Bachelor Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Gamification of Ethical Hacking Lab

Johannes Schraud
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr.-Ing. Alexandra Dmitrienko
First Reviewer

M.Sc. Christoph Sendner
First Advisor

Submission
24. February 2023

www.uni-wuerzburg.de

Abstract

Education is an important topic. Increasing the effectiveness and efficiency of conveying knowledge is interesting for schools of all kinds, in the professional world, but also in the private sector. In this paper, we deal with this highly important topic. Thus, we implement a promising approach for improving education called gamification, for the module Ethical Hacking Lab (EHL) of the Secure Software Systems Group of the Julius-Maximilians-University of Wuerzburg. Summarized into one sentence, in this paper we improve the EHL through gamification in the form of a so-called Capture The Flag (CTF) project.

Specifically, on the one hand, we develop a CTF framework to provide and administer exercises. The framework also allows the integration of virtual laboratory environments. Furthermore, it contains statistics and various game elements to promote the game characteristics and to motivate the participants.

On the other hand, we develop virtual laboratory environments for the binary exploitation exercises of the EHL. Hereby, we aim to further motivate and engage the participants by enabling an actual execution of their developed exploits against a virtual target. In addition, our laboratory environments relieve the participants from tedious setup tasks, as well as the tutors of the EHL from the manual correction of the submitted solutions.

Furthermore, we integrate all other exercises of the EHL into our CTF framework. We implement this by extracting and integrating various questions about the exercises. By developing and executing detailed test plans, we ultimately ensure proper functionality of the CTF project.

Zusammenfassung

Die Lehre ist ein wichtiges Thema. So ist eine Steigerung der Effektivität und Effizienz bei der Vermittlung von Wissen interessant für Schulen jeglicher Art, in der Arbeitswelt, aber auch im privaten Bereich. Mit diesem überaus wichtigen Thema beschäftigen wir uns in dieser Arbeit. So setzen wir einen vielversprechenden Ansatz zur Verbesserung der Lehre namens Gamification, für das Modul Ethical Hacking Lab (EHL) der Secure Software Systems Group der Julius-Maximilians-Universität Würzburg um. Zusammengefasst in einem Satz verbessern wir in der vorliegenden Arbeit das EHL durch Gamification in Form eines sogenannten CTF Projektes.

Konkret entwickeln wir zum einen ein CTF Framework um Aufgaben bereitzustellen und zu administrieren. Auch ermöglicht das Framework das Einbinden von virtuellen Laborumgebungen. Weiterhin beinhaltet es Statistiken und verschiedene Spielelemente um den Spielcharakter zu fördern und die Teilnehmer zu motivieren.

Zum anderen entwickeln wir virtuelle Laborumgebungen für die „Binary Exploitation"-Aufgaben des EHLs. Hierdurch wollen wir die Teilnehmer noch mehr motivieren und begeistern, indem wir eine tatsächliche Ausführung, ihrer in den Aufgaben entwickelten Angriffe, gegen ein virtuelles Ziel, ermöglichen. Zusätzlich entlasten wir durch unsere Laborumgebungen die Teilnehmer von lästigen Setup-Aufgaben, sowie die Betreuer des EHLs von händischer Korrektur der abgegebenen Lösungen.

Weiterhin integrieren wir alle weiteren Aufgaben des EHLs in unser CTF Framework. Dies setzen wir durch das Herausarbeiten und Einbinden verschiedener Fragen zu den Aufgaben um. Durch die Entwicklung und Durchführung von ausführlichen Testplänen, stellen wir letztendlich eine ordnungsgemäße Funktionalität des CTF Projektes sicher.

Contents

1. Introduction 1

2. Background 5
2.1. Gamification . . . 5
2.2. Ethical Hacking Lab . . . 5
2.3. Capture The Flag . . . 7
2.4. Vulnerabilities . . . 8
2.4.1. Buffer Overflows . . . 8
2.4.2. Use-After-Free . . . 9
2.5. Tools . . . 10
2.5.1. Docker . . . 10
2.5.2. Vagrant . . . 10
2.5.3. Flask . . . 11

3. Related Work 13
3.1. Gamification . . . 13
3.2. Capture The Flag . . . 14

4. Architecture of CTF Project 17
4.1. Problem Statement . . . 17
4.2. Use-Case Scenario . . . 18
4.3. Requirement Analysis . . . 20
4.3.1. Virtual Laboratory Environments . . . 20
4.3.2. CTF Framework . . . 22
4.4. Structure of Virtual Laboratory Environments . . . 22
4.5. Structure of CTF Framework . . . 24

5. Implementation of CTF Project 27
5.1. Implementation of Virtual Laboratory Environments . . . 27
5.1.1. Implementation of Target Containers . . . 27
5.1.1.1. Overview . . . 28
5.1.1.2. Enabling of Attack . . . 29
5.1.1.3. Enabling of Privilege Escalation . . . 30
5.1.1.4. Provision of Unique Flags . . . 31
5.1.1.5. Preserving of host system security. . . . 32
5.1.1.6. Provision of Realistic User Experience . . . 33
5.1.2. Implementation of CTF Containers . . . 33
5.1.2.1. Overview . . . 33
5.1.2.2. Enabling of Attack . . . 34
5.1.2.3. Replication of Target Container Environment . . . 35
5.1.2.4. Provision of Necessary Resources . . . 36
5.1.2.5. Provision of Necessary Tools . . . 36

5.1.2.6. Preserving of host system security. 37
5.1.2.7. Provision of Realistic User Experience 37
5.2. Implementation of CTF Framework 38
5.2.1. Overview 38
5.2.2. Implementation of Frontend 39
5.2.2.1. Base Template 39
5.2.2.2. Login, Statistics, Challenges and Single Challenge Templates 40
5.2.2.3. Page Contents 40
5.2.2.4. Backend Communication 40
5.2.2.5. Additional Features 41
5.2.3. Implementation of Backend 41
5.2.3.1. Base Application 42
5.2.3.2. Routing 43
5.2.3.3. Database 43
5.2.3.4. Complex Data Types 44
5.2.3.5. Base Logic 45
5.2.3.6. Challenge Logic 48
5.2.3.7. Extended Challenge Logic 49
5.3. Integration of Remaining Challenges 51

6. Evaluation **53**
6.1. Security Analysis 53
6.1.1. Virtual Laboratory Environments 54
6.1.2. CTF Framework 55
6.2. Setup of Testing Environment 55
6.3. Evaluation of Virtual Laboratory Environments 56
6.4. Evaluation of CTF Framework 57
6.5. Test Results 58

7. Conclusion and Future Work **59**

List of Figures **60**

List of Tables **61**

Listings **63**

Acronyms **67**

Bibliography **69**

Appendix **73**
A. GUI Prototype of CTF Framework 73
B. Additional GUI Screenshots of CTF Framework 76
C. Resources for Virtual Laboratory Environments 77
C.0.1. Target Containers 77
C.0.2. CTF Containers 79
D. User Manuals 80
D.1. Virtual Laboratory Environments 80
D.2. CTF Framework 81
E. Testplans 83
E.1. Virtual Laboratory Environments 83
E.2. CTF Framework 85

1. Introduction

IT security is an increasingly relevant topic. Especially in connection with COVID-19, through the increased use of home offices, or in connection with cyber attacks as a means of war, one is increasingly confronted with the issue. The Hiscox Cyber Readiness Report 2022 [1], with a total of 5,181 participants from over eight countries, revealed that 48% of companies were victims of a cyber attack in 2022. From 2021 to 2022 alone, an increase from 43% to 48% was observed.

Not only companies but also private persons were affected. According to the 2022 Cyber Safety Insight Report [2] by NortonLifeLock, which is based on a survey of over 10,000 people from ten countries, 415.6 million people were victims of cyber attacks last year. The report also revealed that victims spent 4.4 trillion hours in total trying to solve problems caused by the attacks. This translates into an average of 7.8 hours per person. In addition, more than half of the victims reported a financial loss.

IT security is therefore not only in the political and economic interest, but also in the private interest. At the same time, the question arises as to how awareness of IT security can be promoted in companies and among private individuals, but also how education through schools, universities, or other platforms can be made more attractive, more efficient, and more effective.

One promising approach is gamification. In particular, several studies have shown that gamification can drastically change, the efficiency and the joy experienced during execution (p. 15)[3]. Also, several papers reported on the successful implementation of gamification in the field of IT security. [4, 5, 6, 7].

This thesis also aims to implement gamification with the goal of improving learning experience. EHL is a practice-oriented module of the Secure Software Systems group at chair two for computer science at the Julius-Maximilians-University of Wuerzburg. The module consists of nine individual exercises, which are currently all solved locally on the participants' computers and submitted by handing in the source codes. Thus, EHL uses a rather classic teaching method.

In contrast, online learning platforms like TryHackMe [8], Hack The Box [9], etc. take a different approach. Here, participants are familiarized with the topics in a playful way. Participants solve the exercises and are constantly rewarded by submitting so-called flags in the form of questions related to the task. Mostly, learning concepts are taught in a practical

way, for example, by simulating attacks in virtual laboratory environments. Furthermore, elements such as points, levels, or achievements are often used to continuously encourage and motivate the participants.

Using approaches like these in the area of schools or universities is by no means entirely new. As already mentioned, there is also various literature in the field of gamification, or more specifically, in the field of IT security about CTFs. However, most CTF projects have their peculiarities and are not one-to-one transferable to one another. In contrast to many of those works discussed there, the contribution of this thesis is more about the technical implementation of the CTF project itself rather than examining the effects of the gamification.

Therefore, in short, this thesis aims to improve EHL through gamification, by creating a CTF project. More specifically, we aim to motivate and engage the students, increase the efficiency of the exercise execution, and also relieve the tutors of EHL. The term CTF project is, of course, very general in terms of contents and scope, which is why we specify our contributions in the following.

Contributions. In particular, the contributions of this thesis are as follows:

(a) **We develop a CTF framework.** To host and administer the exercises we will develop a CTF framework. Furthermore, the CTF framework will embed the laboratory environments, that we additionally develop, in a simple way into the existing workflow of EHL. The CTF framework will also further motivate and engage the participants through various elements of gamification such as points, leaderboards, or performance graphs.

(b) **We develop customized virtual laboratory environments for the binary exploitation exercises of EHL.** For the binary exploitation exercises 2 and 3 of EHL we develop virtual laboratory environments. The laboratory environments consist of CTF Containers, which serve as a local development and test environment, and Target Containers, which act as the final target machines. This way, participants can actually experience their developed exploits in action. By providing the participants with CTF Containers, which replicate the Target Containers environment, we ensure that the locally developed exploits work in the Target Containers. Additionally, the CTF Containers provide the participants with all the necessary tools and resources for the exercise, which increases efficiency by saving a lot of work and time for setup tasks. Providing an individual Target Container for each participant moreover enables the integration of unique flags, which relieves the tutors of EHL from manual verification of every exercise submission.

(c) **We integrate all other exercises of EHL into our CTF project in the form of a questionnaire.** All other exercises, i.e. exercise 1 and 4 - 9, are integrated via various theory questions. This way we also motivate and engage the participants for the exercises that do not have an extra virtual laboratory environment. Furthermore, targeted questions can also guide the participants in the right direction and support them with solving the exercises. Additionally, by integrating all exercises into the CTF framework, we retain EHL as a single unit and encourage the future development of laboratory environments for the other exercises.

(d) **We evaluate our CTF project through test plans.** In order to evaluate the developed CTF project, we create test plans for the CTF framework as well as for the developed virtual laboratory environments. By then setting up a clean test environment, in which we meticulously work through the developed test plans, we ensure proper functionality.

Outline. This thesis is structured as follows. In Chapter 2, we clarify some important background terms for further understanding of this thesis. After placing our thesis in its literary context about *Gamification* and *Capture The Flag* in Chapter 3, we present the main part of this work. We start with the architecture of the CTF project in Chapter 4, to give a good overview at the start. After that, we present the implementation of the CTF framework in Chapter 5. Next, we evaluate our project in Chapter 6 and, finally, finish this thesis with a conclusion and discussion about possible future work in Chapter 7.

2. Background

In this chapter, we clarify important terms, concepts, and tools which are essential for the implementation of this thesis. In the following sections, we first look at the term *gamification* in Section 2.1), the term or module *Ethical Hacking Lab* in Section 2.2, as well as the term *Capture The Flag* in Section 2.3. We continue with providing details about important software vulnerabilities Section 2.4, and finish by presenting three tools in Section 2.5.

2.1. Gamification

The term gamification is important to clarify, as it represents the basic idea for this work. According to the Cambridge Dictionary, the concept of gamification is, in general terms, "the practice of making activities more like games in order to make them more interesting or enjoyable" [10]. In the context of this work, gamification will be carried out concretely through the development of CTF project (cf. Section 2.3). For further information, we refer the reader to [11, 12].

The professional literature defines gamification in more detail. For example, the *how* and *why*, are defined in more detail. Instead of "making activities more like games", definitions of various authors talk about game elements, game design techniques, game aesthetics, game techniques, or game thinking. This gives a little insight into the different forms and ways gamification can be implemented.

Also, it is often not plainly about "making them more interesting or enjoyable", but also about improving the motivation and even performance of the participants. This makes the concept much more appealing for result-oriented application fields. Application areas can be education, the work environment, as well as other areas.

The concrete implementation of gamification can be realized by game elements such as points, badges, leaderboards, performance graphs, narratives, or avatars. These elements can have different effects in different regards. Some of these elements reappear in the implementation of the CTFs for this project.

2.2. Ethical Hacking Lab

Ethical Hacking Lab is the name of a learning module of the Secure Software Systems Group at chair two for Computer Science at the Julius-Maximilians-University of Wuerzburg.

The module consists of a first part in which the participants are familiarized with the needed theoretical background knowledge. In a second part, the participants are confronted with different exercises about IT security such as known attacks and defenses in a very practical way. The part includes nine exercises. Moreover, the module enables the participants to obtain the *Cisco Netacad Cybersecurity Operations certificate* through an extra exam.

Various exercises of the Ethical Hacking Lab often consist of individual sub-exercises. To avoid ambiguity, we will refer to the exercises as Challenge 1 to Challenge 9 throughout this paper. Individual sub-exercises will be referred to as exercises. The following listing of exercises gives a quick overview of the scope of the module:

- **Challenge 1** is about *Assembly*. Students get a short introduction about things like basic instructions, registers, data, or functions in Assembly. The final main goals are to write an Assembly program that opens a shell as well as a program that prints out the number of arguments and counts down to zero. This challenge is crucial to master for some of the other upcoming challenges. Especially for the following challenges 2 and 3, at least a basic understanding of Assembly is beneficial, if not necessary.

- **Challenge 2** deals with *Buffer Overflows* (cf. Section 2.4.1), more specifically with heap buffer overflows. Students have to exploit a code injection as well as code reuse vulnerability of a given program running with the setUID bit set to perform privilege escalation.

- **Challenge 3** combines both techniques to so-called *Hybrid Exploits* to bypass the security feature of the set NX bit. Another exercise is to use a technique called *use-after-free* to, again, obtain a shell with higher privileges. This gives an outlook on other runtime problems apart from buffer overflows.

- **Challenge 4** opens up a new topic and deals with *PHP Viruses*. The participants first have to startup a vulnerable PHP web server with Apache2. Then, the objective is to write a PHP virus that infects files on a server and, then, transfer the developed virus inside an image to the web server.

- **Challenge 5** deals with the downsides of *Docker*, its vulnerabilities, and possible security risks. The first exercise is to exploit a container started with the `-privileged` flag. In a second exercise, the participants have to gain root access to a container that exposes the docker daemon socket. With the acquired privileges they then have to create a new root user on the host system.

- **Challenge 6** gives an introduction to operating systems and so-called Loadable Kernel Modules (LKMs). The task of the students is to write their own LKM. The module should display "hello world" as well as a string parameter. The written LKM is then also supposed to be made persistent by utilizing the initramfs (initial ram filesystem).

- **Challenge 7** extends its predecessor. In *Rootkit Hiding* the task is to hide the previously written LKM, to prevent the LKM from being detected and removed. This can be achieved by hijacking system calls to, for example, hide files with a certain prefix from `ls`.

- **Challenge 8** further extends its predecessors. The objective is to *call* the *userspace*, from the previously written LKM, and spawn a user-land application, e.g., a shell or a crypto-miner with root rights.

- **Challenge 9** combines some of the previous challenges into one large coherent challenge. This highlights the purpose of the individual exercises, their role in the big picture as well as their interrelationships with each other. The single workflow also serves as a realistic example scenario of how a realistic attack, with all its different steps and stages, could look like.

2.3. Capture The Flag

Besides the common game with physical flags, CTF is a type of cyber security competition and a potential way to improve practical cyber security skills. The term is relevant for this thesis as it describes the practical realization of the gamification (cf. Section 2.1) of the EHL (cf. Section 2.2).

A flag can be considered a simple solution string. The string can, for example, simply be the answer to a question, or be hidden in a secret file. Application areas for questions and challenges range from cryptography over IT forensics to reverse engineering and more. After the well-known CTF archive CTFtime [13] one typically differentiates between three types of CTFs, which are Jeopardy, Attack-Defense, and Mixed:

1. **Jeopardy.** This variant provides everyone with a set of a few questions and challenges. Therefore, the goal is to answer questions and/or solve challenges from the mentioned application areas.

2. **Attack-Defense.** In attack-defense, two teams compete against each other. While one defending team tries to patch vulnerable services and secure their machines, the goal of the attacking party is to exploit the security holes and capture flags.

3. **Mixed.** The mixed form is simply a mixture of the two. It may include more or less characteristics from both styles.

The CTF project developed in this thesis implements the Jeopardy variant. As all students have to solve all the challenges on their own, a team-based version as in Attack-Defense or Mixed is not applicable. For the Jeopardy variant, we will use a mixture of questions and challenges for flags. In the context of this work, we differentiate between the following three types of flags:

1. **Theory-flag.** Theory-flags are flags, that are the answer to theory questions. For instance, a theory flag could be `syscall` as the answer to the question which instruction in x86 Assembly is responsible for executing a system call.

2. **Challenge-flag.** Challenge flags are flags that require the completion of a task. For example, a question could be where a specific object is located in memory. Debugging with GDB would result in the address `0x004C5B30`.

3. **End-flag.** End-flags are unique, i.e., randomly generated, challenge-flags, whose submissions confirm the successful, individual, and full completion of a challenge. In order for a flag to function as a Proof of Work (PoW), it has to be ensured that the flag is only accessible after the successful completion of the corresponding exercise. Consequently, this type of flag is not feasible for every challenge. Suitable exercise types are, for example, exercises that feature some form of privilege escalation. As end-flags are PoW and, thus, free the tutors of the EHL from manual correction, they are especially desirable.

2.4. Vulnerabilities

Various security vulnerabilities are implemented in the virtual laboratory environments. Even if it is not essential to understand all security vulnerabilities in detail, it is necessary to understand the basic ideas, as they are relevant for certain design decisions in the implementation (Chapter 5). Thus, in the following subsections we present buffer overflows in Section 2.4.1, as well as use-after-free in Section 2.4.2.

2.4.1. Buffer Overflows

Buffer overflows are a well-known security problem. Buffer overflows are relevant for this thesis, as they are one of the main security holes in the planned laboratory environments. For more information, we refer the reader to [14, 15].

A buffer overflow occurs when a defined buffer receives more data than it was designed for. This can happen for example if there is no length check on the user input. Especially more hardware near languages like C or C++ without built-in safety precautions are prone to this. With knowledge of the memory layout and calling conventions, an attacker can hijack the program flow through malicious input.

The hijacked program flow can in turn be used to inject and execute potentially malicious code as part of an attack. It is also possible to reuse existing code to bypass evolving security measures with techniques such as Return Oriented Programming (ROP) or return-to-libc as described for example in [16, 17, 18]. ROP is also the technique used in Challenge 2 two of the EHL (cf. Section 2.2). In addition to the different types of payloads, there is also a distinction between stack and heap buffer overflows. These simply differ in whether the occurring buffer overflow happens on the stack or the heap.

Buffer overflows are problematic, as an attacker gains access to the respective machine after successfully executing the attack and obtaining a shell. If the program runs with elevated privileges it can result in complete system control. Furthermore, if the execution is not successful in that manner, it may still lead to a program crash.

Nowadays there are certain security measures. These include, for example, Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), or the use of stack canaries to detect stack manipulation.

To get an idea of the concept a simple example of a classical stack buffer overflow can be seen in Listing 2.1. The example is taken from a paper by Aleph One [19]. This program creates a char array of length 255 filled with A's and calls a function that copies the provided string into a predefined char buffer of size 16. This happens without a prior check of whether the defined buffer has sufficient space available. In Figure 2.1 a replica of the program stack when calling the function "function" can be seen. With this figure as help, we see that by copying the too-large string important following values like the saved frame pointer (SFP) or the return address (RET) are overwritten. In this way, by cleverly filling the buffer, the return address can be overwritten, and thus the program flow can be changed to the attacker's will. By providing shellcode, i.e., code that opens a shell in the remaining buffer and redirecting the program flow to this code, the attacker can obtain a shell.

```
void function(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char large_string[256];
    int i;
    for(i = 0; i < 255; i++) {
        large_string[i] = 'A';
    }
    function(large_string);
}
```

Listing 2.1: Example of buffer overflow vulnerable C program based on [19].

Figure 2.1.: Stack layout when calling "function" [19].

2.4.2. Use-After-Free

Use-after-free is an IT security vulnerability. Besides different variants of buffer overflows (cf. Section 2.4.1) they are one of the security holes in the binary exploitation challenges of EHL. For a more detailed description, we refer the reader to [20, 21].

Use-after-free is based on incorrect memory management and is therefore relevant in C, C++, but also Assembly. Generally speaking, use-after-free occurs when memory is accessed despite it having already been freed.

This can happen for example if memory is freed, but the pointer pointing to it is not cleared. If the freed memory is now reallocated, the old pointer points to new and, therefore, false data. A short example can be seen in Listing 2.2. Here, a char pointer `ptr` is created. Assuming the variable `err` is `true`, we enter the first if-block and set `abrt` to 1, i.e. `true`. Consequently, the second if-block is entered, which results in the use of the already freed pointer `ptr`.

```
char* ptr = (char*)malloc (SIZE);
...
if (err) {
    abrt = 1;
    free(ptr);
}
...
if (abrt) {
    logError("operation aborted before commit", ptr);
}
```

Listing 2.2: Example of use-after-free vulnerable C program based on [20].

Depending on the structure of the remaining program, this can lead to several problems. On the one hand, use-after-free can lead to data corruption. On the other hand, it can result in a program crash. Furthermore, use-after-free can even lead to the execution of arbitrary code and, thus, enable an attacker to gain complete system control.

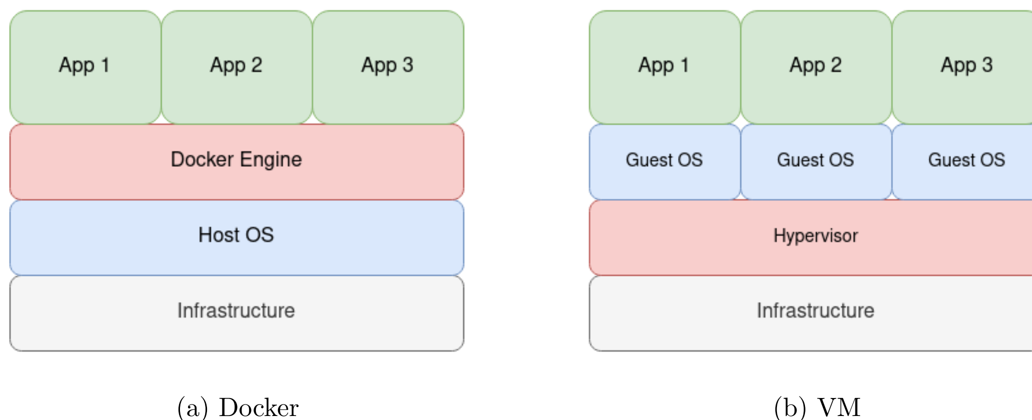
2.5. Tools

For the implementation of the virtual laboratory environments as well as the CTF framework, we use several tools. For this reason, in the following subsections we present a short introduction to Docker in Section 2.5.1, Vagrant in Section 2.5.2, and Flask in Section 2.5.3.

2.5.1. Docker

According to an article provided on the official Docker website [22], Docker is a software solution to address the problem that software may only run on certain systems with the right environment. With Docker, one can encapsulate everything an application needs to run including code, all dependencies, etc., into so-called Docker containers. Isolating applications from their environment by running on top of the Docker engine enables users to run those containers easily everywhere where the Docker engine is installed.

Even though there are similarities between Docker and Virtual Machines (VMs), there are some underlying differences in functionality. While VMs virtualize the system resources, Docker virtualizes the Operating System (OS) of the host. The abstraction shown in Figure 2.2 may help to understand the concept. Sharing the system kernel and not having to include a full copy of an OS comes with the benefit of a much smaller size in the scale of tens of megabytes for Docker containers compared to tens of gigabytes for VMs. Another big advantage for Docker is its much better portability as well as efficiency.



(a) Docker (b) VM

Figure 2.2.: Abstract concept of Docker and VMs [22]

.

2.5.2. Vagrant

Vagrant [23] is a software solution for automatically setting up standardized development environments. Therefore, similar to Docker, it solves the problem of software possibly only running in the right environments with the right dependencies.

Generally, Vagrant and Docker have a few things in common. Thus, for both, you write a configuration file, a Docker- or Vagrant file. Also, on both sides, you choose from a series of base images or boxes as the foundation.

Unlike Docker, however, Vagrant does not use the native containerization function of the OS, but automates the setup of VMs. For this, Vagrant takes advantage of the Command Line Interface (CLI) tools of VirtualBox [24], VMware [25], etc.

One advantage of Vagrant is the standardization of the workflow over different VM providers. Another big advantage is that the configuration with Vagrant is often much easier than with the CLI tools of the providers directly. This significantly improves the workflow for the end users.

2.5.3. Flask

Flask [26] is a very well-known Python web framework, which we use for our CTF framework. It is often used together with the templating Jinja [27], which both are part of the Pallets Project [28]. Flask allows an extremely fast and easy way to get started and create a working website with only a few lines of code. Nevertheless, Flask is also suitable for fairly large and complex projects.

A big advantage of Flask is that it is well-documented with many practical examples. On the official website, several tutorials and template projects can be found. Especially common functionalities like user management or database usage can be adopted from these projects with a little customization. Also, essential things like project layout, a basic application with a start mechanism, or routing are explained in detail with several practical examples. For the implementation of our CTF framework, we will also follow one of the examples [29].

3. Related Work

It is important to place the work in the existing literature context. The reason for this is to know which works already exist in order not to unnecessarily work on things that have already been dealt with or to profit from previous works and to be able to make comparisons. For this reason, in the following sections, we will look at the related work surrounding gamification in Section 3.1) and CTF in Section 3.2.

3.1. Gamification

In the field of gamification, there is the highly cited paper by Deterding et al. [11], in which the authors explore the historical background of gamification. They also differentiate various terms such as game, element, design, or non-game contexts. Finally, they propose the definition of gamification as "the use of game design elements in non-game contexts" [11].

Another highly cited work is by Kapp [30]. In this fundamental work with 14 chapters, the author defines various terms in the context of gamification. He also discusses existing research. The main focus of this work, however, is how gamification can actually be implemented in different areas and ways.

Huotari et al. [31] also published a paper on gamification. However, they rather look at it from an economic or service point of view. The authors want to classify the existing knowledge about gamification in the service and marketing literature. Based on this, they define gamification as "a process of enhancing a service with affordances for gameful experiences in order to support user's overall value creation" [31].

Nah et al. [32] did a literature review on gamification. The focus was on gamification in the context of education. Furthermore, they extracted various game elements based on the review. They also discuss some projects that have already put gamification into practice.

Dicheva et al. [33] did a mapping study to categorize existing work regarding gamification in education. For this purpose, they established different categorization criteria. These categories included game elements, context, implementation, and evaluation.

Sailer [12] deals with the effect of gamification on motivation and performance. The work covers the theoretical foundations of gamification and the state of research. Additionally, two studies on the effect of individual game elements were conducted as part of the book.

Stieglitz et al. [3] published a book where they look at the application of game elements in serious contexts. While the first part covers the basics and definition of gamification, the second part deals with new areas of application of gamification, especially in connection with the work environment or customer contact. In a third part, gamification is further examined in the context of education.

The book by Strahinger and Leyh [34] also deals with the basics and the application of gamification. Again, the authors distinguish between two application domains, business, and education.

The related work on gamification in general is rather theoretical and deals a lot with terminology. Even if we cannot draw practical parallels to our work, we can certainly consider the relevance of gamification in the field of education as positive. Additionally, we can also benefit from useful theoretical background information about concrete game elements.

3.2. Capture The Flag

Looking more specifically at the field of IT security and gamification, there are several papers related to CTF events (cf. Section 2.3). Since these are more related to our work than work on gamification in general, we will look at them in more detail and compare them with our approaches.

Leune et al. [4] provide a study examining the effectiveness of CTFs. The authors did this by interviewing students before and after participating in CTF events. In a study of 24 participants, the authors evaluated the increase in self-confidence, enjoyment during the event, the increase in practical skills, and the reinforcement of theoretical background knowledge. A survey before and after the event resulted in a positive development in all areas. The increase in enjoyment was particularly noticeable. The paper is more concerned with the evaluation of the effectiveness of CTF events rather than the actual technical implementation. Nevertheless, it is relevant to this paper. Answering the paper's original question of whether CTFs improve the effectiveness of cybersecurity education by stating that "CTF exercises increase the effectiveness of cybersecurity education, provided that they are designed appropriately" [4] supports the use of CTFs.

Beltrán et al. [5] describe their experiences in using CTF events in undergraduate computer security labs. Contents include the implementation of the gamified laboratory environment as well as the achieved learning outcomes. In contrast to our work, Beltrán et al. used Facebook's CTF platform. Also, the laboratory environment for the students, in which the practical tasks are solved, was hosted locally on their own computers. The final evaluation endorsed the new project by showing predominantly positive results in terms of learning behavior and enjoyment of the new learning approach. The paper is considerably similar in its approaches to this work. However, they use the Facebook CTF platform, which is unsuitable for our use-case due to the lack of support for randomized flags as well as the direct integration of virtual laboratory environments. Nevertheless, the paper is a good reference point and also confirms the usefulness of CTFs.

The paper [35] deals with Automatic Problem Generation (APG) in CTFs. By slightly changing various parameters in CTF challenges, flag exchange between participants can be prevented and even attempts at cheating can be detected. More complex changes to the challenges should further maintain the essence of the challenges, but still require slightly different solution steps. This should allow the repetition of the CTF for practice purposes. Within their work, they created detailed statistics about cheating, which is an important aspect for the development of our virtual laboratory environments, too. We

want to eliminate cheating by providing randomized flags. Accordingly, this paper is an interesting reference paper. However, in our project, we do not change challenges but the flag obtained at the end whenever possible.

Vykopal et al. [36] explain their experiences with CTFs in terms of benefits and pitfalls when using CTFs in universities. The authors conducted a study with 37 participants on their CTF. Part of the report was the performance of the participants, the usefulness of hints, the flag exchange among participants, and finally the participants' opinion about the CTF. Besides some useful concrete tips for both CTF instructors as well as CTF framework developers, the authors conclude that CTFs are overall a useful alternative to, e.g., homework. In fact, the majority of respondents said that they prefer CTFs to traditional homework. In addition, two plugins for the CTF platform CTFd [37] were published within the context of the paper. Many of the tips given in the paper are related to logging and thereby analysis. Since our CTF project is based on the already existing and fine-tuned challenges of the EHL, we see sophisticated logging and analysis tools as not essential for our purposes. Furthermore, a lot of emphasis has been put on challenge descriptions and hints. We will also implement the possibility for potential flag hints and enable the well-structured presentation of information texts. Interesting is also the tip to put emphasis on predefined rules and procedures for the CTF. This could, e.g., be that the general exchange about solution approaches is indeed desirable. However, the blunt exchange of flags and thus cheating is not. We can include hints and rules of this kind in the information texts.

Ford et al. [6] developed another CTF project. Unique about their project was that it was an unplugged project, which means that all challenges can be solved on paper alone. Students will, nevertheless, be familiarized with tools such as *nmap* or *wireshark*, but only via pre-printed output of these tools. This way, the participants are guided during their first experiences with these tools and only have to analyze the output. According to Ford, many computer science students do not feel qualified enough to participate in CTF events. The consequence of this is that students do not even participate. The goal of the unplugged feature is to overcome this fear by creating CTF challenges that require little to no prior knowledge as well as no complex IT infrastructure. A survey of the 36 participating students after completing the tasks confirmed the effectiveness of the project by showing that a significant increase in knowledge and self-confidence of the students to participate in future CTF events or cybersecurity competitions was achieved. The major part of the paper deals with the concrete formulation of the exercises. Thus, the focus here is not on the technical implementation of CTF laboratories again. Nevertheless, the exercises inspire possible flags for the CTF project in this thesis. The evaluation at the end of the event through a surveying of the participants once again underlines the effectiveness of CTFs.

In his paper "Lowering the Barriers to Capture The Flag Administration and Participation", Chung [7] addresses the problem of complex CTF administration and a daunting learning curve for newcomers. This problem needs to be solved in order to make CTFs interesting for a larger audience and to further establish them in education or as a recruitment tool in professional life. The main problem he describes is the lack of a basic structure for the creation and organization of CTFs. The lack of defined tools, formats for exercises, and rules for the game leads to a great variety of CTF events. Additionally, the widely varying differences in scope and difficulty make it particularly difficult for newcomers to get started. As a solution to push things in the right direction, he suggests the CTF framework CTFd [37] and gives a general overview of the functionality of the open-source project, which simplifies the administration of CTF challenges. As mentioned, we want to closely integrate our own laboratory environments into the CTF framework. Due to the lack of support for randomized flags and the direct integration of virtual laboratory

environments in CTFd [37], we develop our own CTF framework with a strong focus on extensibility for possible future laboratory environments. However, we will orient ourselves on the design of CTFd [37].

Finally, after analyzing different related work on CTFs we can conclude that several papers encourage the use of CTFs by confirming their effectiveness to improve cybersecurity education. Even though most of the conducted evaluations are often rather limited in scope, which justifies doubting the significance of the improvements, a clear tendency towards a positive impact can be observed. Furthermore, by looking at related work, some future works for our project can be identified. This is, for example, the integration of not only randomized flags, but also randomized challenges. This would make cheating significantly more difficult. Moreover, even though, for organizational reasons, we can unfortunately not perform an evaluation in the form of surveying the participants within this work, the numerous evaluations in the other papers suggest a possible later evaluation of our project.

4. Architecture of CTF Project

In this chapter, we present the architecture of the CTF project. As already mentioned in the introduction (cf. Chapter 1) the main goal of this thesis is the improvement of EHL. The improvement of EHL is based on the idea of gamification (cf. Section 2.1) and is accomplished by the development of a CTF project (cf. Section 2.3).

The remaining chapter is structured as follows: To work out the problem and thus the possible improvements we start with a problem statement in Section 4.1. Based on this in Section 4.2 we present a use-case-scenario for the new aspired workflow of EHL. In accordance to the aspired workflow, we continue by defining the concrete requirements for the single components of our CTF project in Section 4.3. Finally, we present the designed structure for the virtual laboratory environments in Section 4.4 as well as the structure of the web framework in Section 4.5.

4.1. Problem Statement

Even though there is generally nothing wrong with a classical teaching approach, we have demonstrated in Section 2.1 and Section 3.1 , that gamification is a legitimate alternative. For this reason, we embed *all* challenges of EHL into a CTF web framework featuring various game elements and statistics to motivate and engage the participants.

Furthermore, we also want to improve the actual workflow for solving the challenges, by embedding the exercises into virtual laboratory environments. As presented in Section 2.2, EHL is quite an extensive module. The development of a virtual laboratory environment for each of the challenges would exceed the scope of this work.

In Section 2.3, we have already introduced the different flag-types, that we use in our work. As we want to include end-flags, we develop our virtual laboratory environments for challenge 2 and 3, as they are particularly suitable because of their privilege escalation. Nevertheless, the development of virtual laboratory environments for other challenges is feasible and possible future work.

Challenge 2 and 3 have already been briefly discussed in Section 2.2. In these challenges the students have to perform different versions of buffer overflows, which we described in Section 2.4.1, as well as use-after-free attack, which we described in Section 2.4.2. In order to identify possible improvements, in the following list, we analyze the workflow of these challenges:

1. As a first step, the students access the exercise instructions via the online platform WueCampus [38] of the Julius-Maximilians-University of Wuerzburg or on-site at the university.
2. Next, the participants have to perform a few reasonably time-consuming setup tasks. Since binary exploits are relatively error-prone, this includes, on the one hand, setting up a VM with a special Ubuntu version, and, on the other hand, the acquisition of resources as well as the installation of tools needed for the exercises.
3. In a third step, the actual exercises get solved locally on the participant's computers. Therefore, there is no actual execution of the developed exploits against a target.
4. As a final step, the participants submit their solutions, which then have to be verified manually one by one of the tutors of EHL.

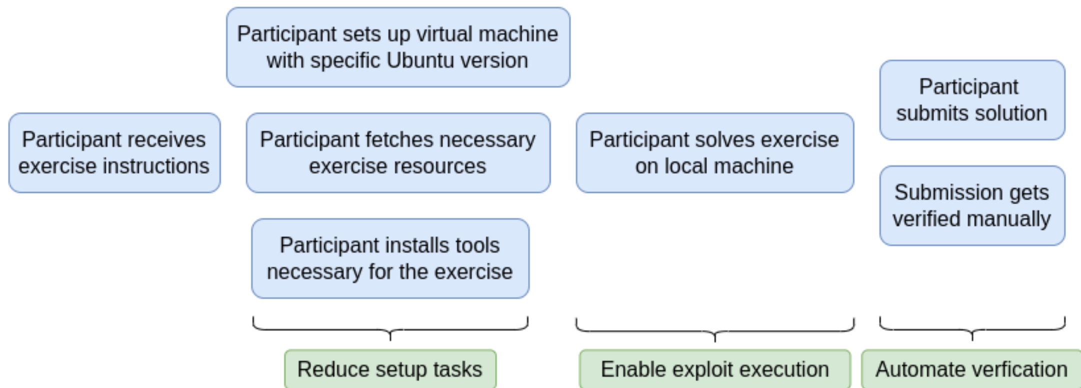


Figure 4.1.: Current workflow of a binary exploitation exercise.

Looking at the current workflow, especially three aspects for improvement can be identified. Those are, first up, time-consuming setup tasks, second, no actual exploit execution against a target, and third, the manual verification of submissions.

4.2. Use-Case Scenario

In Section 4.1, we already described the current workflow of a binary exploitation exercise. In this section, we now present an improved, and with our project aspired, use-case scenario for a binary exploitation exercise.

In the following, we assume that the CTF framework and all virtual laboratory environments are started and ready. The below list together with Figure 4.2 illustrates the aspired use-case scenario with a representative user Bob:

1. Bob starts up his local Personal Computer (PC). Then he accesses the CTF framework with a browser of choice and logs in with his matriculation number.
2. After reviewing the recent statistics, Bob opens the challenges tab, selects a challenge, and reads through the information text. Through the information text, Bob learns what the challenge is about, that he has to access a secret file on a target machine he has normal user access to with given credentials. Of course, he will also find the corresponding Internet Protocol (IP) address and port of his target. Further, Bob will learn that his target machine contains a vulnerable program that can be exploited through some sort of binary exploitation.

Additionally, Bob gets to know how to quickly set up, start, and shut down a CTF machine, which comes prepacked with all the necessary resources and tools right from the start, on his local PC.

3. Within a few minutes Bob easily sets up and starts his local CTF machine with copy-pasteable commands and is immediately provided with all the necessary resources and tools for the challenge.

4. Operating from his CTF machine, Bob accesses his target via Secure Shell (SSH) as a normal user to get an overview of his target.

5. Inspecting his victim, Bob discovers the vulnerable program as well as the `/secret_files` folder, which contains the target file with a randomly generated flag. The flag is not accessible for Bob, as he is logged in as a normal user and does not have root privileges.

6. Bob returns to his CTF machine. As he does not have to deal with setup tasks, he can immediately focus on developing his exploit.

7. After successful implementation, Bob transfers his exploit to his target via Secure Copy (SCP).

8. Bob accesses the target machine again via SSH to execute his exploit on the vulnerable program. If the exploit was successful, Bob receives a root shell.

9. With the newly acquired privileges, Bob searches the `/secret_files` directory to access the secret file containing the end-flag.

10. Bob validates the flag in the CTF framework and receives points accordingly. As the end-flag is randomly generated, the submission is PoW and, therefore, the developed exploit does not need to be manually verified by the tutors of EHL.

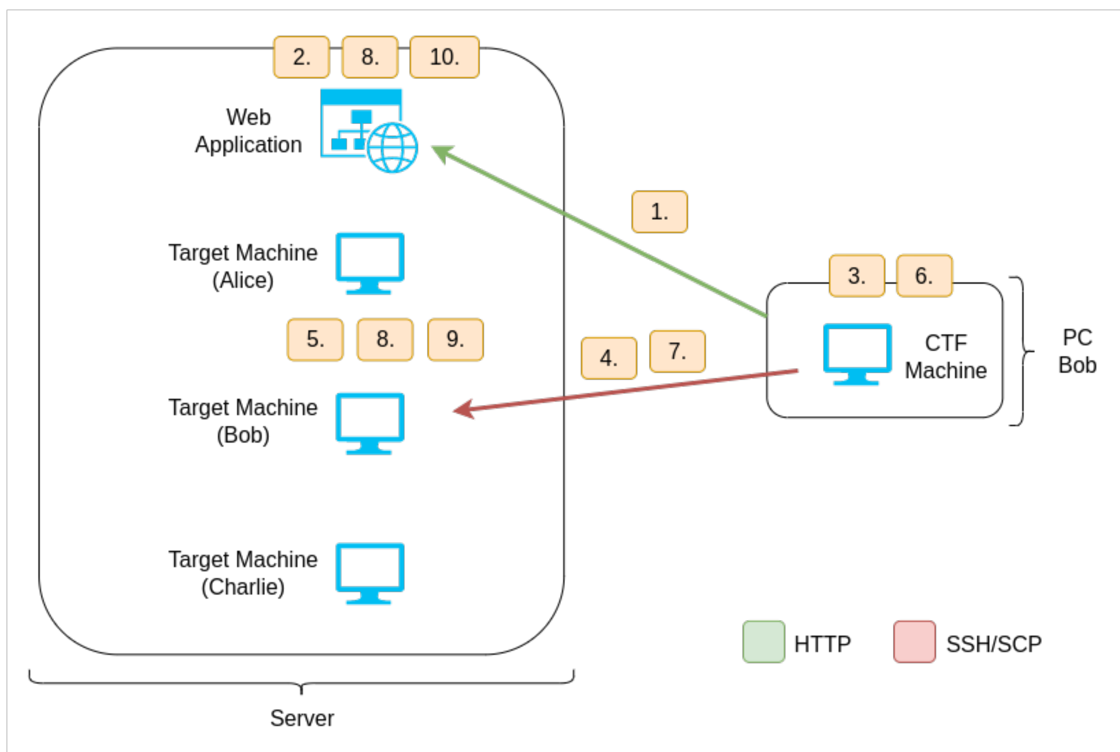


Figure 4.2.: Aspired workflow of a binary exploitation exercise.

In summary, with the described aspired use-case and, thus, ultimately our project, we aim to achieve the following improvements:

a) We motivate the participants through game elements and statistics in our CTF framework.

b) We engage the participants through our virtual laboratory environments, which enable the actual execution of the developed exploits against a virtual target.

c) We increase efficiency by providing the participants with CTF machines that come prepacked with all the necessary resources and tools right from the start.

d) We relieve the tutors of EHL by including randomly generated end-flags whose submissions are proof-of-work and, thereby, replace the manual verification of every submitted exploit.

4.3. Requirement Analysis

In Section 4.2 we already presented the aspired use-case-scenario. To achieve this workflow, we consequently have to develop two components, the virtual laboratory environments as well as the CTF framework.

To concretize those components, in this section we define the exact requirements, which have to be fulfilled. Therefore, in the following sections, we define the exact requirements for the laboratory environments in Section 4.3.1 and the requirements for the CTF framework in Section 4.3.2.

4.3.1. Virtual Laboratory Environments

In Section 4.1 and Section 4.2, we described the problem of the current workflow of the binary exploitation challenges as well as the aspired workflow. Looking at the aspired workflow we can see that we have to develop target- and CTF machines.

The target machines represent virtual victims and enable the participants of EHL to actually execute their developed exploits. Further, they also enable the inclusion of randomly generated end-flags, which will relieve the tutors of EHL from the manual correction of submitted exploits. The CTF machines serve as a local development and test environment. As these already feature all the necessary resources and tools, they relieve the participants of time-consuming setup tasks.

We ultimately implement the "machines" using Docker containers. For this reason, in the following, we refer to the target- and CTF machines as *Target Containers* and *CTF Containers*.

In Chapter 1, we have already mentioned that one reason for using CTF Containers and thus a dual-container setup is to make sure that a locally developed exploit works in the Target Container. This, however, is more critical than that simple explanation suggests.

The exercises in challenge 2 and 3 are different versions of buffer overflow attacks (cf. Section 2.4.1) as well as a use-after-free attack (cf. Section 2.4.2). Those attacks operate on a very low level and rely on analyzing specific memory addresses. The smallest changes in the environment can cause memory locations to shift and, therefore, an attack to fail. With our CTF Containers, we have to replicate the Target Container environment as accurately as possible to ensure that program addresses remain absolutely identical and that a locally developed exploit works in the Target Container.

Assuming an attack scenario in which a victim with a certain vulnerability has been identified, the environment for the exploit development gets set up based on the victim's machine. Accordingly in the following, we first define the Target Container requirements and then the requirements for the CTF Containers.

Target Containers:

(a) **Enabling of attack.** The most important requirement for the Target Containers is that the execution of the attack is possible.

(b) **Enabling of privilege escalation.** Furthermore, it is essential that the attack not only works but also has the desired effect, the privilege escalation. This is essential, as only in this way the flag can be evaluated as proof of work.

(c) **Provision of unique flags.** An important aspect of this project is the integration of unique flags for the individual participants. This contributes to the fact that the flags can be considered as PoW and, therefore, relieves the tutors from manual verification of submitted exploits. Furthermore, this prevents the exchange of flags between the participants.

(d) **Preserving of host system security.** Of course, it is also important to consider IT security. Consequently, another important aspect is not to create any security vulnerabilities through the Target Containers.

(e) **(Optional) Provision of realistic user experience.** Additionally, it is desirable that the Target Containers are somewhat realistic and appear like an actual victim machine. However, as this is not essential for the functionality of the container itself, this is optional.

CTF Containers:

(a) **Enabling the attack.** Here, too, it is self-evident that the exploit has to be possible. Unlike in the Target Containers, this is sufficient and privilege escalation is not necessary.

(b) **Replication of Target Containers environment** In order for the exploit developed locally to also work in the Target Containers, it is important that the same conditions are met as in the Target Containers. Otherwise, address shifts may occur and the exploit will fail in the Target Container.

(c) **Provision of necessary resources.** Providing the necessary resources is similar to mirroring the environment of the Target Containers in the way that it ensures that the locally developed exploit works in the Target Containers, by preventing the use of incorrect versions of the vulnerable program. It also saves the step of acquiring the correct version and thus improves the workflow and efficiency.

(d) **Preserving of host system security.** Similar to the Target Containers, IT security is also important for the CTF Containers. Hence, we have to make sure to not create any security vulnerabilities through the CTF Containers, that could compromise the host machine.

(e) **(Optional) Provision of necessary tools.** Providing the participants with the needed tools reduces setup tasks and therefore further improves the workflow and efficiency of solving the exercises. This way the focus is entirely on learning the concepts of IT security that are relevant to the exercise.

(f) **(Optional) Provision of realistic user experience.** In contrast to the Target Containers, the focus here is on making working in the containers as similar as possible to working on a normal computer. Students should notice as little as possible that they are working in a container and especially should not feel restricted by it.

4.3.2. CTF Framework

Our CTF framework is the center of our project. It embeds every challenge of EHL either in the form of a questionnaire or additionally through a virtual laboratory environment into a single unit. Therefore, the core functionality of our CTF framework is hosting and administrating the challenges of EHL as well as flag management.

Besides this, we also want to integrate the automatic setup, startup, and shutdown of the virtual laboratory environments into our CTF framework to enable easy administration of our CTF project. Another already mentioned aspect is, that our CTF framework should also further increase the attractiveness of the challenges and motivate the participants through various gamification elements.

Overall, we focus on simplicity with our CTF framework, as the focus should remain on the actual challenges and, thus, the content of EHL. We, therefore, define the following requirements for our CTF framework:

(a) **Login of participants.** As the system will be mainly used locally at the university, we do not use any complex user registration for our CTF framework. All participating students are stored in a `.csv` file before starting the CTF framework. The students then log in simply by entering their matriculation number, which is sufficient for identifying the students and processing incoming requests. A detailed justification for this design decision from an IT security point of view is given in Section 6.1.2.

(b) **Display of recent statistics.** In order to promote the game characteristics and motivate the participants, we include statistics on the progress of all users. This includes the display of the users' current points, a line graph, as well as a leaderboard.

(c) **Challenge overview.** Of course, the individual challenges have to be selected somehow. Therefore we include a challenge overview.

(d) **Display of individual challenges.** The main elements of the CTF framework are the pages for the individual challenges. These feature an Hypertext Markup Language (HTML) formatted information text and, of course, the flags. Flags consist of a flag description/question, an input field that displays the answer format, a submit button, and a hint button to display an optional hint.

(e) **Integration of laboratory environments.** Obviously, the laboratory environments are dependent on the CTF framework as they have to feature the randomly generated flags. Thus, integrating the virtual laboratory environments into the CTF framework is a good idea anyway. Further, this way the administration of the CTF project is much easier as a single command is sufficient to start the whole CTF project.

(f) **(Optional) Visually appealing design.** As with any application, a visually appealing design is desirable to improve the user experience. Additionally, small features, such as dynamically displaying completed badges for challenges and flags, rather than, e.g., simply removing them after completion, are desirable. However, as this is not essential to the functionality of the CTF framework, this is considered optional.

4.4. Structure of Virtual Laboratory Environments

In Section 4.3.1, we have defined the exact requirements for Target- and CTF Containers and, thus, the virtual laboratory environments. In this section, we now present the actual structure.

A model of the virtual lab environments can be seen in Figure 4.3. Since this section is about the structure, we refer to the already described use-case in Section 4.2 for the usage.

The structure is mainly as planned. The figure shows the server as well as three different clients. While the clients all start their local CTF Container, the server starts the CTF framework, whose structure is described in more detail in Section 4.5, as well as a Target Container for each client. The communication between the client's PCs and the CTF framework works via HTTP. As in our attack scenario the students already have normal user access to the Target Containers the students access the Target Containers via SSH and SCP.

Other than planned, it can be further be seen that the Target Containers run within a VM. This is due to the fact that the security feature ASLR affects the kernel and cannot be disabled in the containers directly. As in the CTF containers the exploit only has to be developed and work in GDB the problem with ASLR can be solved differently. Consequently, the CTF Containers can run directly on the host machines and do not require a VM. This problem as well as the exact implementation is described in more detail in Section 5.1.

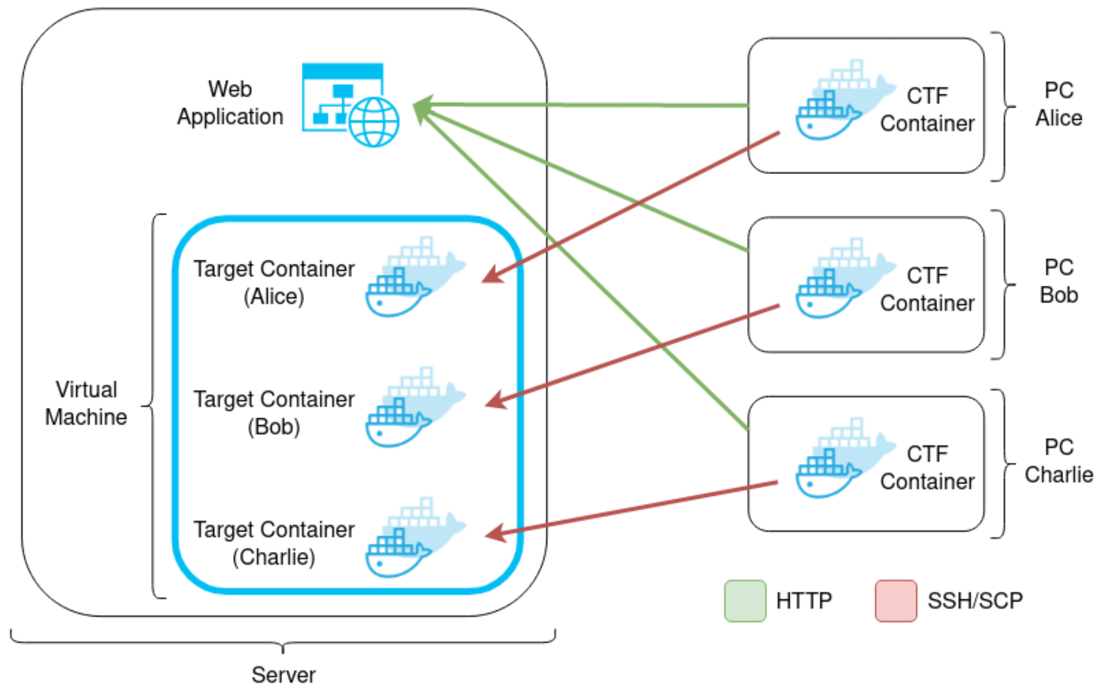


Figure 4.3.: Structure of Virtual Laboratory Environments.

Looking at Figure 4.3, it can be conclude, that the Target Containers respectively the server-side of the virtual laboratory consists of three components. These are as follows:

(a) Target Container Docker image.

(b) Target Container startup mechanism.

(c) VM startup mechanism.

Figure 4.3 also implies the components of the CTF Containers respectively the client-side. The CTF Containers can be partitioned into the following two components:

(a) CTF Container Docker image.

(b) CTF Container startup mechanism.

The concrete implementation of the mentioned components for the Target Containers as well as for the CTF Containers is presented in Section 5.1.

4.5. Structure of CTF Framework

In Section 4.3.2, we have already defined the exact requirements for the CTF framework. in this section we now describe the actual structure. For our CTF framework we have a structure typical for web applications. The general structure of our CTF framework can be seen in Figure 4.4.

The model shows an example client, who accesses the web interface via HTTP. The request gets sent to the corresponding endpoint, which then queries the controller. Depending on the request, challenge logic is queried by the controller. Also important is, that all components, i.e., URL endpoints, controller, and challenge logic can access the database. Finally, the controller sends the answers back to the URL endpoints and eventually to the web interface. A detailed presentation of the concrete implementation of the CTF framework can be found in Section 5.2.

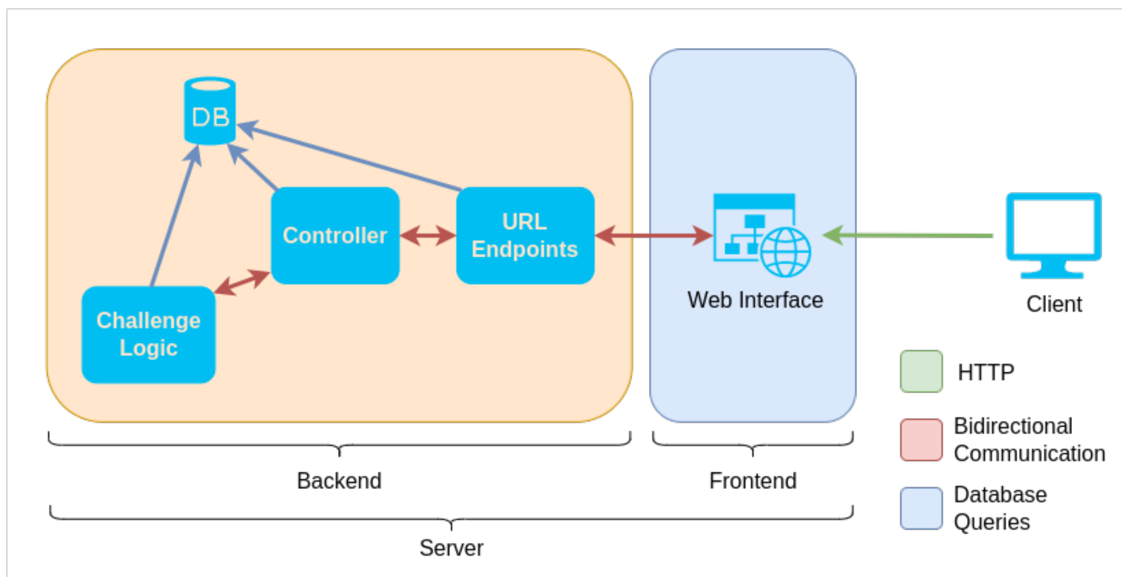


Figure 4.4.: Structure of CTF framework.

Within the scope of this project we also develop a Graphical User Interface (GUI) prototype. This prototype can be found in Appendix A. The final result for the CTF framework is close to the developed GUI Prototype. Only minor changes were made. The most relevant snapshots of the CTF framework can be seen in Figure 4.5, Figure 4.6, and Figure 4.7. Some additional Snapshots can be found in Appendix B.

Figure 4.5 shows the statistics page with a graph, a leaderboard, and the score of the logged-in user. Also, an additional welcome message after login is displayed. Figure Figure 4.6 and Figure 4.7 show a challenge page. On the one hand, one can see the HTML formatted information texts with copy-pasteable commands, as well as dynamic information like port numbers. Also, a few exemplary flags can be seen. Overall, it is also noticeable that some emphasis is placed on a colorful and visually appealing design.

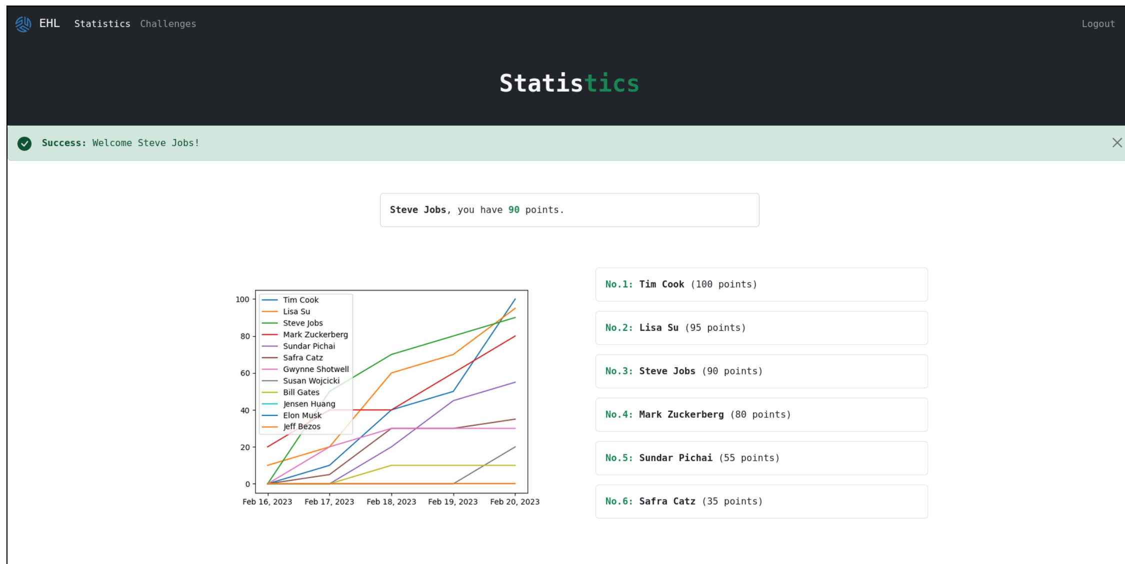


Figure 4.5.: GUI - Statistics page.

EHL Statistics Challenges Logout

Setup

1. Download the heap-buffer-overflow.zip file from the WueCampus course room and unzip it to a location of your choice. Open up a terminal and change directory to the extracted folder.
2. Build the Docker image "heap_buffer_overflow_image" with the following command:

```
docker build -t heap_buffer_overflow_image .
```

Then start the container "heap_buffer_overflow" from the image "heap_buffer_overflow_image" with the following command:

```
docker run \
    --name heap_buffer_overflow \
    --security-opt seccomp=unconfined \
    -v "$PWD/code_injection:/home/bob/ci" \
    -v "$PWD/code_reuse:/home/bob/cr" \
    -it heap_buffer_overflow_image \
    bash
```

(Hint: The name of the image and the container can of course be changed to your will)

3. To exit the container hit **Ctrl+d** or enter **exit** into the terminal.
4. From now on the container can be started from anywhere with the following command:

```
docker start -ai heap_buffer_overflow
```

Hints and Tipps

1. The container should already provide most of the necessary tools. If you still need to install more tools, make sure to update the package lists with **sudo apt-get update** before installing the tool with **sudo apt-get install <package_name>**. The root password for bob is simply **bob**.
2. To work efficiently in a container with several windows open at the same time, you can use tools like **tmux**, **screen** or **terminator** etc. Alternatively, you can use the window in the container for gdb/pwndbg and edit e.g. the file exploit.c from

Figure 4.6.: GUI - Challenge page.

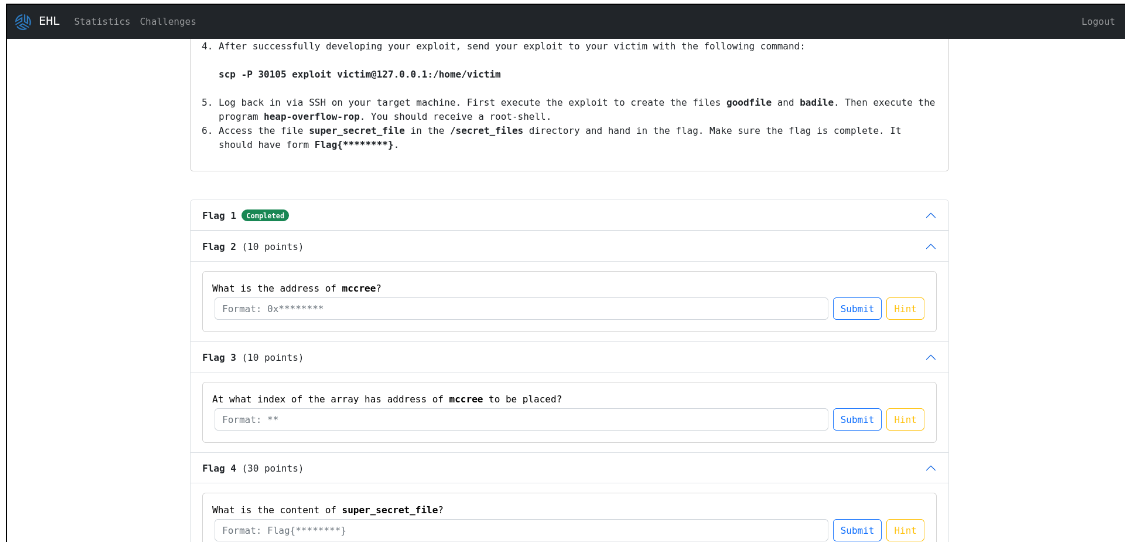


Figure 4.7.: GUI - Challenge page.

5. Implementation of CTF Project

In this chapter, we present the implementation of the CTF project. This includes on the one hand the implementation of the virtual laboratory environments in Section 5.1, and on the other hand the implementation of the web framework in Section 5.2. Finally, we discuss the integration of the remaining challenges in Section 5.3.

5.1. Implementation of Virtual Laboratory Environments

In this section, we present the implementation of the virtual laboratory environments. The implementation of the virtual laboratory environments, excluding resources needed in the containers, includes a total of 320 Lines Of Code (LOC) for both *Heap Buffer Overflow* and *Hybrid Exploits and Use-After-Free*. Consequently, each challenge is realized with about 160 LOC. About 120 lines of these are needed for the Target Containers respectively the server-side, and about 40 for the CTF Containers respectively the client-side.

The laboratory environments for the exercises in *Heap Buffer Overflow* and *Hybrid Exploits and Use-After-Free* are analogous except for the different vulnerable programs. For this reason, we use the *Heap Buffer Overflow* virtual laboratory environment as an example.

Consequently, in the following sections, we deal with the implementation of the virtual laboratory environment for *Heap Buffer Overflow*. We start with the explanation of the server-side, the implementation of the Target Containers in Section 5.1.1 and, then, continue with the implementation of the client-side, the CTF Containers in Section 5.1.2.

5.1.1. Implementation of Target Containers

In this section, we deal with the server-side and, therefore, the development of the Target Containers. As mentioned, the Target Containers respectively the server-side include a total of about 120 LOC.

In Section 4.4, we have already defined the three components of the Target Containers, which are a Dockerfile, a Vagrantfile, as well as a startup mechanism for the Target Containers. In this section, we present the actual implementation of these components. For the implementation, we utilize Docker (cf. Section 2.5.1) and Vagrant (cf. Section 2.5.2) and Bash.

The three mentioned components can all be found in Appendix C.0.1. They are strongly dependent on each other. Therefore, after a short overview of the implementation of the

Target Containers in Section 5.1.1.1, we structure this section as follows: We assume a minimal docker file, i.e., only the Ubuntu image [39], and a minimal `docker run` command, i.e., the command with no parameters other than the required docker image. Then, in the following sections, which are based on the requirements defined in Section 4.3.1, we construct the final components step by step.

We start with enabling the attack in Section 5.1.1.2. Then, we continue with enabling privilege escalation in Section 5.1.1.3, as well as providing unique flags in Section 5.1.1.4. Next, we take care of preserving the host machine security in Section 5.1.1.5. Finally, we look at providing a realistic user experience in Section 5.1.1.6.

5.1.1.1. Overview

The final result of the implementation of the Target Containers is a folder `lab_environment`. This folder can be easily integrated into the CTF framework, and therefore runs on the server side. Consequently, it is managed by the tutors of EHL. For a user manual, we refer to Appendix D.1. Of course, this folder also exists analogously for *Hybrid Exploits and Use After Free*. The structure is as follows:

```
lab_environment
├── code_injection_flags
│   ├── 2363598
│   │   └── super_secret_file
│   ├── ...
│   └── .gitignore
├── code_reuse_flags
│   ├── 2363598
│   │   └── super_secret_file
│   ├── ...
│   └── .gitignore
├── docker_code_injection
│   ├── Dockerfile
│   └── heap-overflow-mprotect.c
├── docker_code_reuse
│   ├── Dockerfile
│   └── heap-overflow-rop.c
├── initial_startup.sh
├── start_containers.sh
└── Vagrantfile
```

Figure 5.1.: Folder structure of Target Containers.

The files marked in *green* are resources needed in the containers. These are on the one hand the secret files in the directories named after matriculation numbers (e.g. 2363589) and on the other hand the vulnerable programs in the `docker_code_injection` and `docker_code_reuse` containers.

Furthermore, two `.gitignore` files in *gray* can be seen. Those are simply responsible for keeping track of the “empty” flag folders and look as follows:

```
*
!.gitignore
```

Listing 5.1: Special .gitignore file to include empty directories.

The files in *blue* are particularly relevant for implementing the three components defined in Section 4.4. All those files can be found in Appendix C.0.1. While the Dockerfiles are necessary for the creation of the Target Container Docker images, the `start_containers.sh` script implements the startup mechanism for the Target Containers. The `Vagrantfile` and the `initial_startup.sh` script are responsible for automated setup and usage of a VM.

5.1.1.2. Enabling of Attack

As mentioned, the virtual laboratory environments are almost identical for both challenges. However, there are little differences which we will discuss in the following paragraph.

Challenge 2 and 3 of EHL, as well as their exercises, have partially different requirements for deactivated security mechanisms. For example, for the code injection exercise the gcc flag `-z execstack` would be necessary to make both stack and heap executable. This is not necessary for the code reuse exercise. In newer Linux kernels, however, `-z execstack` does not have the desired effect on the heap. Consequently, the vulnerable programs have already been adapted to use `mprotect` to mark the required heap regions as executable.

Ultimately, the aspects to enable the attack itself in a container are the same for all exercises, and we can focus only on challenge 2 again. The aspects are as follows:

(a) Enabling of remote access.

(b) Provision of vulnerable program.

(c) Disabling of security feature ASLR.

Enabling of remote access. With our Target Containers, we are misusing Docker in a certain way. Thus, Docker is, e.g., typically used to run web applications on a specific port. SSH access to a Docker container is usually not necessary and, therefore, not a typical functionality.

A first intuitive approach to enable remote access in a Docker container is to simply download `openssh-server` with the Advanced Packaging Tool (APT) and then activate the service via `systemctl`. `systemctl`, however, is not available in containers without considerable effort.

Fortunately, some websites [40, 41] provide a workaround to enable SSH in a container. With this variant, we start the SSH daemon manually. Also, have the create directory `var/run/sshd` manually.

Furthermore, to clarify to the reader of our Dockerfile (cf. Appendix C.0.2) which service and port the container uses, we add `EXPOSE 22` to indicate SSH. The activation of remote access could thus be implemented with the following lines in the Dockerfile:

```
1 EXPOSE 22
2 RUN mkdir -p /var/run/sshd
3 CMD ["/usr/sbin/sshd","-D"]
```

Listing 5.2: Enabling of remote access inside Docker container.

Additionally, a login password has to be set to enable the SSH login. As we need to create a user later anyway, this is dealt with in Section 5.1.1.3. We also need to forward incoming requests at the server into the respective containers. This can be done with the `-p` parameter of the `docker run` command, which is further covered in Section 5.1.1.4.

Provision of vulnerable program. The provision of the vulnerable program is straightforward. To ensure that the same conditions are met in the CTF Containers, we copy the source code into the Docker image and compile it in the Docker image, rather than directly copying the compiled program. This way we ensure, e.g., which GNU Compiler Collection (GCC) flags the program was compiled with.

Disabling of security feature ASLR. Deactivating ASLR is much more complex than originally planned. Initial project considerations did not take into account that ASLR is a kernel feature and, therefore, cannot simply be disabled with a simple `RUN` command in the Dockerfile.

One possibility is to disable ASLR on the underlying host system. However, this would open a huge security hole as the underlying server including all services would run without ASLR.

The remedy for this problem is to install an intermediate layer by means of a VM and to deactivate ASLR on this layer. This way the Target Containers would run as desired without ASLR, but the actual host server would still be equipped with all security features.

For automating the setup and use of a VM, we utilize a software called Vagrant (cf. Section 2.5.2), which simplifies the creation and administration of VMs. In a Vagrantfile, one specifies the requirements for the desired VM. Using the `vagrant up` command, the VM is created, installed, and started. `vagrant halt` shuts a VM down.

Creating a Vagrantfile is straightforward. For example, the command `Vagrant init ubuntu/focal64` generates a Vagrantfile based on the official Ubuntu 20.04 LTS (Focal Fossa) box. This already contains the most common configuration as a template. With `config.vm.provider` general information like Random Access Memory (RAM) or GUI can be defined. With `config.vm.network` we can easily forward ports into the VM and `config.vm.synced_folder` allows the sharing of folders with the virtual machine. Another essential feature is the automated execution of scripts via `config.vm.provision` after setup or startup.

Thus, we write a Vagrantfile to automate the setup of our VM. The in Section 5.1.1.1 mentioned scripts `initial_startup.sh` and `start_containers.sh` will then take care of post-installation tasks, such as installing Docker and disabling ASLR via `sysctl`, as well as ultimately starting the Target Containers. The complete Vagrantfile can be found in Appendix C.0.1.

5.1.1.3. Enabling of Privilege Escalation

In Section 2.4.1 and Section 2.4.2, we explained buffer overflows and use-after-free, which are the targeted security holes in challenge 2 and 3. Assuming that we already have Target Containers that allow the plain execution of the attack, we need to additionally implement the following points to enable privilege escalation through these attacks:

(a) Creation of unprivileged user.

(b) Giving root privileges to vulnerable program.

Creation of unprivileged user. For setting up a non-privileged user, a template can be found in the Dockerfile best practices [42]. To avoid unnecessary problems, we use this template. With the following commands in the Dockerfile, we create a user `victim` with password `victim`:

```
1 RUN groupadd -r victim && useradd --no-log-init -r -m -g victim victim
2 RUN echo "victim:victim" | chpasswd
```

Listing 5.3: Creation of user inside Dockerfile.

Giving root privileges to vulnerable program. Giving root privileges to the vulnerable program is simply done by setting the Set User ID (SUID) bit via chmod, as expected. Therefore, with RUN chmod 4771 <program> in the Dockerfile, we give our vulnerable program the necessary rights.

We further extend all vulnerable programs of the exercises in challenge 2 and 3 by importing <unistd.h> and adding setuid(0); to the beginning of the main methods. This way we ensure that the received shell has root privileges and ultimately enables privilege escalation. As the participants obviously get root rights by their attack, it is important to think about possible attack vectors. This is examined in Section 6.1.

5.1.1.4. Provision of Unique Flags

In order to provide each container with unique flags, we, of course, need to generate the flags to be provided in some form. This, however, will be done by the CTF framework (cf. Section 5.2). Therefore, to enable the provision of unique flags in the containers, we have to implement the following points:

(a) Definition of predefined folder structure for flags.

(b) Startup of Target Containers with unique flag for each participant.

Definition of predefined folder structure for flags. For our virtual laboratory environments, we predefine that the flags are randomly generated and delivered into code_injection_flags/ and code_reuse_flags/ by the CTF framework in form of the following directory structure:

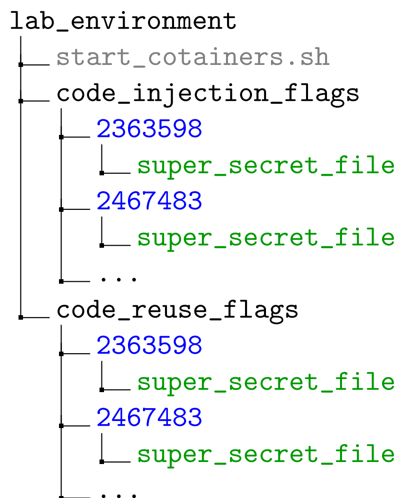


Figure 5.2.: Folder structure predefined by virtual laboratory environments.

Startup of Target Containers with unique flag for each participant. Since the folder structure determines the number of Target Containers needed with the number of folders, we make use of this. As the number of folders and therefore containers is variable, we do not use Docker Compose [43], but instead, write a Bash script to start multiple containers. The basic idea of the start script is to iterate over the provided folders with the uniquely generated flags and host a Target Container for each folder.

The complete start_containers.sh script can be found in Appendix C.0.1. An excerpt of the most important part can be seen in Listing 5.4.

The for-loop, for iterating through the respective folders, is wrapped in a function start_-containers and can, therefore, be reused. The function receives the start port for the

challenge, the flag-folders, as well as the image for the challenge. The starting ports for the challenges are hard-coded and are chosen with the background that EHL has a maximum of 100 participants. An overview of the challenges and their associated ports can be found in Table 5.1. By sorting the flag folders by matriculation number, the port for a student results from the start port of the challenge plus the index of the matriculation number in a sorted list.

Within the function itself we set the permission of the flag folder and flag, such that only root can access the flag in the container. Then, we start the container with `docker run` and the corresponding parameters.

```
start_containers () {
  port=$1
  for folder in $(ls $2 | sort -g); do
    volume="$PWD/$2/$folder"
    chmod 777 "$volume"
    chmod 600 "$volume/"*
    chown root:root "$volume/"*
    echo "Starting container with port $port and folder $volume..."
    docker run -d -p "$port:22" -v "$volume:/secret_files" -t "$3"
    port=$((port+1))
  done
}

start_containers 30000 code_injection_flags code_injection_image
start_containers 30100 code_reuse_flags code_reuse_image
```

Listing 5.4: Excerpt of startup script for Target Containers.

Challenge	Exercise	Ports
Heap Buffer Overflow	Code Injection	30000 - 30099
	Code Reuse	30100 - 30199
Hybrid Exploits and Use-After-Free	Hybrid Exploits	30200 - 30299
	Use-After-Free	30300 - 30399

Table 5.1.: Port range for Heap Buffer Overflow and Hybrid Exploits and Use-After-Free.

Justification of the misappropriation of ls. According to SC2012 of Shellcheck [44], `ls` is for human readable display of folder contents, and the output is not necessarily intended for further use in scripts. Instead, the use of `find` is recommended.

In our case, however, the folder names are limited to combinations of digits, so the use of `ls` is not problematic. An advantage of `ls` is not including the `.gitignore` file, which is necessary to track the empty folders with GIT. Another advantage is the simple possibility to sort the folders by numerical value by piping the output of `ls` to `sort -g`. With `find` we would have a far more complex command to achieve the same result.

5.1.1.5. Preserving of host system security.

In Section 4.3.1, we have defined that we do not want to create any security vulnerabilities through our Target Containers. The IT security aspect of Docker containers is often not about actively taking security measures, but about avoiding common mistakes. For this, we orient ourselves on the OWASP Docker Security Cheat Sheet [45] and the Dockerfile best practices [42]. An analysis of the IT security of virtual laboratory environments can be found in Section 6.1.

5.1.1.6. Provision of Realistic User Experience

In the requirements, we have defined that the Target Containers should be somewhat realistic and engaging. Test runs, however, show that the victim's home directory appears empty. Furthermore, even though it is not necessarily unrealistic for a user to use the `sh` shell, the simple shell impairs the user experience on the victim machine due to, for example, the lack of auto-completion. To fix those problems we implement the following points:

(a) Populating of victim's home directory

(b) Switching of standard shell to `bash`

Both of these points can be fixed without major effort. To make the home directory of the victim less empty and, therefore, the target a bit more realistic, we add some Linux-typical dummy folders like `Desktop`, `Downloads`, `Music`, `Documents`, `Pictures` or `Videos` in the victim's home directory and fill them with dummy files. Also, changing the simple `sh` shell to `bash` is no problem and can be done by adding `usermod -s /bin/bash <user>` to the Dockerfile.

5.1.2. Implementation of CTF Containers

In this section, we present the client-side, the development of the CTF Containers. As mentioned, the CTF Containers respectively the client-side include a total of just about 40 LOC. In Section 4.4, we have already defined the two components of the CTF Containers. These are, on the one hand, the Dockerfile, and, on the other hand, the start mechanism for the CTF Containers. In this section, we present the actual implementation of these components for which we utilize Docker (cf. Section 2.5.1).

The two mentioned components can both be found in Appendix C.0.2. Similar to the components of the Target Containers they are also strongly dependent on each other. Accordingly, in this section, we also proceed in such a way that we assume a minimal docker file, i.e., only the Ubuntu image [39], and a minimal `docker run` command, i.e., the command with no parameters other than the required docker image. Based on these, we construct the final components step by step following the requirements we defined in Section 4.3.1.

This being said, in the following section we, again, start with a general overview in Section 5.1.2.1. Then we begin with the description of enabling the attack in Section 5.1.2.2. After that, we take care of replicating the Target Container environments in Section 5.1.2.3. Next, describe the provision of the necessary resources in Section 5.1.2.4, as well as the required tools in Section 5.1.2.5. In Section 5.1.2.6 we look at preserving the host system security. Finally, we focus on providing a realistic user experience in Section 5.1.2.7.

5.1.2.1. Overview

The result of the development of the CTF Containers is a folder `heap_buffer_overflow`, which is necessary for building the CTF Container Docker image, and, ultimately, starting the CTF Containers. The folder can, e.g., be provided to the participants of EHL via WueCampus [38] as a `.zip` file. Consequently, we are on the client side and the containers are managed by the participants. A user manual can be found in Appendix D.1. It is also worth mentioning that these manuals are embedded into the CTF framework in HTML format with copy-pasteable commands. The folder as well as the mentioned user manuals, of course, also exists analogously for *Hybrid Exploits and Use After Free*. The folder structure is as follows:

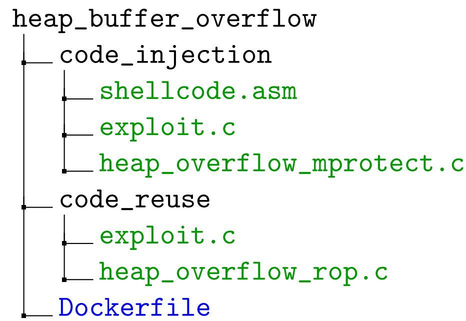


Figure 5.3.: Folder structure of CTF Containers.

The files in *green* are resources needed in the containers. These are the same vulnerable programs as in the target containers as well as the files `exploit.c`, which are templates for the creation of the exploits.

The file `Dockerfile` in *blue* is particularly important as it is responsible for the creation of the CTF Container Docker image, which is one component of the CTF Containers we defined in Section 4.4. Also important is the resulting `docker run` command, which implements the second component we defined in Section 4.4. The command is not part of the folder structure, but both Dockerfile as well as the `docker run` command can be found in Appendix C.0.1.

5.1.2.2. Enabling of Attack

Enabling the attack is in principle easier in the CTF Containers than in the corresponding Target Containers. The aspects to enable the attack itself in a container are as follows:

(a) Provision of vulnerable program.

(b) Disabling of security feature ASLR.

Provision of vulnerable program. The provision of the vulnerable program was originally planned to be implemented by simply utilizing `COPY` in the Dockerfile, however, this results in problems with providing a realistic user experience. Consequently, we include the vulnerable program by mounting folders of the host system into the container. This is explained in more detail in Section 5.1.2.7.

Disabling of security feature ASLR. Since a VM is needed for EHL anyway, one possibility is to simply start the containers in this VM and deactivate ASLR of the VM. However, it is desirable that the CTF Containers also work without an underlying VM.

The difference between the CTF Containers and the Target Containers is that the exploit does not have to run in the container itself, i.e., outside GDB. Consequently, for the development and testing of the exploit, it is sufficient to have no ASLR in GDB.

By default, Docker containers run with a default security profile that disables selected syscalls within the Docker container for security reasons. With `-security-opt seccomp=unconfined` the default security profile can be deactivated. This, among other things, allows GDB's default behavior of disabling ASLR for the debugged process, while keeping the system ASLR activated.

Of course, disabling security features comes with certain security risks. However, as the containers run exclusively locally the security risks are reasonable. The IT security aspect is further discussed in Section 6.1.

Ultimately, the disabling of ASLR for the debugged process inside GDB is possible by extending the `docker run` command as follows:

```
docker run \
    --security-opt seccomp=unconfined \
    -it <image_name> \
    bash
```

Listing 5.5: Enabling of GDB's default behavior to disable ASLR for the debugged process.

5.1.2.3. Replication of Target Container Environment

As mentioned in Section 4.3.1, replicating the environment of the Target Containers is essential to prevent address variations and, thereby, ensure the success of the locally developed exploit in the Target Containers. Therefore, we have to implement the following points:

(a) Adopting of Target Container base image.

(b) Ensuring of identical compilation of vulnerable program.

Adopting of Target Container base image. Docker containers are all based on a base image. Thus, we, of course, adopt the base image of the Target Container. The Target Containers use the `ubuntu:20.04` image [39]. Consequently, we copy this in the CTF Containers.

Ensuring of identical compilation of vulnerable program. We have to ensure that the vulnerable program is compiled in the same way. The vulnerable programs in the Target Container are all compiled with the GCC flag `-static`. This leads to static compilation of the program and thus to significantly more gadgets which is essential for the exercises `code reuse` and `hybrid exploits`. Therefore, it is important to compile in the same way locally. In order to make the participants aware of this fact and not relieve them of too many steps, the vulnerable program is not provided pre-compiled. Instead, an instruction for the correct compilation is included in the information texts provided on the challenge websites.

Elimination of unexpected address fluctuations. A test run with the *code reuse* exercise in challenge 2 runs as planned. Test runs with the three remaining tasks yield problems, contrary to our expectations, and fail. Investigations show that the environment is in principle correctly replicated, i.e., the use of the same Ubuntu base image and the identical compilation on both sides. Also, ASLR is deactivated in all cases. Thus, a failure due to errors in the environments can be excluded.

Crucial for the solution of the problem is the success of *code reuse*. In Section 5.1.2.7, we justify the decision to mount the resources instead of copying them. Thus, when the local CTF Containers are started, two volumes are mounted. One for *code reuse* and one for *code injection* in challenge 2, and analogously one for *hybrid exploits* and *use-after-free* in challenge 3.

The *code reuse* exercise and *hybrid exploits* exercise both use the same vulnerable program `heap-overflow-rop.c`. Examining addresses using GDB, results in minimal address shifts within the containers for challenge 2 and challenge 3. Extensive tests, regarding images, mounted folders, etc. to evaluate the crucial factor for addresses to shift, reveal that the length of the names of the mounted volumes is responsible.

`code_injection`, `hybrid_exploits` and `use_after_free` all are all considerably longer than `code_reuse`. Further tests reveal that using more than twelve letters for the volume names within the container results in addresses being shifted by a few bytes. However, this is sufficient for the failure of the developed exploits in the Target Containers.

For this reason, the names of the mounted volumes within the containers are limited to the first letters of the exercises in Challenge 2 and Challenge 3, i.e., `ci`, `cr`, `heauf` and `uae`. The resulting `docker run` command for correctly mounting the volumes can be found in Section 5.1.2.4.

5.1.2.4. Provision of Necessary Resources

Apart from convenience, providing the resources also ensures the correct version of the vulnerable program and, thus, contributes to replicating the Target Container environment. The original plan was to simply use `COPY` to include the resources in the images and provide the participants with the finished built images.

However, this conflicts with providing a realistic user experience, which is further explained in Section 5.1.2.7. Ultimately, the provision of the necessary resources is, therefore, done by mounting volumes with the following extension of the `docker run` command:

```
1 docker run \
2     -v "$PWD/code_injection:/home/bob/ci" \
3     -v "$PWD/code_reuse:/home/bob/cr" \
4     -itheap_buffer_overflow_image \
5     bash
```

Listing 5.6: Provision of necessary resources inside Docker container.

5.1.2.5. Provision of Necessary Tools

To solve the exercises in challenge 2 and 3, certain tools are needed. The most important is GDB. Furthermore, the use of Pwndbg [46] and the tool ROPgadget [47] is recommended to improve the workflow in GDB. Pwndbg is a GDB plug-in that makes the use of GDB much easier. ROPgadget makes it possible to search for gadgets in binary files. Furthermore, it is useful to provide a basic code editor. Since many Linux distributions come with Vim and Nano preinstalled, we choose those two to allow simple editing of files. In *code injection*, also the assembler NASM is needed. Therefore, to provide all the necessary tools, we have to implement the following points:

(a) Installation of first-party tools.

(b) Installation of third-party tools.

Installation of first-party tools. The installation of GDB, the code editors Vim, Nano, and the assembler NASM via APT is straightforward. The Dockerfile best practices [42] give a an example for using APT. To avoid unnecessary problems follow this guide an install the needed tools with the following command:

```
1 RUN apt-get update && apt-get install -y \
2     <package_name> \
3     && rm -rf /var/lib/apt/lists/*
```

Listing 5.7: Installation of packages inside Docker container.

Installation of third-party tools. The installation of Pwndbg and ROPgadget is a bit more complex. The installation of Pwndbg is done via `git clone` and a `setup.sh` script included in the corresponding repository. This also installs ROPgadget automatically.

In Section 5.1.2.7, we justify the usage of a non-privileged user bob. Therefore, executing the `setup.sh` script, the user is prompted for the `sudo` password. For this reason, the installation of Pwndbg has to be done before changing to the user bob in the Dockerfile.

As a result, the installation itself runs as root, which results in falsely set program paths. So, neither Pwndbg nor ROPgadget work after the installation for the user bob.

Searching for ROPgadget in the container after startup results in ROPgadget being installed, but in the directory `/usr/local/lib/python3.8/dist-packages/bin`. Adding this path to the `PATH` variable in the Dockerfile solves this problem. To solve the problem with Pwndbg not working, we examine the `setup.sh` script. At the end of the script, the following if-statement can be found:

```
if ! grep Pwndbg ~/.gdbinit &>/dev/null; then
    echo "source $PWD/gdbinit.py" >>~/.gdbinit
fi
```

Listing 5.8: Code responsible for setting .gdbinit.

This means that the script writes a `.gdbinit` file in the home directory of the executing user. In our case, this user is root. To fix this issue, we have to analogously create the `.gdbinit` file for bob.

Finally, including the following two lines in our Dockerfile fixes the problem with broken paths, and thereby enables the functioning of Pwndbg and ROPgadget.

```
RUN cp /root/.gdbinit /home/bob/
ENV PATH=/usr/local/lib/python3.8/dist-packages/bin:$PATH
```

Listing 5.9: Commands to fix broken paths after Pwndbg/ROPgadget installation.

5.1.2.6. Preserving of host system security.

In Section 4.3.1, we have defined that we obviously do not want to create any security vulnerabilities through our CTF Containers. As mentioned for the Target Containers, an analysis of the IT security of the virtual laboratory environments can be found in Section 6.1.

5.1.2.7. Provision of Realistic User Experience

In Section 4.3.1, we have defined that we want to have a natural user experience in the CTF Containers. Considering that in most Linux distributions one does not act as root by default, to avoid, for example, the accidental deletion of files, we want to work in the container as a normal user by default.

Further, during test runs, the problem occurs that the workflow is limited to only having one terminal window. Therefore, we additionally have to implement the following points to provide a realistic user experience.

(a) Creation of unprivileged user with `sudo` rights.

(b) Enabling of parallel workflow.

Creation of unprivileged user with sudo rights. For setting up a non-privileged user, we again use the template, which can be found in the Dockerfile best practices [42]. By making a small modification of adding bob to the sudo group, we simultaneously enable the use of sudo for bob.

```
RUN groupadd -r bob && useradd --no-log-init -r -m -g bob -G sudo bob
RUN echo "bob:bob" | chpasswd
```

Listing 5.10: Enabling of sudo in a Docker container.

Enabling of parallel workflow. Enabling multiple terminal windows is slightly more difficult. A simple and recommended solution is to use tools like `Tmux`, `Screen` or `Terminator`. These allow parallel work in a single terminal and, thus, for example, the simultaneous opening of Pwndbg and a code editor. This way, addresses can be efficiently found in Pwndbg and copied into the code editor for the exploit.

However, challenge 2 and 3 can be very challenging. Not only do students need to understand the concepts of each exercise, but they also need to be able to solve them using GDB. Even though Pwndbg is a significant help, the student is still exposed to a lot of information. There is definitely a steep learning curve when using GDB/Pwndbg.

To enable everyone to work efficiently even without the recommended tools, we do not simply use `COPY` in the Dockerfile, but provide the necessary resources via volumes mounted into the container, as explained in Section 5.1.2.4. This makes it possible to simultaneously use Pwndbg within the container and edit the exploit from the host system in a code editor of choice.

5.2. Implementation of CTF Framework

In this section, we present the implementation of the CTF framework. The CTF framework consists of a total of over 1400 LOC. About 1000 lines of these are backend and about 400 are frontend.

To structure this section we distinguish between frontend and backend. Before covering those two, we start with giving an overview of the implementation of the CTF framework in Section 5.2.1. Based on the overview we proceed with the implementation of the frontend in Section 5.2.2 and the implementation of the backend in Section 5.2.3.

5.2.1. Overview

In this section, we give an overview of the implementation. The final result of the implementation of the CTF framework is the root folder `goehl`, whose structure can be seen Figure 5.4. This serves as orientation in the following sections for allocating the explained components.

In gray, the challenge folders can be seen. Thus, each challenge has its own subfolder in `resources`, containing `name.txt`, `info_text`, and `flags.csv`. Optionally, these, additionally, contain a folder `lab_environment`, which corresponds to the virtual laboratory environments developed as explained in Section 5.1.

Furthermore, files in green can be seen. These are `__init__.py` (cf. Section 5.2.3.1), which contains the base application, as well as `auth.py` and `views.py` , which are responsible for the routing (cf. Section 5.2.3.2). `db.py` and `schema.sql` are responsible for the database (cf. Section 5.2.3.3). All these are based on the template project and simply adapted to our needs.

In magenta, you can see the folders `static` and `templates`. These contain the frontend files and are further discussed in Section 5.2.2. Especially interesting are also the blue files and folders. The folder `dataclasses` contains our complex datatypes, which are explained in Section 5.2.3.4. `controller.py` is, as the name suggests, our controller. It contains the base logic of our CTF framework and is further explained in Section 5.2.3.5. Last we have `challenge_creator.py` and the challenges in `challenges`. These are the "abstract" parent class and the implementing child classes. The logic of the challenges is explained in more detail in Section 5.2.3.6 and Section 5.2.3.7.

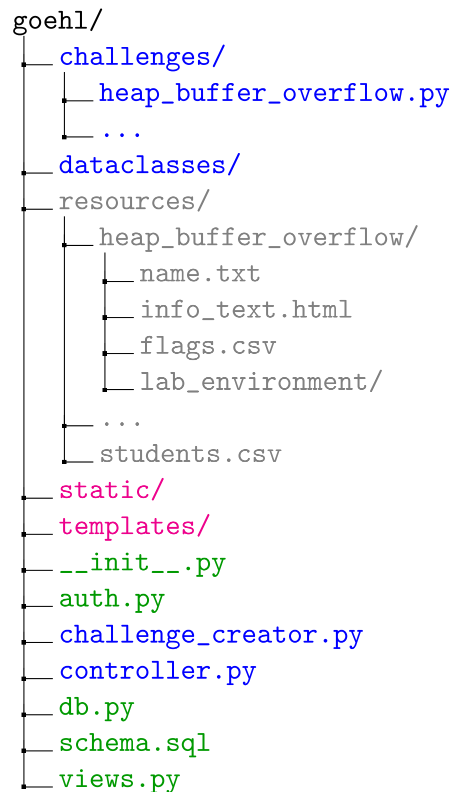


Figure 5.4.: Folder structure of CTF framework.

5.2.2. Implementation of Frontend

In this section, we present the implementation of the frontend. This concerns the directories `static/` and `templates/` in the presented folder structure (cf. Figure 5.4). As mentioned, the frontend consists of more than 400 LOC.

For the frontend, we naturally use HTML. We also use the frontend toolkit Bootstrap 5 [48] for visual appearance. Last, we utilize the templating engine Jinja [27], which is typically used together with Flask (cf. Section 2.5.3).

To structure the presentation of the frontend implementation in a logical way, we divide it into five blocks. Thus, in the following sections, in Section 5.2.2.1, we start with the base template, which is the base for all the other templates, whose implementation is presented in Section 5.2.2.2. In Section 5.2.2.3, we then continue with the provision of the actual page contents coming from the backend. In Section 5.2.2.4, we further explain the backend communication and finally cover certain additional features in Section 5.2.2.5.

5.2.2.1. Base Template

Using the already mentioned templating engine Jinja enables the creation of a base template, which can be extended by other templates. This saves a lot of time by not having to write redundant HTML code and also makes it a lot easier to maintain the code. It, furthermore, provides the advantage of giving the website a uniform look by using recurring elements in each view, like navigation bars or a uniform header section. Thus, we implement the following aspects in the base template:

- **Inclusion of resources.** All resources that are used in several templates are included directly in the base template. This includes Bootstrap5 resources as well as icons that are used.

- **Navigation bar.** We include a navigation bar to allow easy navigation on the CTF framework. The navigation bar is visible on all pages except the login page. It additionally gives the site a consistent theme.
- **Header.** To extend the unified look of the CTF framework, we also define a unified header section in the base template. The other templates fill this header section with their title.
- **Message flashing.** Furthermore, we use message flashing in several places. This applies to after the login as a greeting, as well as in the feedback after the submission of a flag. The messages are integrated directly under the header section. We display the flashed messages in Bootstrap alerts, divided into the categories of success and error.
- **Page content block.** Finally, we define the `page_content` block. This block gets overridden by the other templates. This way, all the other templates are only responsible for their own page content.

5.2.2.2. Login, Statistics, Challenges and Single Challenge Templates

As we have designed a GUI prototype (cf. Appendix A) the implementation of the Login, Statistic, Challenge, and Single Challenge templates is straightforward. The final CTF framework largely corresponds to the prototype.

5.2.2.3. Page Contents

The display of page content is achieved by using Jinja. Jinja not only allows the use of base templates, but also the dynamic creation of HTML pages.

The data needed in the frontend is not transmitted via JavaScript Object Notation (JSON), but can be passed directly as Python objects. This is an additional motivation for the use of dataclasses in the backend, which is further explained in Section 5.2.3.

Jinja further enables the use of Python-based code through different delimiters in the HTML templates. With `{% ... %}` control flow structures like simple for loops or if statements can be included. `{{ ... }}` allows the inclusion of expressions and, thus, data access. For example, by passing a challenge object to the Flask `render_template` method, we dynamically generate the variable number of flags with the following code:

```
1 {% for flag in challenge.flags %}
2 <div class="accordion=item">
3   ...
4 </div>
5 {% endfor %}
```

Listing 5.11: Display of multiple flags in HTML template.

5.2.2.4. Backend Communication

The backend communication is implemented completely without additional JavaScript. Thus, all requests are GET, except two. Those two are the login and the validation of the flags. Both of them are implemented using an HTML form. Additional required parameters for the requests, apart from the direct form input, such as the flag number, are integrated via the HTML input type `hidden`.

5.2.2.5. Additional Features

In order to improve the user experience, we add some additional features by utilizing the already-mentioned delimiters for Jinja. Hereby, we enhance the GUI and display the progress of the participants more clearly. For example, with the following code we automatically close finished flags in the HTML accordion:

```
{% if not flag.is_completed %} show {% endif %}
```

Listing 5.12: Automatic closing of completed flags.

The same approach is used to realize other aspects. In total, we implement the following additional features to improve the user experience:

- Different design of success and error messages
- Display of number of completed flags of a challenge or completed badge if finished.
- Display of points of a flag or completed badge if finished.
- Default display of completed flags in closed state.
- Removing of input field and buttons of already completed flags.

5.2.3. Implementation of Backend

In this section, we present the implementation of the backend. As mentioned, the backend consists of more than 1000 LOC. For the implementation we use Python3 [49], the web framework Flask [26], as well as SQLite3 [50] for the database.

In general, in the whole backend, we use the Python module `logging` to display status messages in the console. Also, to get structure and security into the project, we follow the convention that two underscores (dunder) at the beginning of a function or variable indicate private, one underscore indicates protected and no leading underscores indicate public.

In Figure 5.5 the Unified Modeling Language (UML) diagram of the backend can be seen. It is important to note that this is limited to the classes specifically developed for this CTF framework. Flask-typical functions like `create_app` in `__init__.py` and the modules `auth.py` as well as `views.py`, which are responsible for routing, are not shown. The communication of these with the rest of the backend is solely via the public methods provided by the controller. Furthermore, the database is not part of the diagram. The database is queried by the above-mentioned flask-typical modules as well as all challenge creators and the controller. This being said, the main design considerations are as follows:

(a) **Dataclasses.** Due to the “big size” of the project, we decide to create data classes for `Flag`, `Challenge`, and `Student` to improve the structure of the project.

(b) **Abstract class ChallengeCreator.** To ensure certain methods in the challenges, which the controller can call, we decided to use an abstract parent class. Unfortunately, Python does not natively provide interfaces or abstract classes. A workaround would be the module ABC [51]. However, this module requires the existence of at least one `abstractmethod`. In order to avoid unnecessary overhead by including a dummy method, the design decision remains merely a concept to keep in mind and is not actually implemented in code.

(c) **Singleton class Controller.** The controller is the central control unit of the backend. Since it is only instantiated once, the Singleton design pattern is evident.

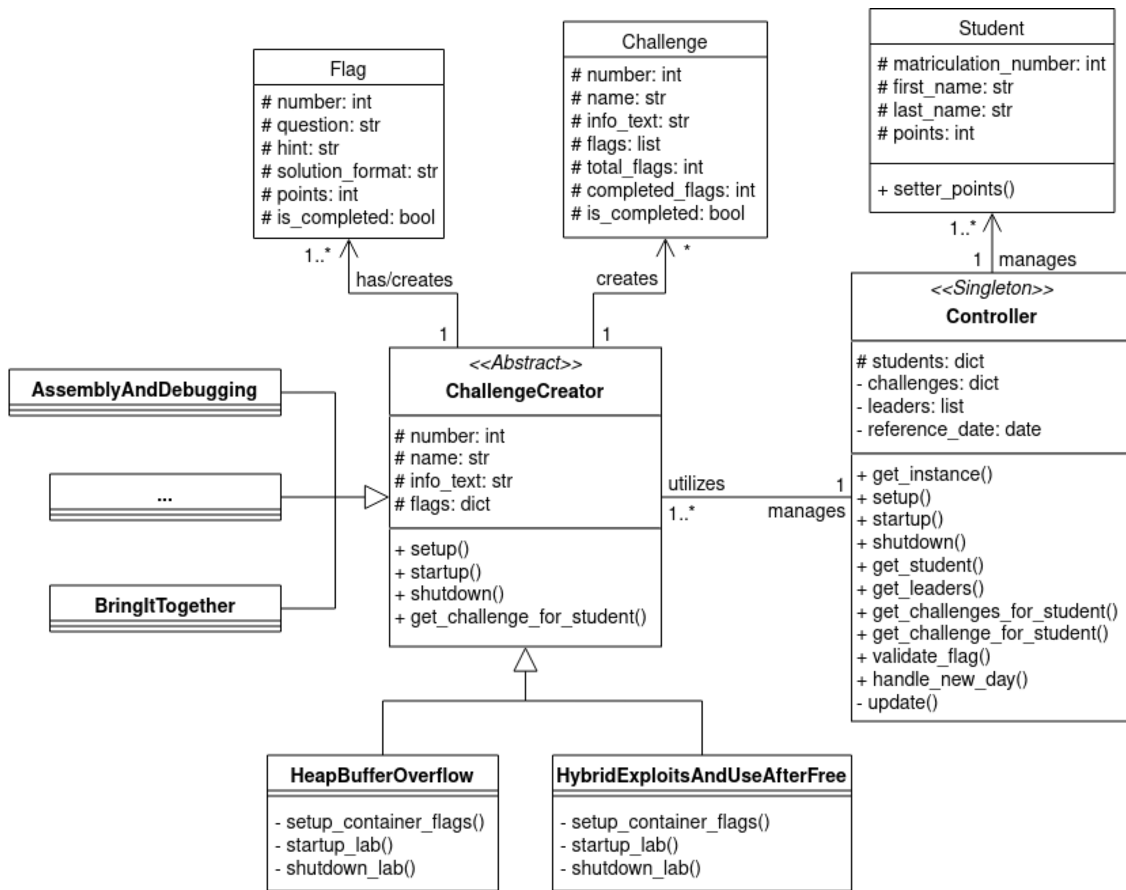


Figure 5.5.: UML diagram of CTF framework.

Partly based on the UML diagram, we structure the rest of this section by dividing it into seven different blocks. Proceeding from small to big, we start with, to our needs adapted, Flask typical parts that are necessary for a functioning CTFframework. Afterward, we deal with the, for our backend, individual parts of the CTF framework.

This being said in the following sections, we first deal with the Flask base application in Section 5.2.3.1. We then continue by explaining routing in Section 5.2.3.2 and the database in Section 5.2.3.3. All those blocks are based on the template project. After that, we present the actual backend. We start by introducing our complex datatypes in Section 5.2.3.4. Second, in Section 5.2.3.5 we explain the base logic, i.e., the controller. Finally, we describe the challenge logic in Section 5.2.3.6 and Section 5.2.3.7.

5.2.3.1. Base Application

As with the general structure, we also follow the Flask templates for the base application, which is located in `__init__.py`. This module contains a method called `create_app`, which returns a `Flask` object. This object represents the base application. In `create_-app`, we further implement the following points:

(a) Application-wide configurations.

(b) Integration of controller setup, startup, and shutdown.

Application wide configurations. The method `create_app` is also the place to make various application-wide configurations. This applies to a possible secret key, the database path, or registering Uniform Resource Locator (URL) endpoints. We further define our logger format in this method.

Integration of controller setup and startup. The core of the CTF framework is the controller (cf. Section 5.2.3.5). Since create_app is the starting point of the CTF framework, the controller has to be integrated here. A first approach to embed the controller methods setup and startup (cf. Section 5.2.3.5), is the Flask before_first_request method. This allows code to be executed once before the first request. However, this is considered deprecated since Flask version 2.2 and will be removed in Flask 2.3. For this reason, we avoid using this method.

The solution to including the controller methods setup and startup is to check for the existence of the database SQL file in the instance/ folder. If the file exists, startup is executed, otherwise setup. Since the setup can take some time, the CTF framework is terminated after the setup with a logger message that the setup is finished and that flask run has to be executed again to start the CTF framework. This separates the setup from the normal operation and simultaneously avoids several problems that would result from the successive call of setup and startup.

Integration of controller shutdown. Another important aspect is the inclusion of the controller method shutdown (cf. Section 5.2.3.5). To implement this aspect, we utilize the Python module atexit. The module atexit allows registering cleanup methods which are called when a program is terminated.

By registering the controller method shutdown in __init__.py, normal python interpreter termination results in the proper call of controller.shutdown.

5.2.3.2. Routing

Overall, we have two Python modules that are responsible for implementing the URL endpoints. These modules are auth.py and views.py. In those two we implement the following aspects:

(a) Authentication of users.

(b) Handling of requests.

Authentication of users. User management and authentication are important aspects. Fortunately, the template project provides the most important functionalities with auth.py. This includes, first, the login and logout of users via the methods login and logout. Second, they provide essential auxiliary functions to protect certain cites against unauthorized access. This includes requiring a valid login for certain sites, which is implemented via login_required and load_logged_in_users. By decorating an URL endpoint with @login_required, before every request we check whether a user is logged in or not. If this is not the case, the user is automatically redirected to the login page.

Handling of requests. Handling the request is done by the methods challenges, statistics, and single_challenge. While challenges and statistics only handle GET requests, single_challenge handles both, GET and POST requests, as single_challenge is responsible for both, the display of the individual challenges and the flag validation.

Unlike in the template project, we want much less program logic in the URL endpoints. For this reason, we transfer most of the logic into the controller. This way we significantly reduce the complexity in the URL endpoints, making them much clearer and easier to work with.

5.2.3.3. Database

The database is implemented with SQLite3 [50]. This is because SQLite is lightweight, easy to set up, and easy to use. The implementation of the database concerns the files schema.sql as well as db.py. With those two we implement the following aspects:

(a) Structure of database.

(b) Integration of database.

Structure of database. In this project, all challenges have their own subfolder in `resources/` containing, among other things, `name.txt`, `info_text.html`, and `flags.csv`. As this information is only read in once at setup and startup and does not have to get accessed or modified during run-time, inserting it into the database would not bring any advantage.

Following this logic, we would do the same with `students.csv`. However, the module `auth.py` mentioned in Section 5.2.3.2 queries a table *user* in the sample project. As modifying `auth.py` to query the controller instead of the database would not bring any advantage, we simply keep a table for the users and populate it with the students in `students.csv` during setup. Ultimately, our database consists of the following three tables:

1. **Student.** This table stores the matriculation number of the students, as well as their first and last names. The table is necessary for the functionality of auth.py.
2. **Flag.** This table stores the challenge number, the flag number, the matriculation number, and the flag value. This allows, on the one hand, to store the uniquely generated flags. On the other hand, it allows storing which flags have already been successfully submitted by setting the flag value to Python's `None`.
3. **Score.** Finally, we have a table score. This table stores the matriculation number as well as the date and the corresponding score. This table is necessary for the creation of statistics.

Integration of database. To include the database in the project, we again rely on the template project. The included module `db.py` provides the most essential functions for the connection and use of the database.

The database can be accessed from anywhere in the project simply by calling `get_db`. Further, the auxiliary functions `init_db` and `init_app`, and `close_db` are included. While `init_db` is used to initialize the database with the tables defined in `schema.sql`, `init_app` is used to register the database with the base application. `close_db` is responsible for properly closing the database connection after a request is completed.

Flask takes care of calling the auxiliary functions with the special Flask object `g`. The object `g` is unique for each request and is used to store data during the lifetime of a request. For example, `g` is used to store a database connection after calling `get_db`. Each subsequent call of `get_db` within the lifetime of the request will now receive the database connection from `g` instead of establishing a new connection.

5.2.3.4. Complex Data Types

The complex datatypes in our project are challenges, flags, and students. With PEP 557 [52], *Dataclasses* came into the standard Python library. Dataclasses allow the simple creation of classes that are mainly intended for storing complex data types and contain little to no logic. Unlike the simple use of tuples, they allow attribute access by keyword as well as default values. Furthermore, functions such as `__init__`, `__repr__` and `__eq__` are generated automatically.

In most CTF frameworks data is sent to the frontend via JSON. However, since our data is passed to Jinja, we pass it directly as Python objects. This results in two use cases for our complex data types flag and student:

(a) **Usage in backend.** Taking `Flag` an example, `Flag` objects in the backend are created during the start of the application and kept within a dictionary in the `ChallengeCreator` classes. They are used to store the data of `flags.csv` in memory and hold the attributes `number`, `question`, `hint`, `solution_format`, and `points`.

(b) **Usage in Jinja.** `Flag` objects for Jinja are created by the `ChallengeCreators` every time the method `get_flags_for_student` is called. Those `Flag` objects are based on the `Flag` objects in the backend and hold additional student-specific information like the attribute `is_completed`.

To avoid writing duplicate code, we combine both use cases by making sort of "hybrid" classes. Thus, we start with normal Dataclasses, which hold all attributes that the frontend needs. Then, we modify the classes by adding default values and setter methods, where needed, to enable seamless usage in the backend, too. Ultimately, the project contains the following three Dataclasses:

- **Challenge.** `Challenge` must be distinguished from the `ChallengeCreator` classes. Thus, `ChallengeCreators` create `Challenge` objects. Contrary to `ChallengeCreators`, `Challenge` objects are related to a specific student and are only used to transport data to Jinja. Special about this dataclasses is that it contains the three additional attributes: `total_flags`, `completed_flags` and `is_completed`, which are automatically set based on the list `flags`.
- **Flag.** `Flag` is used to manage flags in the backend as well as transfer them to Jinja. Special about this dataclasses is, that the parameter `is_completed` is optional to allow seamless usage in the backend.
- **Student.** `Student` is used analogously to `Flag` for the management of students in the backend, as well as for the transfer to Jinja. Special about this Dataclass is, that the `points` attribute can be set to allow seamless usage in the backend.

5.2.3.5. Base Logic

The base logic of the CTF framework is embodied in the `Controller` class. Therefore, the Controller class is the central control unit of the CTF framework. The controller implements the singleton design pattern and is, thus, accessible from anywhere in the project with `Controller.get_instance`.

The controller is integrated into the project in `__init__.py` through its control methods `setup`, `startup` and `shutdown`, which we discussed in Section 5.2.3.1. It is further queried by URL endpoints we discussed in Section 5.2.3.2. To perform its tasks efficiently, the controller always keeps an up-to-date dictionary of the students and challenges, as well as a sorted list of the students in memory. Overall, the controller implements mainly the following three types of functionality:

(a) Setup, startup, and shutdown of CTF framework.

(b) Processing of requests.

(c) Provision of essential auxiliary functionality.

Setup, startup, and shutdown of CTF framework. Setup, startup, and shutdown of the application are essential and give the application an underlying structure. They allow the integration of general setup, startup, and shutdown tasks, but especially of the individual `ChallengeCreators`. The exact functionality of those controller methods is as follows:

- **Setup of CTF framework (setup).** This method is called once when the CTF framework is started for the first time. After loading the controller, it is responsible for initializing the database, which means creating the database itself and filling it with start values, as well as initializing the statistics graph. Furthermore, `setup` is responsible for calling the `setup` method of the individual `ChallengeCreator`s. This ultimately enables the integration of challenge-specific setup tasks in the `ChallengeCreator`s, such as the setup of a laboratory environment. The existence of the `setup` methods of the individual `ChallengeCreator`s is ensured via the (abstract) parent class `ChallengeCreator` (Section 5.2.3.6).
- **Startup of CTF framework (startup).** This method is called at every program start, apart from the first time. After loading the controller, it ensures utilizing the `handle_new_day` method, that the database and statistics graph are always up-to-date. Furthermore, `startup` is responsible for calling the `startup` method of the individual `ChallengeCreator`s. Similar to setup, the challenges are responsible for themselves. The existence of `startup` is also ensured by the parent class `ChallengeCreator`.
- **Shutdown of CTF framework (shutdown).** This method is called every time the python interpreter gets normally terminated. As with `setup` and `startup`, the `shutdown` method is responsible for calling the `shutdown` method of each `ChallengeCreator`. This is essential to ensure proper termination of all laboratory environments that are started during the startup of the CTF framework. Again, analogous to `setup` and `startup`, the parent class `ChallengeCreator` ensures the existence of `shutdown`.

Processing of requests. Another main functionality of the controller is to process requests of the URL endpoints. The requests to the controller can be divided into the following categories:

- **Requests regarding challenges.** The requests regarding challenges are `get_challenges_for_student`, `get_challenge_for_student`, and `validate_flag`. The first two are fairly simple, as they basically forward the request to corresponding `ChallengeCreator` classes.

  The method `validate_flag` checks a submitted flag against the database and returns `True` or `False`, accordingly. If the flag is correct, `validate_flag` further calls `update` to update the whole application.
- **Requests regarding statistics.** The requests regarding statistics are `get_student` and `get_leaders`. Since the information for both of those requests is always kept up-to-date in memory by the controller those two requests are also fairly simple.

  It should also be noted that the statistics tab requires the statistics graph. The statistics graph is placed, also always up-to-date, in the `static/` folder and is included as an image in the `statistics.html` template.

Provision of essential auxiliary functionality. Besides the already mentioned functionality, the controller also has to implement some other essential features. Of course, the controller has more helper methods than listed below. However, they basically serve to implement the following essential auxiliary functionalities:

- **Setup of database (setup_db).** This method is called by `controller.setup` and first up creates the database using the `init_db` method of the `db.py` module. Furthermore, it initializes the database with starting values. This includes the participating students form `students.csv` into the `student` table, as well as dummy data into the `score` table, which is essential for initializing the statistics graph right from the start.

- **Setup of statistics graph (`setup_graph`).** This method is called by `controller.setup` as well `controller.update_graph` as updating works the same as initializing. It fetches the data of the participants of the last five days, creates the statistics graph using the Python module `matplotlib`, and stores it in the `static/` directory. A snapshot of the resulting graph can be seen in Figure 5.6. The method code in listed in Listing 5.13.

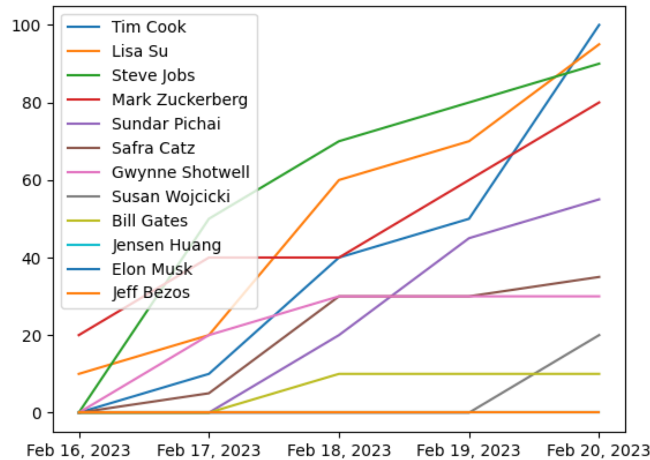


Figure 5.6.: Statistics graph for CTF framework.

```
for student in self.__leaders:
    dates = list()
    points = list()
    data = db.execute(
        '''SELECT * FROM score
           WHERE matriculation_number = ?
           AND date > ?
           ORDER BY date ASC''',
        (
            student.matriculation_number,
            date.today() + timedelta(days=-5)
        )
    ).fetchall()
    for entry in data:
        dates.append(
            datetime.strptime(
                entry['date'],
                '%Y-%m-%d'
            ).strftime("%b %d, %Y")
        )
        points.append(entry['points'])
    leaders_data[student.matriculation_number] = {
        'dates': dates,
        'points': points
    }
```

Listing 5.13: Retrieving of data for the statistics graph.

- **Loading of controller (`load_controller`).** As already mentioned, the controller always keeps an up-to-date dictionary of the students and challenges as well as a sorted list of the students. These are loaded when the CTF framework is set up or started by calling `load_challenges`, `load_students`, and `load_leaders`.

Slightly unusual is that the individual challenges are not imported and initialized directly in `__init__` but in `load_challenges` . This has the simple reason of avoiding circular imports.

- **Handling of new days (`handle_new_day`).** This method is, on the one hand, called by `controller.startup`. On the other hand, it is also called before every request by including it in the Flask `before_request` method in `views.py`. An excerpt of this method can be seen in Listing 5.14.

```
for entry in last_data:
    for i in range((date.today() - self.__reference_date).days):
        db.execute(
            '''INSERT INTO score (
                    matriculation_number,
                    date,
                    points
                )
                VALUES (?, ?, ?)''',
                (
                    entry['matriculation_number'],
                    date.today() + timedelta(days=-i),
                    entry['points']
                )
        )
        db.commit()
```

Listing 5.14: Handling of new days.

This method ensures that the database and, thereby, the statistics graph are always up-to-date by checking if a new day began, correspondingly inserting new data into the score table, and finally updating the statistics graph.

- **Update of data after successful flag submission (`update`).** Another important functionality required is updating all data after a successful flag submission. This is a reasonably complex task and therefore split up into four methods. Those methods are `update_db`, `update_students`, `update_leaders`, and `update_graph`. The `update` method then simply combines them all into a single method.

5.2.3.6. Challenge Logic

The implementation of the individual challenges is realized using an (abstract) parent class `ChallengeCreator`, as well as a child class for each challenge of EHL. As mentioned, unfortunately, Python does not natively provide abstract classes. For this reason, the design decision remains merely a concept to keep in mind and is not actually implemented in code.

The individual `ChallengeCreators` get essentially called by the controller methods `setup`, `startup`, `shutdown`, as well `get_challenge_for_student`. Additionally, a small auxiliary function `get_points_for_flags` is necessary.

`ChallengeCreator` implements the default functionality of a challenge and simultaneously ensures the existence of the functions required by the controller. Implementing an actual `ChallengeCreator` child class is then as easy as calling the parent `__init__` method. The default functionalities they inherit are as follows:

- **Setup of challenge (`setup`).** A challenge stores its information name, information text, and flags in the corresponding `resources/` folder which is read in at the start. Thus, the setup is limited to adding the solutions of the flags to the `flag` table. All

other information is read-in at every start of the program. Reading in the `flags.csv` file is done via the Python module `csv`.

For some of the challenges, it is necessary to generate unique flags. As this functionality is possibly relevant for future extensions of the project, we already implement this functionality directly in the default `setup` method.

To indicate to the CTF framework that a flag should be uniquely generated, we include the tag `??random??` in the solution. The choice of the tag `??` instead of `<` and `>` is rooted in the aim of not conflicting with HTML tags. Although it is unlikely that these tags appear in the solutions, they do appear in the information texts, questions, and hints. To keep the use of tags consistent across the project, we use `??` as the tag here as well.

Therefore, when reading in the flags, we use the following code to additionally check for the tag `??random??`, which is then replaced by a random eight-character string of letters, numbers, and special characters utilizing Python's `secrets` module, which can be seen in Listing 5.15. The choice for the structure of the random flags is justified in Section 6.1.2.

```
if '??random??' in flag_solution:
    if flag_number not in self._unique_flags:
        self._unique_flags.append(flag_number)
        symbols = (
            string.digits +
            string.ascii_letters +
            string.punctuation
        )
        random_string = ''.join(
            secrets.choice(symbols) for i in range(8)
        )
        flag_solution = flag_solution.replace(
            '??random??', '{' + random_string + '}'
        )
```

Listing 5.15: Generation of unique flags.

- **Startup of challenge (`startup`).** As mentioned before, all information, i.e., name, information text, and flags have to be read in at the startup of a challenge. While the name and information text are stored as normal strings, the flags are stored as `Flag` objects in a dictionary of the respective `ChallengeCreator`.
- **Shutdown of challenge (`shutdown`).** Since the standard `ChallengeCreator` does not have a laboratory environment and therefore no cleanup tasks, this method does not implement any functionality by default.
- **Return of Challenge object for student (`get_challenge_for_student`).** One of the most important functions of `ChallengeCreator` is the creation of a `Challenge` object for a student. To ensure that this method always returns a `Challenge` object and has the correct challenge number, this method should **not** be overridden, but instead only the methods `get_name_for_student`, `get_info_text_for_student` and `get_flags_for_student`. While the first two simply return the name and information text by default, `get_flags_for_student` returns a list of flags specific for a student, containing the attribute `is_completed`, which is queried from the database.

5.2.3.7. Extended Challenge Logic

The actual challenges of the EHL all implement a class and inherit from the (abstract) class `ChallengeCreator`. Thus, they automatically have the default functionalities of

ChallengeCreator (cf. Section 5.2.3.6). This is sufficient for all challenges of EHL, which do not have a virtual laboratory environment. Since they are all ChallengeCreators, consequently, they are essentially called by the controller methods setup, startup, shutdown, as well get_challenge_for_student.

Heap Buffer Overflow as well as *Hybrid Exploits And Use-After-Free*, are implemented by the classes HeapBufferOverflow and HybridExploitsAndUseAfterFree and have their own virtual laboratory environments. Therefore, they extend ChallengeCreator beyond the default functionality.

Since both classes are very similar, there is some redundant code. Pulling some of the methods up in the parent class ChallengeCreator only makes sense to a limited extent. The code is similar for HeapBufferOverflow and HybridExploitsAndUseAfterFree, but useless for all other ChallengeCreators. So, on the one hand, we try to avoid an excessively crowded ChallengeCreator parent class for clarity reasons and also to avoid name collisions. On the other hand, we also want to avoid overcomplicating the methods by parameterizing them and pulling them up into an extra parent class for those two challenges. For this reason, we tolerate some redundancies. The additional functionality of the extended ChallengeCreator classes is as follows:

- **Setup of container flags (setup_container_flags).** In Section 5.1, we have already implemented the laboratory environments and presupposed the directory structure. The CTF framework is now responsible for following this structure and filling the code_injection_flags and core_reuse_flags folders with the appropriate flags. This is done by the setup_container_flags method. It fetches the respective flags from the database and creates folders according to the student's matriculation number. Those folders then contain the flag as a text file.

  To prevent flags from past setups from interfering with the current ones, we delete the contents of the flag folders before creating new content. We do this with our own delete_folder_content method. This method deletes all folder contents, no matter if a file or a folder, except hidden files to preserve, e.g., .gitignore. The delete_folder_content method has been moved to the parent ChallengeCreator class as a static method and is, thus, available to all ChallengeCreators.

- **Display of dynamic information texts (get_info_text_for_student).** Providing students with customized information texts is important to dynamically display information, such as the IP address and port number of the victim machines in the laboratory environment. To implement this, the HTML information text contains tags in certain places to indicate that this is dynamic content. As with the generation of the flags, we use ?? as the start and end tag instead of < and > in order not to interfere with HTML tags. For example, we include the variable IP address and port number in the HTML information text with ??ip_address??:??port_code_injection??.

  The method get_info_text_for_student then searches for the tags and utilizes get_ports_for_student to replace the information accordingly. Since the laboratory environments are started sorted by matriculation number, the assignment of the port and a student can also be done by index. In Section 5.1.1.4, we have defined that port for student results from the start port of the challenge plus the index of the matriculation number in a sorted list. Therefore, the ports for a student then result from the following code:

```python
def __get_ports_for_student(self, matriculation_number):
    index = sorted(
        self._controller.students.keys()
    ).index(matriculation_number)
    return {
        "code_injection": str(30000 + index),
        "code_reuse": str(30100 + index)
    }
```

Listing 5.16: Association between students and ports.

- **Startup of the laboratory environments (`startup_lab`).** The laboratory environments were created in Section 5.1. We have also declared that the laboratory environments can be started and stopped using `vagrant up` or `vagrant halt` in the folder containing the Vagrant file. We implement this in the `startup_lab` method using the Python module `subprocess` and start the laboratory environment with the following code:

```python
log.info('Executing "vagrant up gohl_hbo"...')
log.info('This may take some time.')
process = subprocess.run(
    ['vagrant', 'up', 'gohl_hbo'],
    capture_output=True,
    text=True,
    cwd=self.__lab_path
)
```

Listing 5.17: Setup of a laboratory environments.

  The setup of the lab environment takes place automatically when `vagrant up` is called for the first time. For this reason, no extra setup method for the laboratory environments is needed. The `setup` method of `HeapBufferOverflow`, therefore, simply calls `startup_lab`.

- **Shutdown of the laboratory environments.** It is important to end the laboratory environments properly to avoid unnecessary processes on the server after the CTF framework has been terminated. Analogous to the start of the laboratory environments, this is done using the Python module `subprocess`.

5.3. Integration of Remaining Challenges

In this section, we deal with the integration of the other challenges, which do not receive an extra virtual laboratory environment. Furthermore, this section can also be seen as a general explanation for the integration of a challenge.

In Section 5.2.1, we have defined that each challenge has a subfolder in `goehl/resources/`. These subfolders contain `name.txt`, `info_text.html`, and `flags.csv`. Optionally, it can also contain a additional resources like a virtual laboratory environment. The CTF framework, further, specifies the structure in which the flags have to be entered. This is as follows:

```
'question','hint','solution','solution_format', 'points'
'Who is the partner of Bob?','She is a girl!','Alice','X****', '5'
```

Listing 5.18: Example .csv file of flags.

Further, there are a couple of things to note about the .csv files. These are as follows:

(a) **HTML formatting.** The flags are marked as `safe` in the frontend, which means they are rendered as HTML, accordingly. This allows, for example, the highlighting of content through `<strong>Bob</strong>`. Also, all Bootstrap [48] classes can be used.

(b) **Default hint.** The hint field can optionally be left blank. If this is the case, the CTF framework automatically inserts the default hint "No hint available.".

(c) **Randomized flags.** In Section 5.2.3.6, we have already mentioned that flags can be randomized. However, this is usually associated with a corresponding virtual laboratory environment. To randomize a flag, we add `??random??` to the solution. This is then replaced by the CTF framework with e.g. `{pB$SzYkE}`.

(d) **Solution format convention.** Furthermore, it is reasonable to agree on a convention for the solution format. TryHackMe[8], for example, uses asterisks. This, however, can lead to problems with upper and lower case. Formatting all answers in the backend to lower case was avoided, as answers such as `ls -a` and `ls -A` have different meanings and are case sensitive. For this reason, we agree on the convention that X stands for uppercase and * for lowercase. Of course, the solution format can also differ completely and contain two answer options, such as `goodfile or badfile`.

For the actual creation of the flags, the `.pdf` instructions for the challenges, provided on WueCampus [38], are used. These already contain some questions. Alternatively, they help with ideas for possible flags, such as syscall numbers or commands.

When creating the flags we try to use simple flags for fundamental things. An example from challenge 5 would be the question, which command is used to determine the version number of Docker, in order to verify the successful installation of the Docker engine.

With the questions, we do not want to put an additional burden on the students, as the main task are the challenges themselves. Rather, we want to use the flags to motivate the students and give them a first sense of achievement, as well as guide them in the right direction.

6. Evaluation

In this chapter, we evaluate our CTF project. This includes both, the virtual laboratory environment as well as the CTF framework. For both components of the CTF project, certain resources and information are necessary for the evaluation. Similar to Section 5.1, a distinction can be made between *Heap Buffer Overflow* and *Hybrid Exploits and Use-After-Free*. A clear comparison of the relevant information can be seen in Table 6.1.

In general, we use a very practical approach. We evaluate both, the virtual laboratory environments as well as the CTF framework, by practical testing of the required functionality and requirements. The general approach for the tests is described in the following sections. Based on this description, concrete, repeatable test plans written in imperative language can be found in Appendix E.

In the following section, we start with the security analysis of the CTF project in Section 6.1. Next, we describe the setup of our test environment in Section 6.2. Afterward, we deal with the evaluation of the virtual laboratory environment in Section 6.3 and the evaluation of the CTF framework in Section 6.4.

	HBO		HEAUAF	
	Code Injection	Code Reuse	Hybrid Exploits	Use-After-Free
Ports	30000 - 30099	30100 - 30199	30200 - 30299	30300 - 30399
Vulnerable Program	heap-overflow-mprotect.c	heap-overflow-rop.c	heap-overflow-rop.c	use-after-free.c
Additional Resources	shellcode.asm exploit.c	exploit.c	exploit.c	exploit.c
Files generated by exploit	goodfile badfile	goodfile badfile	goodfile badfile	goodfile badfile chapterfile

Table 6.1.: Relevant testing information for Heap Buffer Overflow (HBO) and Hybrid Exploits and Use-After Free (HEAUAF).

6.1. Security Analysis

In this section, we deal with IT security analysis of our CTF project. Unfortunately, within the scope of this work, we can not perform a comprehensive security analysis in the

form of a penetration test. However, we need to discuss a few critical design aspects of the CTF project. Therefore, in the following, we elaborate on the IT security of the virtual laboratory environments in Section 6.1.1, as well as the IT security of the CTF framework in Section 6.1.2.

6.1.1. Virtual Laboratory Environments

In this section, we deal with IT security analysis of our virtual laboratory environments. As already mentioned in Section 5.1.1.5, the IT security aspect of Docker containers is often not about actively taking security measures, but about avoiding common mistakes. For this, we orient ourselves on the OWASP Docker Security Cheat Sheet [45], as well as the best practices for writing Dockerfiles [42].

As we actively enable privilege escalation with our virtual lab environments, certain points are simply not feasible. Nevertheless, this does not necessarily lead to serious security vulnerabilities. Looking at our project, considering the OWASP cheat sheet as well as Dockerfile best practices, the following points need further discussion.

(a) Standard authentication for Target Containers

(b) Privilege escalation in Target Containers

(c) Removal of standard security profile from CTF containers

Standard authentication for Target Containers. the login on all target containers is done with username and password `victim`. If a student knows the port number of another student, which is easy to determine, he can log in to the victim machine of another student. Here, a malicious student could theoretically delete important files, such as the vulnerable program, or steal the compiled form of the exploit.

The aspect of the possibility to damage to others is bearable. Thus, an attacker does not gain any advantages by sabotaging others. If this nevertheless is the case, the virtual laboratory environment can easily be reset (cf. Appendix D.1). Identifying the attacker in such a small group should be fairly feasible, for example by checking the bash history of the participants or on a verbal level.

Also, the possibility to steal the exploit in binary form is acceptable. The participants have to write a protocol for the challenges anyway and include source code. Accordingly, the binary form of the exploit alone is not valuable.

Privilege escalation in Target Containers. With our virtual laboratory environments, we actively enable privilege escalation. Consequently, the aspect of avoiding privilege escalation or using a non-root user in the Docker containers is not feasible. However, we avoid known security risks for privilege escalation in Docker containers. Thus, we neither use the `-privileged` flag nor do we mount the Docker daemon socket. Also, we use the default security profile and do not add any extra capabilities. This makes breaking out of the container, even as a root, very unlikely. Furthermore, we have another intermediate layer through the VM. By running virtually on the host the VM brings additional protection to the server. The only connection between the server and the VM are synchronized folders. However, these are explicitly mounted as read-only to prevent any changes on the server.

Removal of standard security profile from CTF containers. To disable GDB's default behavior of disabling ASLR for the debugged process, we disable the default security profile of the CTF Containers. This, however, is justifiable. The connection between the container and the host system is even actively established by mounting the resources. This, however, is not problematic. The containers run exclusively locally. Not forwarding ports into the containers prevents any remote access to the CTF Containers.

6.1.2. CTF Framework

In this section, we deal with the IT security analysis of the CTF framework. In general, the orientation on the template project [29] and thus the use of Flask [26] and SQLite3 [50] already takes care of a lot of common security problems. This concerns the management of logged-in users and thus the implementation of page access. The correct implementation of page access in our CTF project is checked in Section 6.4. Also, by default, Flask together with the database library of SQLite3 protects against Structured Query Language (SQL)- and HTML injection [29].

Furthermore, we do not have to worry about secure password management, as we use a simple login by matriculation number. However, this is, among others, one conceivably vulnerable aspect that have been implemented in the CTF framework. Ultimately, the following three aspects need further discussion:

(a) Login solely via matriculation number.

(b) Receiving user input.

(c) Selection of flag length for randomized flags.

Login solely via matriculation number. The login solely via matriculation number is a potential security risk. However, it has to be considered what can actually be done by accessing a foreign account. Possible considerations are sabotaging other students, as well as stealing flags from other students.

The only input in the CTF framework is the matriculation number for login, as well as the submission of flags. Neither do we have flag submission attempt limits nor do we have penalty points for wrongly submitted flags. Thus, no damage can be done to other students. Also, stealing flags is not possible. After the successful submission of a flag, the flag only contains the flag question as well as a completed badge. Consequently, the solution is not visible.

Receiving user input. Receiving user input is a potential security risk for SQL- and HTML injection. Also, incorrect user input can lead to unexpected behavior or a program crash. As already mentioned, by using Flask [26] and SQLite3 [50] we are protected against SQL- and HTML injection by default. In Appendix E.2, we further test the interception of unexpected input at login.

Selection of flag length for randomized flags. With the randomized flags there is potential for brute-force attacks. Consequently, there has to be a large enough number of combinations for the flags. Our flags consist of eight characters of upper and lower case letters, numbers, as well as special characters. According to [53], this corresponds to a brute force time of well over one year, which is sufficient for this project.

6.2. Setup of Testing Environment

In this section, we deal with the setup of the testing environment. Thus, it is important to test the developed project and especially the setup separately from the original development environment to make sure that everything works on a replicable system. For the setup of our test environment, we have to implement the following points

(a) Setup of testing VM

(b) Installation of Docker inside VM

(c) Installation of Vagrant inside VM

(d) Acquisition of project repository

All of these points are fairly straightforward to implement. For installation guides for Docker and Vagrant, we again refer to the official website [54, 23]. For the setup of the testing VM, we utilize Vagrant to automate the process. Retrieving the repository is also quickly done via `git clone`.

However, we have to pay attention to two things in our test environment. First, we have to set the network adapter to *Bridged Adapter*. Otherwise, the Target Containers are not reachable from the CTF Containers. Second, we have to enable nested virtualization in the testing VM. We do this by enabling *Enable Nested VT-x/AMD-V* in the Central Processing Unit (CPU) settings of the VM in VirtualBox. A bug where the option is grayed out could be bypassed by the VBoxManage [55] CLI with the following command.

```
1 VBoxManage modifyvm <VM ID> --nested-hw-virt on
```

Listing 6.1: Enabling of nested virtualization in VM.

6.3. Evaluation of Virtual Laboratory Environments

In this section, we will deal with the evaluation of the virtual laboratory environments. As mentioned, we take a quite practical approach. To evaluate the virtual laboratory environments, we have to test the following things:

(a) Setup, startup, and shutdown of Target Containers

(b) Setup, startup, and shutdown CTF Containers

(c) Exploit Development including provided resources and tools, installation of tools, and parallel workflow

(d) Exploit transfer and exploit execution

Setup, startup, and shutdown of Target Containers. We try to perform our tests as isolated from each other as possible. For this reason, we refrain from setting up the virtual laboratory environments conveniently via the CTF framework, but manage and operate them manually.

In Appendix D.1 a user manual for the independent use of the Target Containers can be found. Thus, we start by copying the test flags from `test_resources` into the corresponding flag folders of the challenge we want to test. Subsequently, we start the laboratory environment to be tested.

To finally check the successful setup or startup, we simply test the SSH connection to the respective Target Containers of the challenge to be tested (cf. Table 6.1). After a successful startup, we terminate the laboratory environments. To test the successful shutdown of the laboratory environments, we use `vagrant global-status` to list all VMs and check that the corresponding VM and, thus, our laboratory environments are actually shut down.

Setup, startup, and shutdown CTF Containers. As the CTF Containers are already running independently from the CTF framework, verifying setup, startup, and shutdown of the CTF Containers is fairly simple. To do this, we first copy the folder of the challenge we want to check from `project/challenge_resources_for_students` to an arbitrary location. Then we change to the copied folder and follow the manuals for setup, startup, and shutdown from Appendix D.1. At the same time, we check that everything is working properly and no errors occur.

Exploit Development including provided resources and tools, installation of tools, and parallel workflow. Testing the exploit development is a bit more extensive. We start by launching our CTF Container for the challenge we want to check (cf. Appendix D.1). Then we verify that all necessary resources are available. A list of the required resources can be found in Table 6.1. To test the parallel workflow and the inclusion of new resources in the container, we simply create a test file inside the container. Then, we edit this file from the local host and check if the files are identical in the container and the local host.

Furthermore, we have to test if all required tools are installed and working properly and if the actual attack inside the container respectively inside the GDBs is possible. To test this, we first compile our vulnerable program. We also copy the exercise solution from `project/text_resources` into the container. After compiling and executing the exploit, we verify that goodfile, badfile, and chapterfile are in the directory, depending on the challenge (cf. Table 6.1). As a next step, we open the compiled vulnerable program in GDB. Here, we first check if Pwndbg is loaded successfully. Also, we verify in GDB that ROPgadget works properly. To test if the exploits are working, we run the vulnerable programs in GDB and check if a new shell opens. Even if the new shell in GDB is terminated in some cases, we check that at least the GDB output contains "process ... is executing new program: ...".

Last, we test that the installation of tools works. This means, basically, the functionality of APT and `sudo`. This test is also done quickly by simply updating our package list, and then checking if the installation of an arbitrary tool using APT is working.

Exploit transfer and exploit execution. Also, the evaluation of exploit transfer and execution is a bit more complex. Logically we, again, have to start both the Target- and CTF Containers (cf. Appendix D). Then we copy our solution from `project/text_resources` into the CTF container. As a next step, we check if the exploit transfer using SCP is successful. The challenge ports can be found in Table 6.1. After successful transmission, we connect to the victim using SSH.

Before checking the exploit, we first check on the victim side that `secret_file` is actually non-accessible without a successful exploit. For this, we change to the `/secret_files` directory and verify that `secret_file` is not readable. Then we test that neither via `sudo -s` nor `su` it is possible to switch to root. Additionally, we verify with `groups victim` that victim is not in the sudo group.

To test that the actual exploit works, we first run the exploit to create goodfile, badfile or chapterfile, respectively. Then we run the vulnerable program to verify that a shell opens and the exploit works. After the successful exploit, we check with `whoami` that we actually have a root shell. With the newly obtained root rights, we then read `secret_file` and verify that it is identical to the flag initially mounted in the VM for the tested participant.

6.4. Evaluation of CTF Framework

In this section, we describe the evaluation of the CTF framework. As in Section 6.3, we describe the general test approach. The concrete test plans can be found in Appendix E. In order to structure the evaluation of the CTF framework, we divide the functionalities to be tested into the following four points:

(a) Setup, startup, and shutdown

(b) Authentication and page access

(c) Navigation and static page content display

(d) Flag validation and dynamic page content display

Setup, startup, and shutdown. Testing setup, startup, and shutdown is fairly simple. As a first step, we copy the `students.csv` file from `test_resources` to `goehl/resources` for testing reasons. Then we simply follow the manuals from Appendix D and verify that there are no errors in setup, startup, and shutdown, that the correct logs are displayed, and that the website is accessible after startup. Furthermore, we verify that in `project/instance` the database file `goehl.sqlite` has been created and that it holds the correct database entries.

Challenge-specific we also verify that the flag folders of Heap Buffer Overflow and Hybrid Exploits and Use-After-Free contain the required flags. Using `vagrant global-status`, we further verify that the laboratory environments are properly started and stopped by the application.

Authentication and page access. To check authentication and page access, we logically, again, start our CTF framework first. After that, we visit the login page and try to log in with different fake credentials including numbers and words to prevent unauthorized login. Afterward, we verify that a login with valid credentials is working.

Furthermore, we test that the session management works correctly. For this, we copy the URL of different pages and log out. Then we try to access the copied URLs by so-called force browsing. Thereby, we verify that without a valid login, none of the pages can be accessed and, instead, the user is redirected to the login page.

Navigation and static page content display. The procedure for testing the navigation and static page content display is also straightforward. We, again, start with the launch of the CTF framework. After that, we visit all pages including statistics, challenge overview, and single challenge pages. On these, we verify that the pages are displayed with the desired elements and information. Here, we also test that all click events work and forward correctly.

Flag validation and dynamic page content display. The evaluation of the flag validation and dynamic page content display is a bit more complex. We, again, start with launching the CTF framework. Then, we log in with two accounts in two different browsers at the same time. As a next step, we extract the solutions for a randomly generated flag for a user out of the database file and submit the flag on both accounts. Hereby we check that the flags are indeed different for the students. Next, we check on both accounts if all dynamic displays as well as the statistics are updated correctly. For another test, we use the time settings of the VM. Thus, we advance the time by several days and check if the statistics are updated correctly.

6.5. Test Results

The final execution of the tests is successful. Neither the virtual laboratory environments nor the CTF framework show any errors. Also, the additional security aspects defined in Section 6.1, namely page access and user input, are all implemented properly.

7. Conclusion and Future Work

In this thesis, we have improved EHL through gamification respectively the development of a CTF project. Specifically, we developed a CTF framework to host and administrate challenges of EHL. We also integrated various game elements like points and statistics into the CTF framework to enhance the game character and to motivate the participants.

Furthermore, we have developed virtual laboratory environments for the binary exploitation exercises of EHL using Docker and Vagrant to allow the participants to actually execute their developed exploits. With the laboratory environments, we also relieve the participants from tedious setup tasks as well as the tutors of the EHL from manual verification of submitted solutions.

We have also integrated all other exercises of EHL into the CTF framework. We integrated them in the form of various questions about the exercises, i.e. in the form of a questionnaire. In order to ensure proper functionality of our project, we have developed and conducted test plans as part of the evaluation process.

Future work. For future work, an additional evaluation of the project in the form of surveys of the participants would be interesting. This would give insights into the actual effectiveness of the improvements of the CTF project. However, since EHL is an official university module, it is difficult to form two reference groups, as every participant should have equal opportunities.

Another possibility for future work is extending the project. As our project was developed with a lot of emphasis on extensibility, the extension of the CTF project in the form of further virtual laboratory environments for other challenges is quite reasonable. Especially challenge 4 of EHL would be suitable for this. Also, an extension of the CTF framework through randomized challenges would be useful. This would make cheating significantly more difficult. Another possible enhancement would be the inclusion of additional game elements, such as avatars or pseudonyms, which would further improve the game characteristics of the CTF framework.

List of Figures

2.1. Stack layout when calling “function” [19]. 9
2.2. Abstract concept of Docker and VMs [22] 10

4.1. Current workflow of a binary exploitation exercise. 18
4.2. Aspired workflow of a binary exploitation exercise. 19
4.3. Structure of Virtual Laboratory Environments. 23
4.4. Structure of CTF framework. 24
4.5. GUI - Statistics page. 25
4.6. GUI - Challenge page. 25
4.7. GUI - Challenge page. 26

5.1. Folder structure of Target Containers. 28
5.2. Folder structure predefined by virtual laboratory environments. 31
5.3. Folder structure of CTF Containers. 34
5.4. Folder structure of CTF framework. 39
5.5. UML diagram of CTF framework. 42
5.6. Statistics graph for CTF framework. 47

A.1. GUI Prototype - Login page. 73
A.2. GUI Prototype - Statistics page. 74
A.3. GUI Prototype - Challenges page. 74
A.4. GUI Prototype - Single Challenge page. 75
B.5. GUI - Login page. 76
B.6. GUI - Challenges page. 76

List of Tables

5.1. Port range for Heap Buffer Overflow and Hybrid Exploits and Use-After-Free. 32

6.1. Relevant testing information for Heap Buffer Overflow (HBO) and Hybrid Exploits and Use-After Free (HEAUAF). 53

D.1. Variable parameters for Heap Buffer Overflow (HBO) and Hybrid Exploits and Use-After Free (HEAUAF). 80

Listings

2.1. Example of buffer overflow vulnerable C program based on [19]. 9
2.2. Example of use-after-free vulnerable C program based on [20]. 9

5.1. Special .gitignore file to include empty directories. 28
5.2. Enabling of remote access inside Docker container. 29
5.3. Creation of user inside Dockerfile. 30
5.4. Excerpt of startup script for Target Containers. 32
5.5. Enabling of GDB's default behavior to disable ASLR for the debugged process. 35
5.6. Provision of necessary resources inside Docker container. 36
5.7. Installation of packages inside Docker container. 36
5.8. Code responsible for setting .gdbinit. 37
5.9. Commands to fix broken paths after Pwndbg/ROPgadget installation. . . . 37
5.10. Enabling of sudo in a Docker container. 37
5.11. Display of multiple flags in HTML template. 40
5.12. Automatic closing of completed flags. 41
5.13. Retrieving of data for the statistics graph. 47
5.14. Handling of new days. 48
5.15. Generation of unique flags. 49
5.16. Association between students and ports. 51
5.17. Setup of a laboratory environments. 51
5.18. Example .csv file of flags. 51

6.1. Enabling of nested virtualization in VM. 56

7.1. Vagrantfile for Target Containers. 77
7.2. Initial startup script for setup tasks inside of VM. 77
7.3. Startup script for Target Containers. 78
7.4. Dockerfile for Target Containers. 78
7.5. Dockerfile for CTF Containers. 79
7.6. Docker command for starting of CTF Containers. 79

Acronyms

ASLR Address Space Layout Randomization

CLI Command Line Interface

CTF Capture The Flag

EHL Ethical Hacking Lab

VM Virtual Machine

ROP Return-oriented programming

OS Operating System

DEP Data Execution Prevention

APG Automatic Problem Generation

LOC Lines Of Code

GUI Graphical User Interface

GCC GNU Compiler Collection

SSH Secure Shell

APT Advanced Packaging Tool

RAM Random Access Memory

SUID Set User ID

ROP Return Oriented Programming

URL Uniform Resource Locator

JSON JavaScript Object Notation

HTML Hypertext Markup Language

IP Internet Protocol

PC Personal Computer

SCP Secure Copy

UML Unified Modeling Language

LKM Loadable Kernel Module

PoW Proof of Work

CPU Central Processing Unit

SQL Structured Query Language

Bibliography

[1] "Hiscox Cyber Readiness Report 2022." https://www.hiscoxgroup.com/cyber-readiness. Accessed: 08. September 2022.

[2] "2022 Norton Cyber Safety Insights Report: Special Release – Online Creeping." https://www.nortonlifelock.com/us/en/newsroom/press-kits/2022-norton-cyber-safety-insights-report-special-release-online-creeping/. Accessed: 08. September 2022.

[3] S. Stieglitz, C. Lattemann, S. Robra-Bissantz, R. Zarnekow, and T. Brockmann, eds., *Gamification: Using Game Elements in Serious Contexts.* Cham: Springer International Publishing, 2017. eBook ISBN: 978-3-319-45557-0.

[4] K. Leune and S. J. Petrilli, "Using Capture-the-Flag to Enhance the Effectiveness of Cybersecurity Education," in *Proceedings of the 18th Annual Conference on Information Technology Education*, (New York, NY, USA), pp. 47–52, Association for Computing Machinery, 2017.

[5] M. Beltrán, M. Calvo, and S. González, "Experiences Using Capture The Flag Competitions to Introduce Gamification in Undergraduate Computer Security Labs," in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, (Los Alamitos, CA, USA), pp. 574–579, IEEE Computer Society, 2018.

[6] V. Ford, A. Siraj, A. Haynes, and E. Brown, "Capture the Flag Unplugged: An Offline Cyber Competition," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, (New York, NY, USA), p. 225–230, Association for Computing Machinery, 2017.

[7] K. Chung, "Live Lesson: Lowering the Barriers to Capture The Flag Administration and Participation," in *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, (Vancouver, BC), USENIX Association, 2017.

[8] "TryHackMe." https://tryhackme.com/. Accessed: 05. October 2022.

[9] "Hack The Box." https://www.hackthebox.com/. Accessed: 05. October 2022.

[10] "Cambridge Dictionary - Gamification." https://dictionary.cambridge.org/de/worterbuch/englisch/gamification. Accessed: 07. September 2022.

[11] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From Game Design Elements to Gamefulness: Defining "Gamification"," in *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, (New York, NY, USA), p. 9–15, Association for Computing Machinery, 2011.

[12] M. Sailer, *Wirkung von Gamification auf Motivation.* Wiesbaden: Springer Fachmedien Wiesbaden, 2016. eBook ISBN: 978-3-658-14309-1.

[13] "CTF Time - What is Capture The Flag?." https://ctftime.org/ctf-wtf/. Accessed: 10. September 2022.

[14] "OWASP - Buffer Overflow." https://owasp.org/www-community/vulnerabilities/Buffer_Overflow. Accessed: 10. September 2022.

[15] "ProSec - Buffer Overflow Angriff." https://www.prosec-networks.com/blog/buffer-overflow-angriff/. Accessed: 10. September 2022.

[16] M. Prandini and M. Ramilli, "Return-Oriented Programming," *IEEE Security & Privacy*, vol. 10, no. 6, pp. 84–87, 2012.

[17] "Ret2Libc and ROP." https://exploit.ph/x86-32-linux/2014/08/06/ret2libc-and-rop/index.html. Accessed: 10. September 2022.

[18] "Code Arcana - Introduction to return oriented programming." https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html. Accessed: 10. September 2022.

[19] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[20] "OWASP - Using freed memory." https://owasp.org/www-community/vulnerabilities/Using_freed_memory. Accessed: 27. December 2022.

[21] "Kaspersky - Use-After-Free." https://encyclopedia.kaspersky.com/glossary/use-after-free/. Accessed: 27. December 2022.

[22] "Docker - Use containers to Build, Share and Run your applications." https://www.docker.com/resources/what-container/. Accessed: 23. April 2022.

[23] "HashiCorp - Introduction to Vagrant." https://developer.hashicorp.com/vagrant/intro. Accessed: 27. December 2022.

[24] "VirtualBox." https://www.virtualbox.org/. Accessed: 31.10.22.

[25] "VMware." https://www.vmware.com/. Accessed: 02.01.23.

[26] "Flask." https://palletsprojects.com/p/flask/. Accessed: 31.10.2022.

[27] "Jinja." https://palletsprojects.com/p/jinja/. Accessed: 02.11.2022.

[28] "The Pallets Project." https://palletsprojects.com/. Accessed: 02.11.2022.

[29] "Flask Sample Project." https://flask.palletsprojects.com/en/2.2.x/tutorial/layout/. Accessed: 31.10.2022.

[30] K. M. Kapp, *The Gamification of Learning and Instruction: Game-Based Methods and Strategies for Training and Education.* Pfeiffer & Company, 2012. ISBN: 978-1-118-09634-5.

[31] K. Huotari and J. Hamari, "Defining Gamification: A Service Marketing Perspective," in *Proceeding of the 16th International Academic MindTrek Conference*, (New York, NY, USA), p. 17–22, Association for Computing Machinery, 2012.

[32] F. F.-H. Nah, Q. Zeng, V. R. Telaprolu, A. P. Ayyappa, and B. Eschenbrenner, "Gamification of Education: A Review of Literature," in *HCI in Business*, (Cham), pp. 401–409, Springer International Publishing, 2014.

[33] D. Dicheva, C. Dichev, G. Agre, and G. Angelova, "Gamification in Education: A Systematic Mapping Study," *Journal of Educational Technology & Society*, vol. 18, no. 3, pp. 75–88, 2015.

[34] S. Strahringer and C. Leyh, *Gamification und Serious Games: Grundlagen, Vorgehen und Anwendungen*. Wiesbaden: Springer Fachmedien Wiesbaden, 2017. ISBN: 978-3-658-16742-4.

[35] J. Burket, P. Chapman, T. Becker, C. Ganas, and D. Brumley, "Automatic Problem Generation for Capture-the-Flag Competitions," in *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*, (Washington, D.C.), USENIX Association, 2015.

[36] J. Vykopal, V. Švábenský, and E.-C. Chang, "Benefits and Pitfalls of Using Capture the Flag Games in University Courses," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, (New York, NY, USA), p. 752–758, Association for Computing Machinery, 2020.

[37] "CTFd." `https://github.com/CTFd/CTFd`. Accessed: 24. April 2022.

[38] "WueCampus." `https://wuecampus2.uni-wuerzburg.de/moodle/`. Accessed: 30.12.2022.

[39] "Ubuntu 20.04 Docker Image." `https://hub.docker.com/_/ubuntu`. Accessed: 31.1.2023.

[40] "Teleport - SSH into Docker Container." `https://goteleport.com/blog/shell-access-docker-container-with-ssh-and-docker-exec/`. Accessed: 31.10.22.

[41] "How to SSH into Docker Container." `https://adamtheautomator.com/ssh-into-docker-container/`. Accessed: 31.10.22.

[42] "Docker - Best practices for writing Dockerfiles." `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`. Accessed: 31.10.2022.

[43] "Docker - Docker Compose." `https://docs.docker.com/compose/`. Accessed: 07.01.2023.

[44] "ShellCheck - SC2012." `https://www.shellcheck.net/wiki/SC2012`. Accessed: 31.10.22.

[45] "OWASP - Docker Security Cheat Sheet." `https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html`. Accessed: 22.02.23.

[46] "Github - Pwndbg." `https://github.com/pwndbg/pwndbg`. Accessed: 01.11.2022.

[47] "Github - ROPgadget." `https://github.com/JonathanSalwan/ROPgadget`. Accessed: 02.11.2022.

[48] "Bootstrap5 - Getting started." `https://getbootstrap.com/docs/5.0/getting-started/introduction/`. Accessed: 02.11.2022.

[49] "Python." `https://www.python.org/`. Accessed: 31.10.2022.

[50] "SQLite3." `https://docs.python.org/3/library/sqlite3.html`. Accessed: 11.11.2022.

[51] "Python - Abstract Base Classes." `https://docs.python.org/3/library/abc.html`. Accessed: 30.12.2022.

[52] "PEP 557 - Data Classes." `https://peps.python.org/pep-0557/`. Accessed: 02.01.23.

[53] "Datenschutz - Brute Force." `https://www.datenschutz.org/brute-force/`. Accessed: 23.02.2023.

[54] "Docker - Orientation and setup." https://docs.docker.com/get-started/. Accessed: 23. April 2022.

[55] "VBoxManage." https://www.virtualbox.org/manual/ch08.html. Accessed: 31.10.22.

Appendix

A. GUI Prototype of CTF Framework

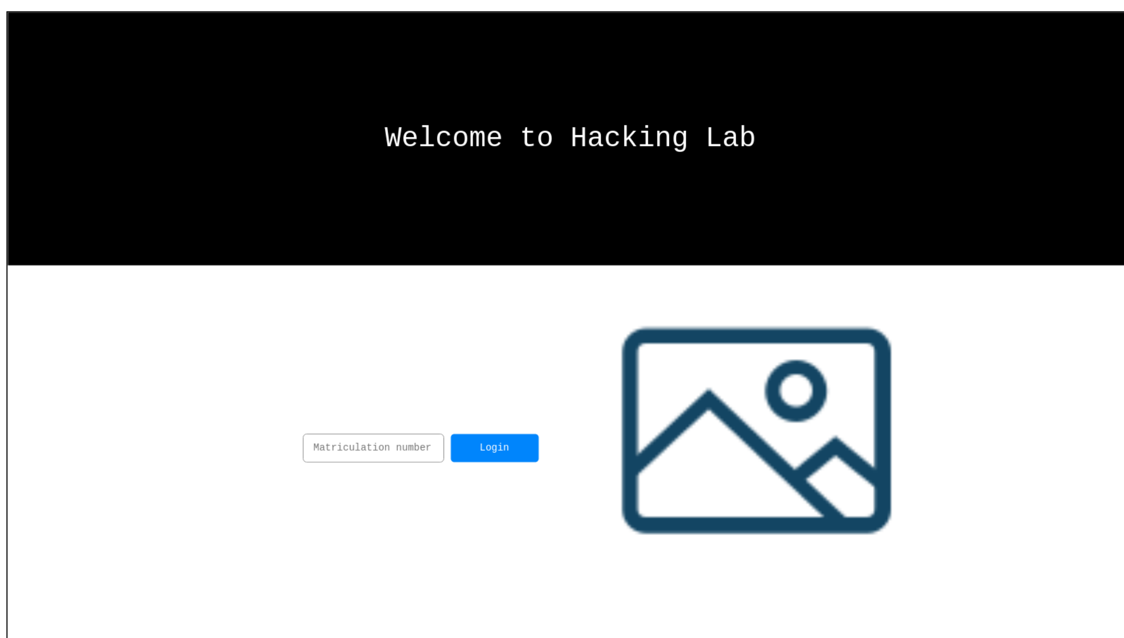


Figure A.1.: GUI Prototype - Login page.

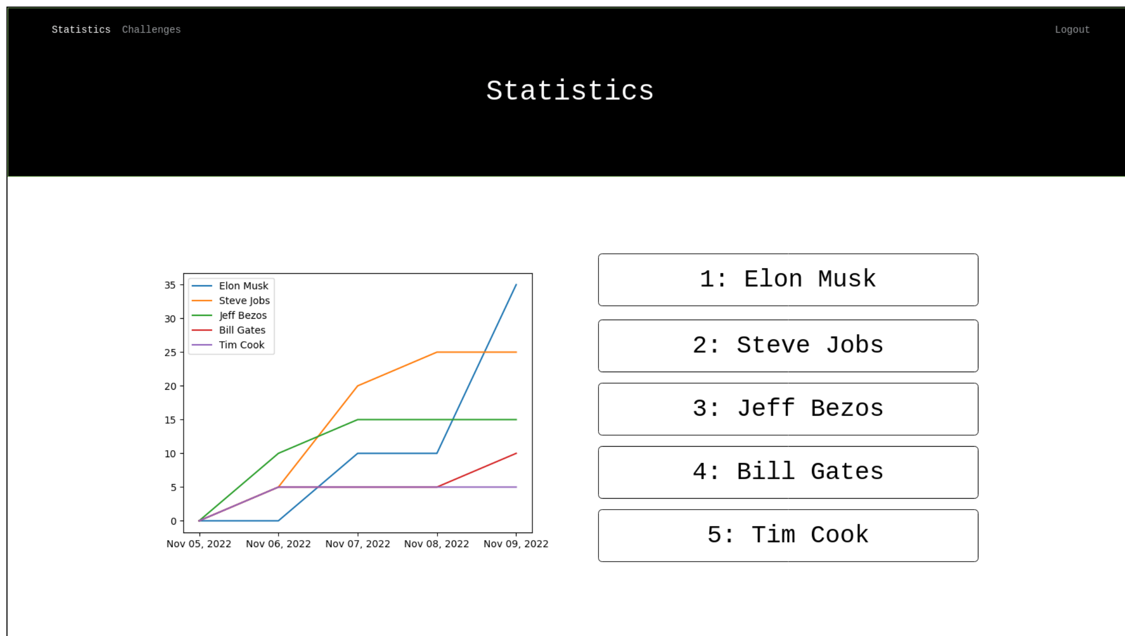


Figure A.2.: GUI Prototype - Statistics page.

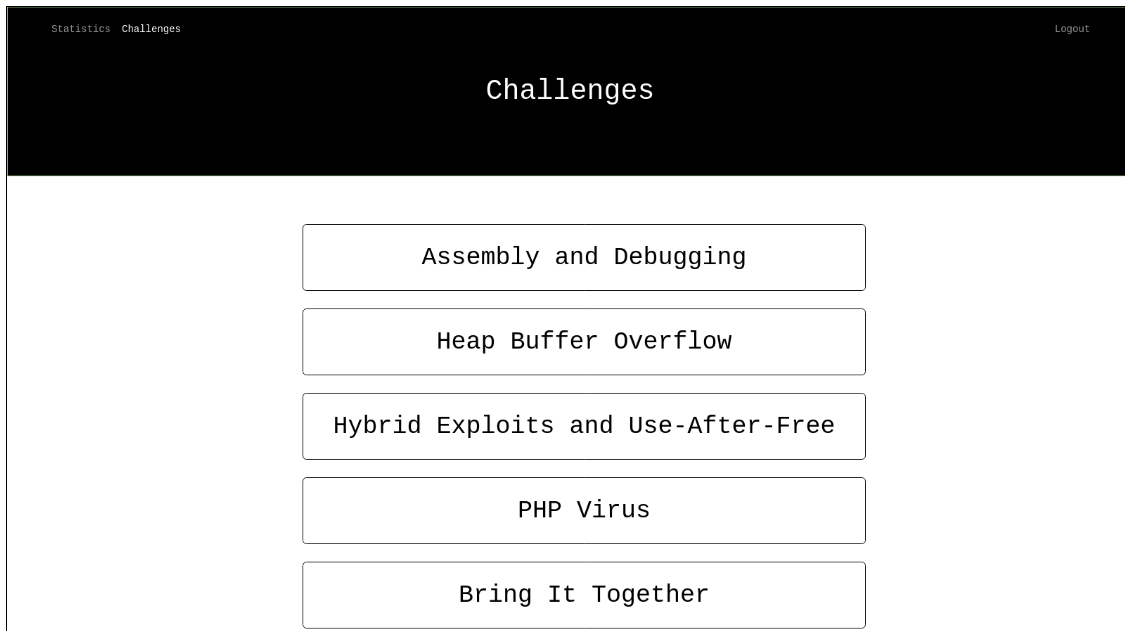


Figure A.3.: GUI Prototype - Challenges page.

Statistics Challenges Logout

Heap Buffer Overflow

Lorem ipsum

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Lorem ipsum?

Completed

Lorem ipsum?

Format: *** ****** Submit Hint

Lorem ipsum?

Format: *** ****** Submit Hint

Lorem ipsum?

Format: *** ****** Submit Hint

Figure A.4.: GUI Prototype - Single Challenge page.

B. Additional GUI Screenshots of CTF Framework

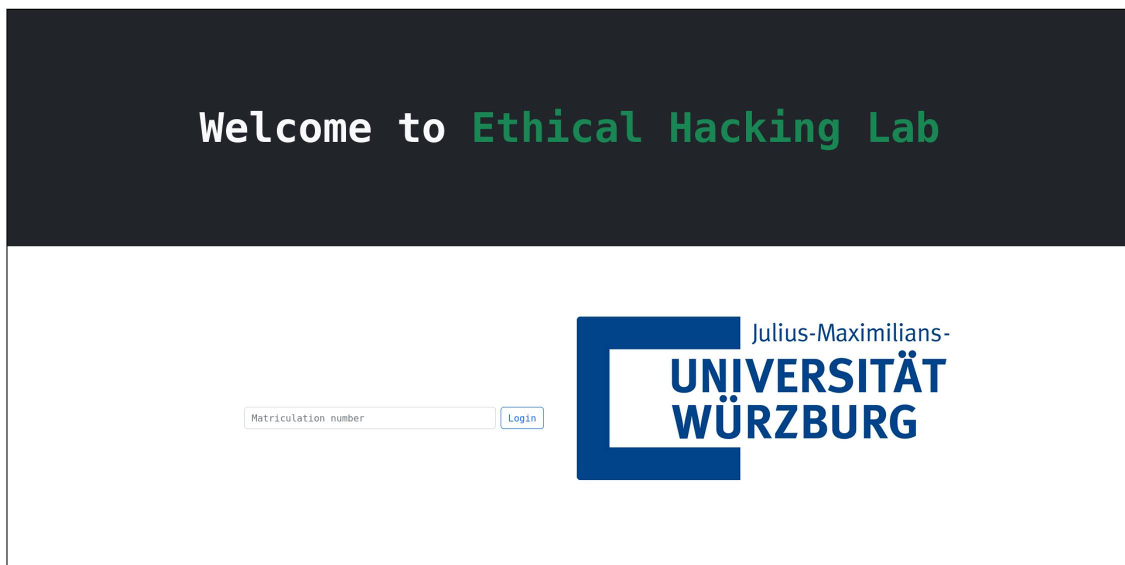


Figure B.5.: GUI - Login page.

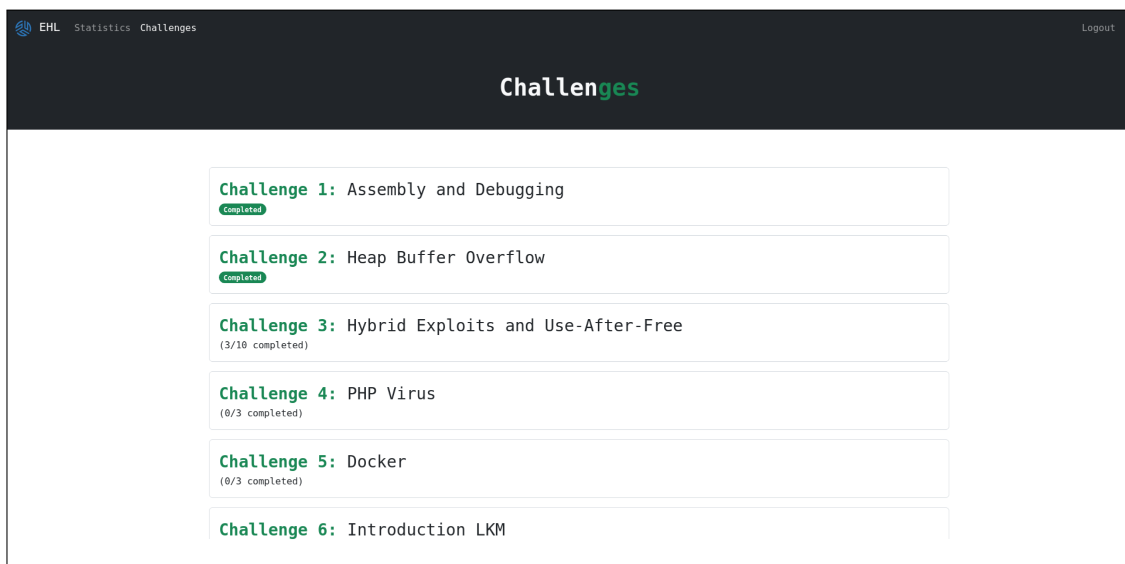


Figure B.6.: GUI - Challenges page.

C. Resources for Virtual Laboratory Environments

C.0.1. Target Containers

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  config.vm.define "goehl_hbo"
  config.vm.box = "ubuntu/focal64"

  config.vm.provider "virtualbox" do |vb|
    vb.gui = false
    vb.memory = "2048"
  end

  for i in 30000..30199
    config.vm.network "forwarded_port", guest: i, host: i
  end

  config.vm.synced_folder "./code_injection_flags", "/home/vagrant/vagrant_code_injection_flags", :mount_options => ["ro"]
  config.vm.synced_folder "./code_reuse_flags", "/home/vagrant/vagrant_code_reuse_flags", :mount_options => ["ro"]
  config.vm.synced_folder "./docker_code_injection", "/home/vagrant/vagrant_docker_code_injection", :mount_options => ["ro"]
  config.vm.synced_folder "./docker_code_reuse", "/home/vagrant/vagrant_docker_code_reuse", :mount_options => ["ro"]

  config.vm.provision "shell", path: "initial_startup.sh"
  config.vm.provision "shell", path: "start_containers.sh", run: 'always'
end
```

Listing 7.1: Vagrantfile for Target Containers.

```sh
#!/bin/sh

apt-get remove -y docker docker-engine docker.io containerd runc
apt-get update
apt-get install -y \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
apt-get update
apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin

docker build -t code_injection_image /home/vagrant/vagrant_docker_code_injection
docker build -t code_reuse_image /home/vagrant/vagrant_docker_code_reuse
```

Listing 7.2: Initial startup script for setup tasks inside of VM.

```
#!/bin/sh

sysctl -w kernel.randomize_va_space=0
rm -rf code_injection_flags code_reuse_flags
mkdir code_injection_flags code_reuse_flags
cp -r vagrant_code_injection_flags/* code_injection_flags/
cp -r vagrant_code_reuse_flags/* code_reuse_flags/

start_containers () {
  port=$1
  for folder in $(ls $2 | sort -g); do
    volume="$PWD/$2/$folder"
    chmod 777 "$volume"
    chmod 600 "$volume/"*
    chown root:root "$volume/"*
    echo "Starting container with port $port and folder $volume..."
    docker run -d -p "$port:22" -v "$volume:/secret_files" -t "$3"
    port=$((port+1))
  done
}
start_containers 30000 code_injection_flags code_injection_image
start_containers 30100 code_reuse_flags code_reuse_image
```

Listing 7.3: Startup script for Target Containers.

```
FROM ubuntu:20.04
ARG DEBIAN_FRONTEND=noninteractive

RUN groupadd -r victim && useradd --no-log-init -r -m -g victim victim
RUN echo "victim:victim" | chpasswd
RUN usermod -s /bin/bash victim
RUN usermod -s /bin/bash root
RUN apt-get update && apt-get install -y \
  openssh-server \
  gcc \
  vim \
  nano \
  && rm -rf /var/lib/apt/lists/*

WORKDIR /home/victim
RUN mkdir Desktop Downloads Music Documents Pictures Videos
RUN chmod -R 755 /home/victim
WORKDIR /home/victim/Music
RUN touch dont-stop-believin_journey.mp3 africa_toto.mp3 never-gonna-give-
    you-up_rick-astley.mp3
WORKDIR /home/victim/Pictures
RUN touch 1982-03-23_spain_cats.jpeg 1984-05-16_france_dogs.jpeg
WORKDIR /home/victim/Videos
RUN touch 1982-03-23_spain_cats-walking.mp4 1984-05-16_france_dogs-walking.
    mp4

RUN mkdir /root/secret_files
WORKDIR /home/victim/
COPY heap-overflow-mprotect.c .
RUN gcc -static -o heap-overflow-mprotect heap-overflow-mprotect.c
RUN chmod 4771 heap-overflow-mprotect
RUN rm heap-overflow-mprotect.c

EXPOSE 22
RUN mkdir -p /var/run/sshd
CMD ["/usr/sbin/sshd","-D"]
```

Listing 7.4: Dockerfile for Target Containers.

C.0.2. CTF Containers

```
FROM ubuntu:20.04
ARG DEBIAN_FRONTEND=noninteractive
ENV LC_ALL C.UTF-8

RUN groupadd -r bob && useradd --no-log-init -r -m -g bob -G sudo bob
RUN echo "bob:bob" | chpasswd
RUN usermod -s /bin/bash bob
RUN mkdir /home/bob/ci
RUN mkdir /home/bob/cr

RUN apt-get update && apt-get install -y \
    sudo \
    python3-pip \
    git \
    gdb \
    vim \
    nano \
    nasm \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /opt
RUN git clone https://github.com/pwndbg/pwndbg
WORKDIR /opt/pwndbg
RUN ./setup.sh

RUN cp /root/.gdbinit /home/bob/
ENV PATH=/usr/local/lib/python3.8/dist-packages/bin:$PATH
RUN chown -R bob:bob /home/bob

WORKDIR /home/bob
USER bob
```

Listing 7.5: Dockerfile for CTF Containers.

```
docker run \
    --name heap_buffer_overflow \
    --security-opt seccomp=unconfined \
    -v "$PWD/code_injection:/home/bob/ci" \
    -v "$PWD/code_reuse:/home/bob/cr" \
    -it heap_buffer_overflow_image \
    bash
```

Listing 7.6: Docker command for starting of CTF Containers.

D. User Manuals

D.1. Virtual Laboratory Environments

Target Containers:

Setup, Startup and Shutdown:

Instruction 1: Acquire the `lab_environment` folder for the challenge. Open up a terminal and change directory to the acquired folder. Then, populate the flag folders for both exercises with folders named after the participants' matriculation numbers, each containing a file `super_secret_file` with a unique flag.

Instruction 2: Make sure you are in the `lab_environment` directory. Then, execute `vagrant up` to setup the virtual laboratory environment.

Instruction 3: To shutdown the virtual laboratory environment execute `vagrant halt`.

Instruction 4: From now on the the virtual laboratory environment can be started by executing `vagrant up` as in the setup.

Reset/Removal:

Instruction 1: If the laboratory environments were altered and have to be reset use `vagrant global-status` to list all vagrant VMs. Then use vagrant destroy <id> to destroy the challenge VM.

CTF Containers:

Setup, Startup, and Shutdown:

For the challenges Heap Buffer Overflow and Hybrid Exploits and Use-After-Free different parameters are required for the setup commands. These are listed in Table D.1. The ready to use copy pasteable commands can be found on the challenge pages of the website.

	HBO		HEAUAF	
	Code Injection	Code Reuse	Hybrid Exploits	Use-After-Free
c_name	heap_buffer_overflow		hybrid_exploits_and_use_after_free	
e_name	code_injection	code_reuse	hybrid_exploits	use_after_free
e_abbr	ci	cr	he	uaf
c_image	<c_name>_image		<c_name>_image	

Table D.1.: Variable parameters for Heap Buffer Overflow (HBO) and Hybrid Exploits and Use-After Free (HEAUAF).

Instruction 1: Acquire the `heap_buffer_overflow` folder. Open up a terminal and change directory to the acquired folder. Then, run the following command to build the `heap_buffer_overflow_image` image.

```
docker build -t <c\_image> .
```

Instruction 2: Run the following command to set up the CTF Container:

```
docker run \
    --name <c\_name> \
    --security-opt seccomp=unconfined \
    -v "$PWD/<e\_name>:/home/bob/<e\_abbr>" \
    -v "$PWD/<e\_name>:/home/bob/<e\_abbr>" \
    -it <c_image> \
    bash
```

Instruction 3: To exit the container hit `ctrl+d` or enter "exit" into the terminal.

Instruction 4: From now on the container can be started from anywhere with the following command:

```
docker start -ai <c_name>
```

Reset/Removal:

Instruction 1: Use `docker ps -a` to list all Docker containers. Then, use `docker rm <container id>` to delete the desired container.

Instruction 2: If desired use `docker images` to list all Docker containers. Then use `docker rmi <image id>` to delete the challenge image.

D.2. CTF Framework

Installation:

Instruction 1: Make sure Docker and Vagrant are installed.

Instruction 2: Clone the project git repository.

Instruction 3: Change directory to `project`. Then run the following command to set up a python virtual environment:

```
python3 -m venv venv && . venv/bin/activate
```

Instruction 4: Use the following command to install the needed requirements:

```
python -m pip install -r requirements.txt
```

Instruction 5: Run the following command to refresh the python virtual environment:

```
. venv/bin/activate
```

Setup, Start, and Shutdown:

Instruction 1: Enter all participating students into `students.csv`. If desired edit `name.txt`, `info_text.html`, or `flags.csv` of the challenges inside `resources`.

Instruction 2: Make sure the python virtual environment is activated. Then run the following command to set up the CTF framework (This may take some minutes):

```
flask --app goehl run
```

Instruction 3: Repeat the previous command to start the CTF framework (This may take some seconds).

Instruction 4: Terminate the Python interpreter to shut down the CTF framework.

Reset:

Instruction 1: Delete, rename, or remove the file `goehl.sqlite` in `project/goehl/instance`.

Instruction 2: If the laboratory environments were altered and have to be reset we refer to Appendix D.1.

Deinstallation:

Instruction 1: Use `vagrant global-status` to list all vagrant VMs. Then use `vagrant destroy <id>` to destroy all VMs starting with "goehl".

Instruction 2: Delete the entire `project` folder.

E. Testplans

E.1. Virtual Laboratory Environments

TID-1: Setup, Startup, Shutdown of Target Containers

Instruction 1: Reset the Target Containers (cf. Appendix D.1.

Instruction 2: Use the flags from `project/test_resources/test_flags` and follow Appendix D.1 to set up and start the Target Containers.

Instruction 3: Open up a new terminal and try to connect to Target Container for the challenge via SSH (cf. Table 6.1). The username and password are both `victim`:

Expected Result 1: The command prompt `victim@<random code>` appears.

Instruction 4: Switch to the terminal, which started the containers to shut down the containers by running `vagrant halt`.

Expected Result 2: The shutdown of the VM is successful. The SSH connection gets closed.

Instruction 5: Use `vagrant global-status` to list all vagrant VMs.

Expected Result 3: The output contains the challenge VM with the state "poweroff".

TID-2: Setup, Startup, and Shutdown of CTF Containers

Instruction 1: Copy the challenge folder of the challenge to be tested from `project/challenge_resources_for_students` to any location on your machine. Change the directory to the copied folder and build the challenge Docker image (cf. Appendix D.1).

Expected Result 1: Building the image works flawlessly.

Instruction 2: Use `chmod` to set the permissions of `code_injection` and `code_reuse` folders to 777.

Instruction 3: Set up the challenge CTF Container (cf. Appendix D.1):

Expected Result 2: Setting up the container works flawlessly. The command prompt `bob@<random code>` appears.

Instruction 4: Use the command `exit` to leave the container:

Expected Result 3: Exiting the container works flawlessly

Instruction 5: Change to any other directory on your machine. Restart the container (cf. Appendix D.1):

Expected Result 4: Starting the container works flawlessly. The command prompt `bob@<random code>` appears.

TID-3: Exploit Development - Provided Resources and Tools, Installation of Tools, and Parallel Workflow

Instruction 1: Setup and start your local CTF Container (cf. Appendix D.1).

Instruction 2: Change directory to both challenge exercises and list their contents.

Expected Result 1: The directories contain all required resources (cf. Table 6.1).

Instruction 3: Use `touch` to create an empty file and set the permissions to `666` (The sudo password is `bob`).

Instruction 4: Modify the test file from your local machine with any text editor. Then use `cat` to display the contents of the test file in the CTF Container.

Expected Result 2: The modifications of the test file are the same in the CTF Container.

Instruction 5: Compile the vulnerable program with `gcc -stattic -g`. Then copy the challenge solution from `project/test_resources` into the CTF Container by copying it into the mounted Folder on your machine.

Instruction 6: Compile and run the exploit and verify that correct files have been created by the exploit (cf. Table 6.1. Then open the vulnerable program inside GDB

Expected Result 3: GDB starts successfully. The prompt `pwndbg>` appears.

Instruction 7: Run the command `start` inside Pwndbg.

Expected Result 4: Pwndbg functions flawlessly.

Instruction 8: Run the command `rop` inside Pwndbg.

Expected Result 5: The tool ROPgadget functions flawlessly. The last line of the output contains "Unique gadgets found: ...".

Instruction 9: Run the command `run` inside Pwndbg.

Expected Result 6: A new shell opens inside GDB. If the opened shell closes directly, the output at least contains "process ... is executing new program: ...".

Instruction 10: Use APT to update your package lists and install an arbitrary package (The sudo password is `bob`):

Expected Result 7: The installation of packages via APT works flawlessly.

TID-4: Exploit Transfer and Exploit Execution

Instruction 1: Setup and start the Target Containers (cf. Appendix D.1) and your local CTF Container (cf. Appendix D.1).

Instruction 2: Copy the challenge solution from `project/test_resources` into the CTF Container by copying it into the mounted Folder on your machine. Then compile it using gcc.

Instruction 3: Transfer your exploit to the Target Container for the challenge to be tested via SCP (cf. Table 6.1). Then use SSH to connect to your victim. The username and password are both `victim`.

Expected Result 1: The command prompt `victim@<random code>` appears. The file `exploit` was transferred successfully.

Instruction 4: Try to access the target file in `secret_files`:

Expected Result 2: Permission is denied.

Instruction 5: Use the command `groups victim` to list victim's groups.

Expected Result 3: `victim` is only part of his own group.

Instruction 6: Use the command `sudo -s` to try to open a root shell.

Expected Result 4: The command fails as `sudo` is not installed.

Instruction 7: Use the command `su` to try to switch to root user.

Expected Result 5: The root password is prompted. Since no root password has been assigned, root access is blocked.

Instruction 8: Run the exploit. Then run the vulnerable program.

Expected Result 6: A root command prompt appears.

Instruction 9: Access the target file and compare it to the content the corresponding flag file mounted in the challenge VM.

Expected Result 7: The access works flawlessly. The output contains the random flag and is identical to the mounted flag.

E.2. CTF Framework

TID-1: Setup, Startup, and Shutdown

Instruction 1: Reset the CTF framework (cf. Appendix D.2). Then replace the file `students.csv` file from `project/goehl/resources` with `students.csv` from `project/goehl/test_resources`.

Instruction 2: If existing delete all sub-folders of the challenges flag folders.

Instruction 3: Setup the CTF framework (cf. Appendix D.2).

Expected Result 1: The logs of the CTF framework are displayed. All desired challenges are set up. The `project/instance` folder contains the file `goehl.sqlite`. The challenges' flag folders contain a subfolder for each student in `students.csv`. Each subfolder contains a file `super_secret_file` with a unique string.

Instruction 4: Use `vagrant global-status` to list all Vagrant VMs.

Expected Result 2: The output contains a VM with the name *goehl_hbo* and *goehl_heauaf* and state "poweroff".

Instruction 5: Start the CTF framework (cf. Appendix D.2). Then open up a new terminal and use `vagrant global-status` to list all Vagrant VMs.

Expected Result 3: The output contains the VMs with the names *goehl_hbo* and *goehl_heauaf* and state "running".

Instruction 6: Hit `ctrl+c` once to shut down the CTF framework. Then open up a new terminal and use `vagrant global-status` to list all Vagrant VMs.

Expected Result 4: The output contains the VMs with the name *goehl_hbo* and *goehl_heauaf* and state "poweroff".

TID-2: Authentication and Page Access

Instruction 1: Reset the CTF framework (cf. Appendix D.2). Then replace the file `students.csv` file from `project/goehl/resources` with `students.csv` from `project/test_resources`.

Instruction 2: Setup and start the CTF framework (cf. Appendix D.2) and open the link for the CTF framework in a browser of your choice.

Expected Result 1: The *Login* page gets be displayed.

Instruction 3: Enter `23` for the matriculation number and click the `Login` button:

Expected Result 2: The login was not successful. The *Login* page along with the error message "The entered matriculation number is not registered!" gets displayed.

Instruction 4: Enter `asdf` for the matriculation number and click the `Login` button:

Expected Result 3: The Login was not successful. The *Login* page along with the error message “Please enter a valid matriculation number!” gets displayed.

Instruction 5: Enter `2683658` for the matriculation number and click the `Login` button:

Expected Result 4: The login was successful. The *Statistics* page along with the success message “Welcome Tim Cook!” gets displayed.

Instruction 6: Copy the URL and open it in a new tab. Keep the URL in your clipboard!

Expected Result 5: The *Statistics* page gets displayed.

Instruction 7: Click on `Logout` in the navigation bar in an arbitrary tab.

Expected Result 6: The *Login* page gets displayed.

Instruction 8: Open the URL from your clipboard in a new tab.

Expected Result 7: The *Statistics* page does *not* get displayed. Instead, the *Login* page gets displayed.

Instruction 9: Repeat the last three Instructions for the *Challenges* page and the *Heap Buffer Overflow* page as a representative.

Expected Result 8: Neither the *Challenges* page nor the *Heap Buffer Overflow* page can be accessed without a valid login.

TID-3: Navigation and Static Page Content Display

Instruction 1: Reset the CTF framework (cf. Appendix D.2). Then replace the file `students.csv` file from `project/goehl/resources` with `students.csv` from `project/test_resources`.

Instruction 2: Setup and start the CTF framework (cf. Appendix D.2) and open the link for the CTF framework in a browser of your choice.

Expected Result 1: The *Login* page gets be displayed. The top screen shows the title “Welcome to Ethical Hacking Lab”. The right side shows the logo of the Julius-Maximilians-University of Wuerzburg. The left side shows an input field for the matriculation number and a `Login` button.

Instruction 3: Log in with the following matriculation number `2683658`.

Expected Result 2: The *Statistics* page gets displayed. Under the page title, the user’s points are displayed. The left side shows a graph of the top 5 participants. The right side shows a leaderboard of the top 5 participants.

Instruction 4: Click on `Challenges` in the navigation bar.

Expected Result 3: The *Challenges* page gets displayed. All challenge subfolders in `project/goehl/resources` are displayed according to their `name.txt` file.

Instruction 5: Select the Challenge *Heap Buffer Overflow* as a representative.

Expected Result 4: The *Heap Buffer Overflow* page gets displayed. The information text is correctly formatted as HTML according to its `info_text.html` file. All flags from `flags.csv` are displayed.

Instruction 6: Open and close some flags by clicking on them. Also, click on some `Hint` buttons.

Expected Result 5: The flags open and close flawlessly. The flag question, flag format, points, and hints match `flags.csv`. The flag questions and hints are possibly correctly formatted as HTML.

Instruction 7: Click on a `Submit` button.

Expected Result 6: The error message "The submitted flag was empty!" gets displayed.

Instruction 8: Click on `Statistics` in the navigation bar.

Expected Result 7: The *Statistics* page gets displayed.

Instruction 9: Click on `EHL` or the logo in the navigation bar.

Expected Result 8: The *Challenges* page gets displayed.

TID-4: Flag Validation and Dynamic Page Content Display

Instruction 1: Reset the CTF framework (cf. Appendix D.2. Then replace the file `students.csv` file from `project/goehl/resources` with `students.csv` from `project/test_resources`.

Instruction 2: Setup and start the CTF framework (cf. Appendix D.2) and open the link for the CTF framework in two browsers of your choice and log in with two the matriculation numbers of your choice, which we refer to as A and B.

Instruction 3: Use A to check the *Statistics*-, *Challenge*-, and *Heap Buffer Overflow* page.

Expected Result 1: On the *Statistics* page every user has zero points both in the graph and in the leaderboard. On the *Challenges* page none of the challenges' flags are completed. On the *Heap Buffer Overflow* page all flags are uncompleted and open. In the information text for A the port number is different than for B.

Instruction 4: Use a database browser of your choice to open the file `project/instance/goehl.sqlite` and retrieve the solution for the end-flag of *Code Injection* for A. Then submit the flag as B.

Expected Result 2: The error message "The submitted flag was not correct!" gets displayed.

Instruction 5: Submit the flag as A.

Expected Result 3: The success message "The submitted flag was correct!" gets displayed. The submitted flag is now closed with a completed badge. The input field as well as the `Submit` and `Hint` buttons are no longer displayed.

Instruction 6: Click on `Challenges` in the navigation bar.

Expected Result 4: One flag of *Heap Buffer Overflow* is completed.

Instruction 7: Click on `Statistics` in the navigation bar.

Expected Result 5: Both the graph as well as the leaderboard have been updated correctly.

Instruction 8: Finish the *Heap Buffer Overflow* challenge by using the `goehl.sqlite` file. Then click on `Challenges` in the navigation bar.

Expected Result 6: The *Heap Buffer Overflow* challenge has a completed badge.

Instruction 9: Open the date and time settings on your machine and advance the date by one day. Then click on the `Statistics` in the navigation bar and reload the page.

Expected Result 7: Both the graph as well as the leaderboard have been updated correctly.

Instruction 10: Shut down the CTF framework by hitting `ctrl+c` in the terminal and advance the date by one day again. Then restart the CTF framework and reload the *Statistics* page.

Expected Result 8: Both the graph as well as the leaderboard have been updated correctly.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Place, 24. February 2023

....
(Johannes Schraud)