and patches in the simulation setting and thus should be doubled to ensure that a team's probability of finding a new exploit is exactly as high as their probability of patching a service.

$$p_{\text{total}} = 2\alpha \cdot \sum_{i=1}^n p_i$$

4.2.5. Integration Into the Simulation Program

Having determined methods for calculating both a distribution of experience levels and a success probability for each experience level, these need to be integrated into the simulation code. This should happen in such a way that the user-specified number of simulated teams will be generated with an accurate distribution of experience levels, and the teams will be equipped with their success probabilities so that they can be used in the random test described in Section 4.1.1.

These requirements were fulfilled in the main application by a class that is primarily responsible for the correct generation of teams and an enumerator that stores, for each level of expertise, two pieces of information. The first piece of information conveys the prevalence of a team's particular experience level in the simulated competition, while the second piece of information stores the likelihood that a team of this experience level possesses for exploiting or patching a service.

The class that is responsible for generating teams additionally defines a method for analyzing a scoreboard file in JSON format that users can optionally supply. This method will be called in the initial configuration phase of the simulation setup, and it calculates an experience level distribution along with the related probabilities according to the steps laid out in the previous section. It will then use these values to readjust the distribution and probability values stored in the experience level enumerator. If a user does not supply a scoreboard file, this method will return a set of default values to be used for the distribution and success probabilities. Listing A.3 and Listing A.4 illustrate, firstly, the class method responsible for generating teams following the analysis of such a scoreboard file and, secondly, the enumerator that was used to store for each experience level the required information.

4.3. Different Modes of Simulation

A last crucial feature that should be included in the simulation program comprises different modes of running the simulation, which allows organizers of CTF competitions to observe their service and checker implementations under realistic circumstances as well as under conditions of extensive load. For this purpose, three additional simulation modes have been devised, facilitating variations of a stress testing environment. Each of these stress testing modes will offer a particular benefit over the other variants, which will be described in detail in the following sections.

4.3.1. Basic Stress Testing

One stress testing environment, which has been inspired by the existing setup of the Enowars organization, consists of an engine, a checker, and a small number of vulnbox virtual machines. [35] This mode would require a readjustment of the number of flags and noise deposited into the services in every round to a higher number. For instance, it would result in 50 flags and 50 instances of noise being deposited in each service at the beginning of every round, which results in services and checkers being exposed to a heavy load. A problem with this basic mode of stress testing is that the services are not being tested on every functionality, and their intended vulnerabilities are not being tested due to the lack of exploit methods being executed.

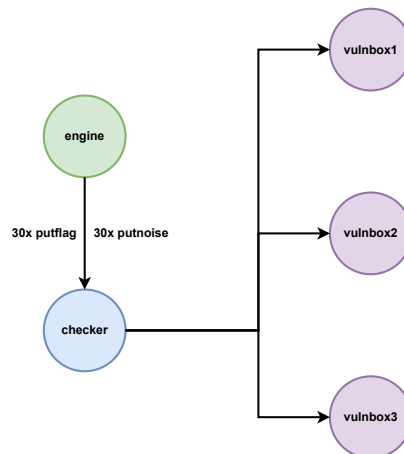


Figure 4.8.: Basic Stress Testing

4.3.2. Regular Stress Testing

Another mode of stress testing simply entails creating a CTF simulation infrastructure as laid out in the previous chapters and, instead of determining exploits via a random test, having every team exploit every other team right from the beginning of the simulation. Some drawbacks of this approach are that building an entire CTF infrastructure for the main purpose of stress testing is typically time-consuming and can, depending on the cloud provider, quickly become quite expensive. A regular CTF setup will include anywhere from 50 to 100 vulnboxes, which all have to be deployed, configured, and paid for by the hour. For this reason, less complicated infrastructures are required that mitigate the costs of such a setup.

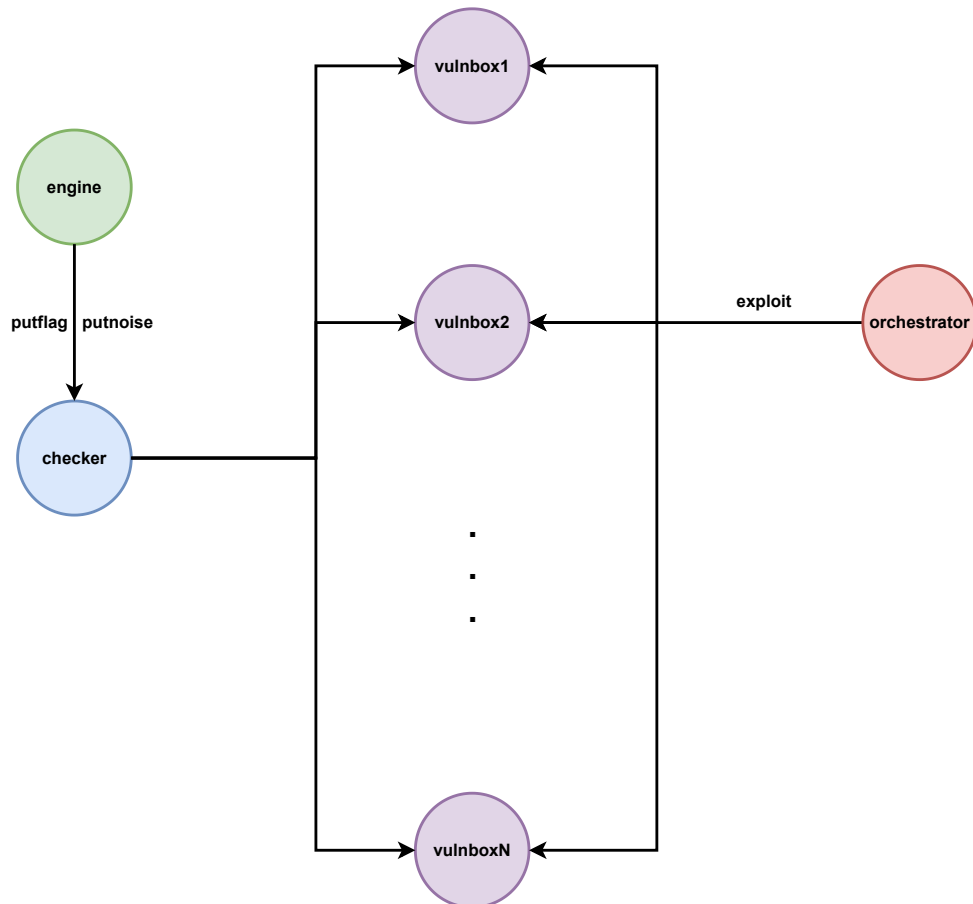


Figure 4.9.: Regular Stress Testing

4.3.3. Combined Stress Testing

A stress testing environment that solves both the issues of cost and effort concerning the latter mode and the lack of exploit traffic in the former mode is a simple combination of both modes. This way, a simulation setting would have to be generated with only a small number of components, but the services would still be tested on every front. This mode of simulation entails creating a CTF infrastructure consisting of a small number of vulnboxes, an engine component, and a checker component. The engine will deploy a large number of flags and noise into the services running on the vulnboxes, while each team will be made to exploit every other team on every available flagstore.

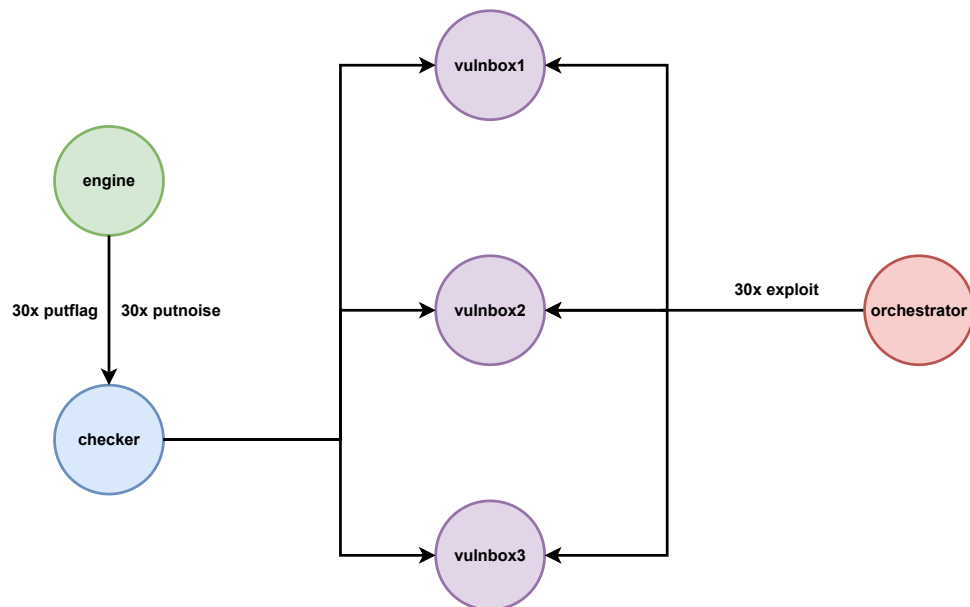


Figure 4.10.: Combined Stress Testing

5. Results

This chapter aims to present the resulting scores and monitoring information from simulating an attack-defense CTF competition according to the methods and procedures laid out in the previous chapters. It will further lay the foundation for a more intricate interpretation of the generated data in the subsequent chapter.

5.1. Scoreboard Comparison

Firstly, a comparison will be made between a scoreboard taken from the Enowars 7 attack-defense CTF competition and its simulated counterpart. [40] More specifically, to facilitate the simulated competition, the scoreboard file was analyzed according to the steps laid out in chapter 4 to derive a distribution of experience levels and success probabilities, and then the competition was run over a course of 400 rounds, which resulted in approximately 7 hours.

Place	Team	CTF points
1	C4T BuT S4D	68526.763
2	TeamItaly	61471.707
3	Bushwhackers	43569.422
4	Czech Cyber Team	41321.049
5	aetrulia	40749.683
6	CyberSecurityAustria	36603.675
7	The Flat Network Society	36044.700
8	SKSD	36010.962
9		35949.926
10	ECSC TeamNL 2023	35547.156
11	ECSC Team France	35542.785
12	FluxFingers	35168.190
13	saarsec	35068.851
14	FaKappa	34687.339
15	BinaryBears	34387.918

(a) Original Scoreboard

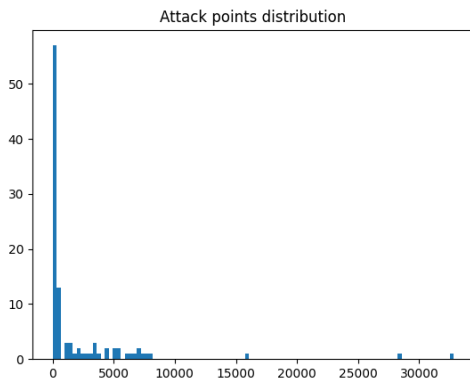
Place	Team	CTF points
1	Pomchi	54550.0
2	Goblin Shark	50090.0
3	Umbrellabird	47820.0
4	Jack-Chi	44590.0
5	Quokka	43650.0
6	Assassin Bug	42620.0
7	Pacific Spaghetti Eel	41460.0
8	Forest Cuckoo Bumble Bee	40500.0
9	Eastern Racer	39890.0
10	Chihuahua Mix	38460.0
11	Midget Faded Rattlesnake	38380.0
12	Common Spotted Cuscus	38360.0
13	Norwegian Elkhound	38210.0
14	Crappie Fish	36720.0
15	Lone Star Tick	36630.0

(b) Simulation Scoreboard

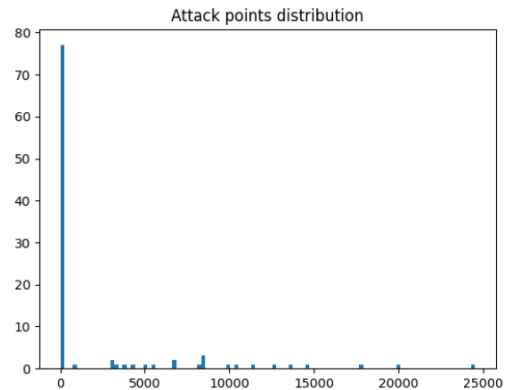
Figure 5.1a displays the scoreboard that was generated during the course of the original Enowars 7 attack-defense competition. Figure 5.1b depicts the scoreboard generated from a simulation of that same competition. The following values were extracted from the original scoreboard for a distribution of teams and their specific success probabilities for exploiting and patching services.

Experience Level	Success Probability	Distribution Percentage
Beginner	0.3%	91%
Amateur	1.1%	6%
Intermediate	2.1%	1%
Advanced	3%	0%
Professional	5.8%	2%

It becomes apparent that the distribution of scores in the simulated scoreboard is similar to the real scoreboard, with the difference that the scores are distributed slightly more evenly. The highest score in the simulated version is lower than the highest score in the real competition, and the lower scores are slightly higher than the lower scores in the real competition. Furthermore, a larger number of simulated teams have achieved an attack score of exactly zero, while in the realistic competition, there were more teams that attained a score slightly higher than zero. These differences are further highlighted in the following comparison between the achieved attack scores of the teams in the real competition and the simulated teams.



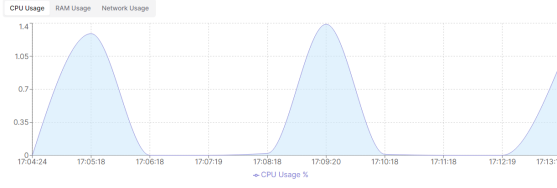
(a) Original Attack Scores



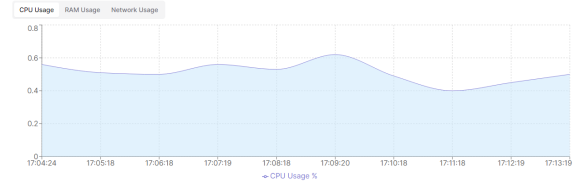
(b) Simulation Attack Scores

5.2. Monitoring Information

In this section, the monitoring information that was generated during a simulated competition will be presented. For this, the focus will especially lie on the generated resource utilization graphs for each service and checker implementation, as well as the generated monitoring information for the virtual machines.

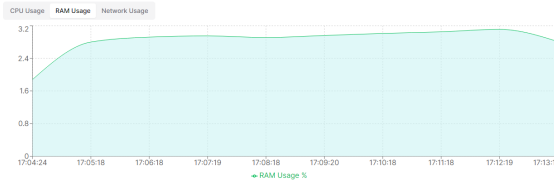


(a) Service CPU Usage

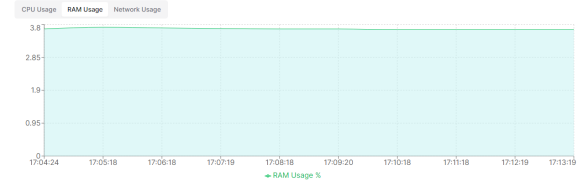


(b) Checker CPU Usage

Figure 5.3a depicts the CPU usage graph that has been generated for a specific service. Figure 5.3b depicts the CPU utilization graph that has been generated for the checker of the same service. Both of these were generated during a simulated competition that was executed with the regular stress testing configuration elaborated on in Section 4.3.2. It becomes apparent that for this particular service and checker implementation, there are no excessive CPU spikes, and the CPU utilization remained at a moderate level of approximately 1% for the service and 0.5% for the checker.



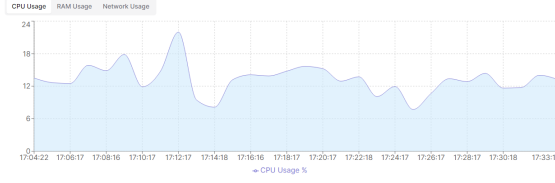
(a) Service RAM Usage



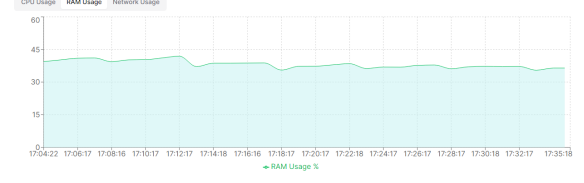
(b) Checker RAM Usage

The above graphics depict the RAM utilization statistics for the service and checker applications. The RAM usage remained at a very consistent level of approximately 3% for the service and 3.5% for the checker.

The following statistics depict the resource utilization statistics of the engine, checker, and vulnbox virtual machines generated in the same simulation. It should be noted that the virtual machines used in this configuration each included a single-core CPU, 2 gigabytes of RAM, and 20 gigabytes of storage.

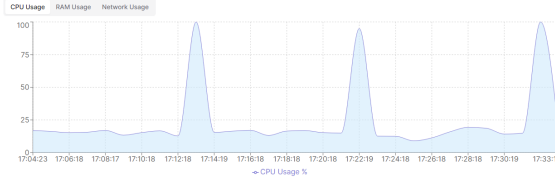


(a) Engine CPU Usage

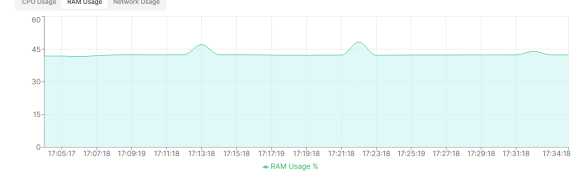


(b) Engine RAM Usage

The overall CPU utilization on the engine virtual machine fluctuated between 12% and 18%. The RAM utilization steadily remained around 40%.

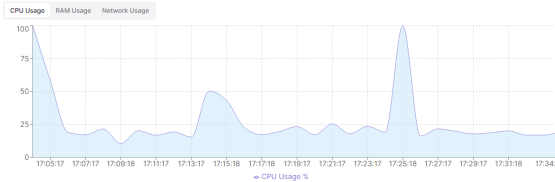


(a) Checker CPU Usage



(b) Checker RAM Usage

The checker virtual machine maintained a CPU utilization of approximately 20% with occasional spikes to 100%, which may indicate problematic checker implementations. The RAM usage did not exceed 40%.



(a) Vulnbox CPU Usage



(b) Vulnbox RAM Usage

The vulnboxes utilized on average 20% of the CPU, with infrequent spikes to 100%. The RAM usage remained steadily around 60%.

6. Conclusions

To conclude this work, this chapter seeks to evaluate and interpret the results presented in the previous chapter. For this purpose, the results of simulating a CTF competition and the monitoring data generated during such a competition will be examined. Lastly, some suggestions will be made for any future work attempting to build on the foundation laid out here.

6.1. Interpreting Simulation Data

Examining the presented distribution of experience levels and the graph of achieved attack points in the real CTF competition indicates that the distribution has been calculated successfully according to the methods proposed. 91% of the teams in the competition had achieved a score between 0-20% of the highest score. Therefore, 91% percent of simulated teams have been categorized as beginners. Following this, it could be observed that the success probabilities that have been calculated for each team resulted in a final simulation scoreboard that shows significant similarities to the real scoreboard.

Unfortunately, only one competition has been simulated, which might not offer enough data to arrive at a conclusion in terms of how well the simulation program is able to emulate a wider range of CTF competitions. This was due to previous competitions using an older version of the game engine that calculated scores differently and could not be accounted for with the simulation program developed in this work. [25] However, it can be concluded that the methods presented enable a sufficiently accurate simulation of CTF competitions using the most up-to-date scoring formula. To arrive at an even more realistic simulation and, as a result, more similar scoreboards, the following section will propose some suggestions.

6.2. Considerations for More Realistic Simulations

When comparing the scoreboards of real CTF competitions with their simulated counterparts, noticeable differences presented themselves in the attack scores accumulated during the competitions. These included, for instance, a larger number of teams with an attack score of exactly zero in the simulations and a more even distribution of scores across the entire set of teams. This is likely due to the formula used to calculate the success probabilities for the teams, which has been described in detail in chapter 4. The scalar α used in the formula $p_{\text{total}} = 2\alpha \cdot \sum_{i=1}^n p_i$ may be overadjusting the overall probability for less experienced teams, which leads to a larger number of teams achieving an attack score of exactly zero. In the simulation setting, the percentage of the maximum score for a particular experience level has been used for α , which equates to a value of 0.2 for beginner teams, for instance. To reach a higher level of accuracy in the calculation of a team's success probability, it may be necessary to discover a different way of factoring in the team's experience level into their particular success probability.

6.3. Interpreting Monitoring Data

Examining the resource utilization statistics generated for a particular service and its checker implementation revealed sufficiently detailed insight into the performance of these applications over the course of the simulated competition to be able to identify potential issues that may appear in the form of either consistently high CPU or RAM utilization or only intermittent appearances of spikes in these performance indicators.

Similarly, the resource utilization statistics generated for each virtual machine allowed an insight into the performance of each component in the CTF infrastructure. It could be observed in the presented statistics that the checker virtual machine had significant spikes in its CPU utilization, which would indicate to the user that they should either optimize their checker implementations or use a virtual machine with better system specifications.

It can therefore be concluded that the monitoring information accumulated during the simulation of a CTF competition proved detailed enough to be able to identify insufficiencies on both the level of particular service and checker implementations, as well as the virtual machines making up the CTF infrastructure.

6.4. Suggestions for Future Work

Various improvements can still be made to the formula proposed in chapter 4 for calculating a team's success probability. These include the changes to the scalar α that were suggested in the preceding section as well as taking into account the particular formula that the game engine uses to calculate the attack scores of the teams in every round. It has been mentioned that a difficulty presented itself in the simulation program being unable to simulate competitions that were held with an older version of the game engine software that used a different scoring formula. For this reason, it may be a good idea to include an option in the configuration section of the simulation that lets the user choose the specific version of the game engine that should be used for the simulated CTF competition.

While the monitoring system was devised for the purpose of tracking the simulation process and the performance of each specific component, it may not be able to offer sufficiently detailed information regarding the causes of potential performance issues. For instance, there are numerous reasons why the CPU utilization statistics for the checker virtual machine that could be observed in chapter 5 indicated occasional load spikes. These could have been caused by the checker implementations not being sufficiently optimized for performance, but they also could have been caused by entirely unrelated issues, such as, for example, a lack of disk storage on the virtual machine that significantly slows down its performance. It would therefore be helpful to include more detailed analytical reports in the monitoring system.

List of Figures

3.1. CTF Infrastructure	8
3.2. CTF Simulation Infrastructure	16
4.1. Simulation Workflow	18
4.2. Vulnbox Configuration	21
4.3. Exploit Workflow	21
4.4. Monitoring User Interface	24
4.5. Normal Distribution Concept	27
4.6. Attempt at a Normal Distribution	28
4.7. Attack Score Distribution	28
4.8. Basic Stress Testing	33
4.9. Regular Stress Testing	34
4.10. Combined Stress Testing	35

Listings

3.1. Configuration	9
3.2. Secrets	10
3.3. Build Script Template	11
3.4. Configuration Script Template	14
4.1. Random Test	19
4.2. Updating a Random Flagstore	19
4.3. Collecting System Statistics	23
4.4. Asynchronous Exploits	25
A.1. Terraform Script Template	50
A.2. Template Conversion	51
A.3. Generating Teams	52
A.4. Experience Enumerator	52

Bibliography

- [1] Shivaraj Kengond Mohammed Moin Mulla Amit M Potdar, Narayan D G. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171:1419–1428, 2020.
- [2] Dylan Fox Daniel Votipka Michelle L. Mazurek Benjamin Gregory Carlisle, Michael Reininger. On the other side of the table: Hosting capture-the-flag (ctf) competitions an investigation from the ctf organizer’s perspective. 2020.
- [3] Pritha Bhandari. How to calculate standard deviation. <https://www.scribbr.com/statistics/standard-deviation/>. Accessed: 2023-10-20.
- [4] Gianluca Canitano. Development of framework for attack/defense capture the flag competition, 2021.
- [5] Software Freedom Conservancy. Git documentation. <https://git-scm.com/docs>. Accessed: 2023-10-20.
- [6] Robert G. Byrnes Daniel J. Barrett, Richard E. Silverman. *SSH, The Secure Shell: The Definitive Guide, 2nd Edition*. O’Reilly Media, 2005.
- [7] SciPy Developers. Scipy user guide. <https://docs.scipy.org/doc/scipy/tutorial/index.html>. Accessed: 2023-10-20.
- [8] Lucas Druschke. Checker protocol v2. https://github.com/enowars/specification/blob/main/checker_protocol.md. Accessed: 2023-10-20.
- [9] Lucas Druschke. Enochecker test. https://github.com/enowars/enochecker_test. Accessed: 2023-10-04.
- [10] Haque Nawaz Fanila Ali Agha. Comparison of bubble and insertion sort in rust and python language. *International Journal of Advanced Trends in Computer Science and Engineering*, 10(2):1020–1025, 2021.

- [11] Python Software Foundation. Subprocess documentation. <https://docs.python.org/3/library/subprocess.html>. Accessed: 2023-10-04.
- [12] Hetzner Online GmbH. Hetzner documentation. <https://docs.hetzner.com/cloud/>. Accessed: 2023-10-20.
- [13] Hamid Mcheick Hafedh Mili, Amel Elkharraz. Understanding separation of concerns. 2004.
- [14] Hashicorp. Packer documentation. <https://developer.hashicorp.com/packer/docs>. Accessed: 2023-10-20.
- [15] Hashicorp. Terraform documentation. <https://developer.hashicorp.com/terraform>. Accessed: 2023-10-20.
- [16] Red Hat. Ansible documentation. <https://docs.ansible.com/>. Accessed: 2023-10-20.
- [17] Caleb Hattingh. *Using Asyncio in Python*. O'Reilly Media, 2020.
- [18] Prof. Dr. Hans-Wolfgang Henn. Einführung in die stochastik. https://www.ghg-alsdorf.de/fachkonferenz/mathe/Einfuehrung_Stochastik_gesamt.pdf, 2002. Accessed: 2023-10-16.
- [19] Norbert Henze. *Stochastik: Eine Einführung mit Grundzügen der Maßtheorie*. Springer-Verlag GmbH, 2019.
- [20] Docker Inc. Docker remote access. <https://docs.docker.com/config/daemon/remote-access/>. Accessed: 2023-11-12.
- [21] Vercel Inc. Nextjs documentation. <https://nextjs.org/docs>. Accessed: 2023-10-20.
- [22] Aidan Lyon. Why are normal distributions normal? *The British Journal for the Philosophy of Science*, 65(3):621–649, 2014.
- [23] Dominik Maier. Checker documentation. <https://enowars.github.io/docs/service/checker/checker/>. Accessed: 2023-11-10.
- [24] Dominik Maier. Checker methods. <https://enowars.github.io/docs/infrastructure/round/>. Accessed: 2023-11-10.
- [25] Dominik Maier. Engine documentation. <https://github.com/enowars/>

- EnoEngine. Accessed: 2023-11-10.
- [26] Dominik Maier. Enowars documentation. <https://enowars.github.io/docs/play/general/>. Accessed: 2023-11-10.
- [27] Dominik Maier. Enowars infrastructure. <https://enowars.github.io/docs/>. Accessed: 2023-11-10.
- [28] Dominik Maier. Service tenets. <https://enowars.github.io/docs/service/tenets/>. Accessed: 2023-11-10.
- [29] Microsoft. Azure documentation. <https://learn.microsoft.com/en-us/azure/?product=popular>. Accessed: 2023-10-20.
- [30] Microsoft. What is iaas? <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-iaas/>. Accessed: 2023-10-04.
- [31] Piyush Pandey Navin Sabharwal, Sarvesh Pandey. *Infrastructure-as-Code Automation Using Terraform, Packer, Vault, Nomad and Consul*. Apress Berkeley, 2021.
- [32] Sebastian Neef. Example service. <https://github.com/enowars/enowars-service-example>. Accessed: 2023-10-20.
- [33] Jannik Novak. Simulation github. <https://github.com/ashiven/enosimulator>. Accessed: 2024-01-11.
- [34] Benedikt Radtke. Enoctfportal. <https://github.com/enowars/EnoCTFPortal>. Accessed: 2023-10-20.
- [35] Benedikt Radtke Julian Beier Lucas Druschke Sebastian Neef, Otto Bitner. Enowars infrastructure github page. <https://github.com/enowars/bambictf>. Accessed: 2023-10-04.
- [36] Sang Kil Cha Seongil Wi, Jaeseung Choi. Git-based ctf: A simple and effective approach to organizing in-course attack-and-defense security competition. In *ASE @ USENIX Security Symposium*, 2018.
- [37] Alberto Lluch Lafuente Antonio Ruiz Martínez Karo Saharinen Antonio Skarmeta Pierantonio Sterlini Simone Fischer-Hübner, Matthias Beckerle. Quality criteria for cyber security moocs. In *Information Security Education*.

- Information Security in Action*, pages 46–60. Springer International Publishing, 2020.
- [38] Matt Bishop Steven Furnell. Education for the multifaith community of cybersecurity. In *Information Security Education. Information Security in Action*, pages 32–45. Springer International Publishing, 2020.
- [39] CTFTTime Team. Ctftime home page. <https://ctftime.org>. Accessed: 2023-10-04.
- [40] CTFTtime team. Enowars 7 scoreboard. <https://ctftime.org/event/2040>. Accessed: 2023-11-08.
- [41] CTFTTime Team. Enowars weight rating. <https://ctftime.org/event/2040>. Accessed: 2023-10-04.
- [42] Isaac D. Scherson Umesh Krishnaswamy. A framework for computer performance evaluation using benchmark sets. *IEEE Transactions on Computers*, 49(12):1325–1338, 2000.

A. Appendix


```
1 variable "vulnbox_count" {
2   type      = number
3   default   = _placeholder_
4 }
5
6 locals {
7   dynamic_vm_map = merge(
8     var.vm_map,
9     {
10      for vulnbox_id in range(1, var.vulnbox_count + 1) :
11        "vulnbox${vulnbox_id}" => {
12          name = "vulnbox${vulnbox_id}"
13        }
14      }
15    )
16   vm_map = local.dynamic_vm_map
17 }
18
19 variable "vm_map" {
20   type = map(object({
21     name = string
22   }))
23   default = {
24     "engine" = {
25       name = "engine"
26     }
27     "checker" = {
28       name = "checker"
29     }
30   }
31 }
```

Listing A.1: Terraform Script Template

```
1 # Configure vulnbox count in variables.tf
2 VM_COUNT_LINE = 2
3 await replace_line(
4     f"{self.setup_path}/variables.tf",
5     VM_COUNT_LINE,
6     f"    default = {self.config.settings.teams}\n",
7 )
8
9 # Configure vm image references in variables.tf
10 sub_id = self.secrets.cloud_secrets.azure_service_principal["
    subscription-id"]
11 basepath = f"/subscriptions/{sub_id}/resourceGroups/vm-images/
    providers/Microsoft.Compute/images"
12 await insert_after(
13     f"{self.setup_path}/variables.tf",
14     "    name = string",
15     "    subnet_id = number\n"
16     + "    size = string\n"
17     + "    source_image_id = string\n"
18     if self.use_vm_images
19     else "",
20 )
21 await insert_after(
22     f"{self.setup_path}/variables.tf",
23     '    name = "engine"',
24     f"    subnet_id = {self.config.settings.teams + 2}\n"
25     + f'    size = "{self.config.setup.vm_sizes["engine"]}"\n'
26     + f'    source_image_id = "{basepath}/{self.config.setup.
vm_image_references["engine"]}"\n'
27     if self.use_vm_images
28     else "",
29 )
```

Listing A.2: Template Conversion

```
1 def generate(self) -> Tuple[List, Dict]:
2     ctf_teams = []
3     setup_teams = dict()
4     team_id_total = 0
5
6     for experience, teams in self.team_distribution.items():
7         for team_id in range(1, teams + 1):
8             id = team_id_total + team_id
9             ctf_team = self._generate_ctf_team(id)
10            ctf_teams.append(ctf_team)
11            setup_team = self._generate_setup_team(id, experience)
12            setup_teams.update(setup_team)
13
14            team_id_total += teams
15
16     return ctf_teams, setup_teams
```

Listing A.3: Generating Teams

```
1 class Experience(Enum):
2     """
3     an enum representing the experience level of a team.
4
5     The first value stands for the probability of the team
6     exploiting or patching a
7     vulnerability in any given round. The second value stands for the
8     prevalence of an
9     experience level in real ctf competitions and will be used to
10    distribute teams in
11    the simulation.
12    """
13
14    BEGINNER = (0.015, 0.08)
15    AMATEUR = (0.04, 0.54)
16    INTERMEDIATE = (0.06, 0.29)
17    ADVANCED = (0.09, 0.07)
18    PRO = (0.12, 0.02)
19    HAXXOR = (1, 1)
```

Listing A.4: Experience Enumerator