



ÉCOLE DES MINES DE SAINT-ÉTIENNE

INDUSTRIAL PROJECT
PENTESTS VIRTUAL AUTOMATION LABORATORY

Technical report - Industrial Project

Students (class 2021):

Tom BOURBON
Thierry CADINOT
Arnaud GRILLET
Tristan KREMSER
Vincent LALANNE
Hugo BESANÇON
Alexis BOUSSAROQUE
Éva PINLONG

Industrial tutor:

R. V.

Academic tutor:

Raphaël VIERA

Project made from 1 February 2023 to 23 June 2023

Report written on 22 February 2023

Contents

1	Acknowledgements	2
2	Glossary	3
3	Bibliography	4
4	Introduction	5
4.1	Aim of the project	5
5	Project progress	7
5.1	Working mode	7
5.2	Web application	7
5.3	Hackboxes	8
5.4	Orchestrator/hackbox communication	9
5.5	Detection of hackboxes	10
5.6	Global project	10
6	Technical production	12
6.1	Web application	12
6.1.1	Organisation of files	12
6.1.2	Drag & drop	14
6.1.3	Inputs and outputs	15
6.1.4	Indexing HTML elements	15
6.1.5	Connection in code	16
6.1.6	Visual connection	16
6.1.7	Write and display the database	19
6.1.8	Creating, editing, and deleting the database	22
6.1.9	Database transfer in JavaScript	24
6.1.10	Security and authentication	26
6.2	Hackboxes	26
6.3	Hackbox 1 - sniffing	26
6.3.1	Packet interception	26
6.3.2	Packet capture	27
6.3.3	JSON file analysis and report file generation	28
6.4	Hackbox 2 - HTTP frame analysis	29
6.5	Orchestrator / hackboxes communication	30
6.5.1	Design	30
6.5.2	Realisation	31
6.5.3	Improvement suggestions	32
6.6	Detecting hackboxes	33
7	Conclusion	34

1 Acknowledgements

If this project has gone well, it is thanks to the investment and efforts of dedicated teams, to whom we would like to extend our warmest thanks. First and foremost, we would like to thank Mr Raphaël Viera, who was our academic tutor. His expertise, availability and willingness to support us enabled us to complete the project successfully. Our thanks also go to Mx C. D., Mx M. F. and Mx R. V., engineers from A..., who supervised the project with great care and professionalism. We thoroughly enjoyed working with this committed and attentive team. Our sincere thanks go to them all. Thanks also to the University team responsible for monitoring the project. This project was made possible by their availability and commitment.

2 Glossary

Back-end: all the code concerning all the aspects of a web page that are invisible to the user, such as data management, conversation archiving, etc. The Python language is regularly used.

Front-end: all the code relating to the graphical appearance and user interface of a web page. The languages used to develop such code are typically HTML, CSS and JavaScript.

Framework: coherent set of structural software components used to create the foundations and outline of all or part of a piece of software, i.e. an architecture.

Endpoint: link to a page in a Flask web application (`www.test.com/endpoint`). Used to identify different pages on the same site.

Pentest: intrusion test to determine the susceptibility of a computer or electronic system to attack.

Brute force: method used in cryptanalysis to find a password or key. It involves testing, one by one, all the possible combinations.

Sniffing: technique for monitoring and capturing all the data packets on a network.

HTTP frames: blocks of data transmitted via an HTTP client-server communication protocol.

Raspberry Pi: electronic card equipped with an operating system, enabling programmes to be stored locally and executed remotely.

3 Bibliography

Documents relating to the web application:

Flask: <https://flask.palletsprojects.com/en/2.3.x/>

Model-View-Controller:

<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

Use and purpose of APIs: L. Qi, Q. He, F. Chen, X. Zhang, W. Dou and Q. Ni, "Data-Driven Web APIs Recommendation for Building Web Applications," in IEEE Transactions on Big Data, vol. 8, no. 3, pp. 685-698, 1 June 2022, doi: 10.1109/TB-DATA.2020.2975587.

"SSH: Understanding and Using this Protocol Better": <https://www.hostinger.fr/tutoriels/ssh-linux>

Documents relating to the use of Python libraries:

Paramiko: <https://www.paramiko.org/>

sqlite3: <https://docs.python.org/3/library/sqlite3.html>

4 Introduction

As part of our studies at ISMIN 2nd year, we are invited to carry out a group project over a period of four months. The aim of this project is to bring the students closer to the real working conditions encountered in a company, in particular by working with a customer, in this case A....

4.1 Aim of the project

The work required of the students is a proof of concept for the automation of attack scripts on embedded systems. As a proof of concept, the expected deliverable is not a fully finalised and secure web application, but a solid and functional working base that can be taken over by A.... engineers. The documentation work is therefore particularly important, as it will make it easier to take over the project.

More specifically, the aim of the project is to produce a web application with an intuitive graphical interface, enabling an attack scenario to be created by arranging *boxes* and linking them together. These *boxes* are actually *hackboxes*, which are hardware attack modules. These attack modules - or hackboxes - are linked by network or wire to any embedded system, and contain attack scripts that will be executed in the order and with the parameters chosen by the operator from the web application. Figure 1 shows a possible design for the web application attack scenario creation system. As this is a mock-up, the deliverable will obviously not be as aesthetically pleasing, but will still have to comply with a "no-code" requirement, i.e. an inexperienced user who is totally unfamiliar with computer systems will be able to create and launch an attack scenario.

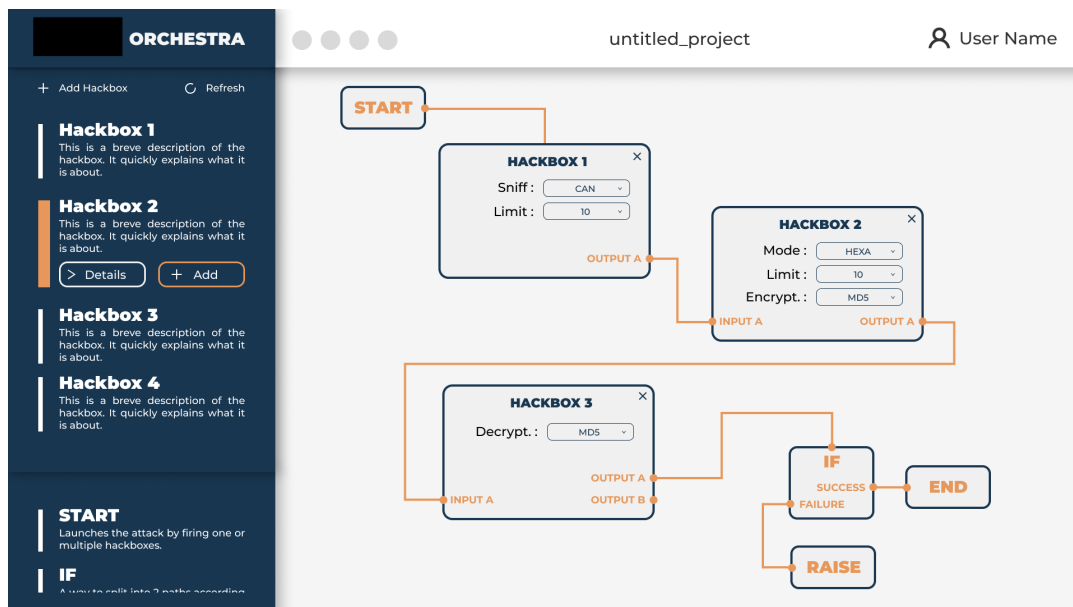


Figure 1: Web interface mock-up

The work requested by A... is limited to two mandatory hackboxes and one optional hackbox. The first two hackboxes will carry out a network analysis attack to glean information about the target system. The third hackbox will contain a brute force attack. In this way, it will be possible to deliver a functional project with considerable room

for improvement - it will in fact be A...’s engineers who will create more sophisticated hackboxes.

5 Project progress

5.1 Working mode

This industrial project was divided into two major phases: a design phase and a implementation phase. During the design phase, we drew up a set of specifications and a framework note for the project. This phase was also an opportunity to carry out the necessary documentary research and to deepen the theoretical knowledge we had accumulated. The implementation phase logically corresponds to the period when the project was actually carried out.

An agile methodology was adopted throughout the project. Firstly, an 'agile' specification was drawn up, in place of an SRS (Software Requirements Specifications), which is a type of specification specific to software development projects, particularly heavy and dense in information. In contrast, an agile specification contains less information, and is produced just in time to be used. This reduces the time needed to design the project, enabling a more efficient implementation phase. Generally speaking, we all wanted to minimise the learning curve so that we could get down to practical work as quickly as possible. The disadvantage of this method is that we develop very specific skills relating to our project, without having an overall theoretical view of the fields used. The advantage is that we can make faster progress on the project and maintain a desire to learn as the project progresses.

Our organisational strategy was therefore to hold regular meetings with our academic or industrial tutor, or with each other, in order to regularise progress and review any difficulties. This way of working enabled the team to quickly pool everyone's knowledge and develop a collective and shared vision of the project. Fortunately, this project is divided into several distinct parts and languages, allowing an intuitive division of labour. We divided up these different parts into small groups (between one and three people) so that we could make collective progress in parallel. There are three distinct parts on which work can be divided: the web application, the hackboxes and, finally, the interfacing of the two. In fact, the development of the first is independent of the development of the second: it is therefore possible to make parallel progress on both. A project manager has been assigned to each task, assisted by one or two support staff.

5.2 Web application

The web application is the central element of the project. It is an orchestrator with multiple roles. This web application must:

- enable one or more attack scenarios to be created and recorded
- interact with the hackbox database (create, read, edit, delete)
- ensure data communication between several hackboxes
- be secure via a system of users and authentication
- store user passwords securely

- be able to be used by a novice operator

The web application currently has an authentication system with user management. Passwords are encrypted and stored in such a way that they cannot be read by a third party. However, no administrator access to the platform has been implemented. It is also possible to create an attack scenario, but not to record it. This would require an additional table in the database, and this feature did not seem to us to be the most urgent to implement. Management of the hackbox database is well implemented and easy to use. Generally speaking, all the functions implemented in the web application are intuitive, so that any user can use it. Finally, communication between the hackboxes has not yet been fully completed at the time of writing, as this is the most complex requirement to satisfy. Overall, the basic functionalities are all present in the web application. Only one function is missing: the translation of the attack scenario created graphically by the user into data and instructions to be executed by the hackboxes.

5.3 Hackboxes

The web application, which we have named 'PenMaker', although central, must above all remain an orchestrator whose main role is to list the available attack modules and ensure communication between them. These hackboxes can have very different roles: theoretically, there could be as many hackboxes as there are different cyber attacks. As far as the hackboxes we use are concerned, the customer requires at least two: a sniffing hackbox and an HTTP frame analysis hackbox. The third hackbox, the brute force hackbox, is optional.

The first hackbox is a statistical network frame analyser that reconstructs flows and calculates statistical data. To do this, the hackbox reads communication datagrams and reconstructs them in the form of flows, enriching the information. The hackbox must be able to propose several categories of statistical values to be calculated and have parameters that the operator must be able to set before launching the attack (attack time, number of packets to be sniffed, etc.). In our case, we are counting the number of packets passing through the network and calculating consistency statistics between IP addresses and MAC addresses.

We were able to write a sniffing code that fully met the specification. The specification asks that a sniffing function be able to answer two questions: 1) How many packets pass through a given network interface per second? and 2) Is there any inconsistency in the correspondence between IP address and MAC address? The 'Hackboxes' section of this report sets out to explain how such a feature has been implemented. The functionality developed answers these two questions, as well as giving the user a great deal of scope for customising the sniffing script.

The output data from the sniffing hackbox is processed data, as shown in figure 2.

The second hackbox is an HTTP frame analyser. During HTTP protocol communication, the data transmitted is described in headers and stored in the body of the transmission. It is then possible to retrieve data from the headers to gather information about the transmitted data. As far as the specifications are concerned, no specifications beyond a functional code are required. At present, the code in this hackbox is functional

```

1 Number of packets captured per second : 12.091699407840844
2 The MAC 52:54:00:12:35:02 may have 2 different IP. The IP of this MAC was 34.246.155.13 in packet number 4 but was 192.168.130.33 in packet number 1.0
3 The MAC 52:54:00:12:35:02 may have 2 different IP. The IP of this MAC was 185.86.168.138 in packet number 15 but was 34.246.155.13 in packet number 2.0
4 The MAC 52:54:00:12:35:02 may have 2 different IP. The IP of this MAC was 142.251.37.174 in packet number 72 but was 185.86.168.138 in packet number 3.0
5
6 Packet number 0 :
7 Packet interface : eth0
8 Packet protocol : DNS
9 Packet time : Jun 15, 2023 10:58:37.572124786 EDT
10 Packet IP source address : 10.0.2.15
11 Packet MAC source address : 08:00:27:95:bd:54
12 Packet IP destination address : 192.168.130.33
13 Packet MAC destination address : 52:54:00:12:35:02
14
15
16 Packet number 1 :
17 Packet interface : eth0
18 Packet protocol : DNS
19 Packet time : Jun 15, 2023 10:58:37.624951003 EDT
20 Packet IP source address : 192.168.130.33
21 Packet MAC source address : 52:54:00:12:35:02
22 Packet IP destination address : 10.0.2.15
23 Packet MAC destination address : 08:00:27:95:bd:54
24
25
26 Packet number 2 :
27 Packet interface : eth0
28 Packet protocol : DNS
29 Packet time : Jun 15, 2023 10:58:37.625460382 EDT
30 Packet IP source address : 10.0.2.15
31 Packet MAC source address : 08:00:27:95:bd:54
32 Packet IP destination address : 192.168.130.33
33 Packet MAC destination address : 52:54:00:12:35:02

```

Figure 2: Output of the sniffing process

for HTTP frames; it analyses HTTP-type network packets and returns the information contained in them. An example of the analysis of 3 HTTP packets is shown in figure 3. The most important improvement would be to be able to analyse HTTPS frames and therefore to be able to decrypt the data encrypted in the packet secured by the HTTPS protocol.

```

Source IP: 192.168.194.233
Destination IP: 128.59.105.24
Request Full URI: http://www.columbia.edu/~fdc/picture-of-something.jpg
Request Method: GET
Request Version: HTTP/1.1
Host: www.columbia.edu
User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0 Safari/537.36
Source IP: 192.168.194.233
Destination IP: 128.59.105.24
Request Full URI: http://www.columbia.edu/~fdc/sample.html
Request Method: GET
Request Version: HTTP/1.1
Host: www.columbia.edu
User Agent: curl/8.0.1
Source IP: 192.168.194.233
Destination IP: 216.251.32.98
Request Full URI: http://help.websiteos.com/websiteos/example_of_a_simple_html_page.htm
Request Method: GET
Request Version: HTTP/1.1
Host: help.websiteos.com
User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0 Safari/537.36

```

Figure 3: HTTP frame analysis result

5.4 Orchestrator/hackbox communication

This section details the expectations and the work carried out concerning the communication protocol enabling the transit of information between the web application and the hackboxes, and vice versa. This communication is essential for the attack scenario created from the orchestrator to run smoothly. Essentially, two types of data are exchanged. The first type corresponds to the execution of the attack scripts embedded on the Raspberry: the type of attack desired by the user is recognised and transmitted to the corresponding Raspberry. The second type of data exchanged between the orchestrator and the Raspberry are files. These files are used to summarise the progress of the attack, and are necessary for the user to obtain the useful information gathered during the attack.

The specification for communication between the orchestrator and the hackboxes does not have any strict clauses, meaning that the only expectation for this part is that it works. We were free to choose which tools to use, and these choices are detailed in the Technical Implementation section. At present, communication between the two parties is via the SSH protocol. We were able to initialise the protocol and send the data we wanted. In this respect, the specifications have been met. However, communication is not perfect, particularly in terms of security. A user needs a user name and password to access a hackbox, and these are stored in clear text in a file, which is of course a practice to be avoided.

5.5 Detection of hackboxes

In order for the orchestrator to be able to use any future hackbox created by the company, we had to implement in the orchestrator the possibility of adding a hackbox to the database by entering its characteristics. The most useful element of the hackbox is its MAC address. The MAC address corresponds to the hardware address of the hackbox, which remains invariant unlike IP addresses, which can change. However, we need the IP address of the hackboxes to open an SSH connection to communicate and carry out the scenarios. To do this, we created a python code that uses ARP requests to link the MAC addresses to their IP addresses on the network.

5.6 Global project

Generally speaking, the project required the use of a wide variety of technologies, far removed from those taught at ISMIN. For the creation of the web application, the languages commonly used are HTML, CSS and JavaScript, which are not taught at the University. This part also requires knowledge of software architecture, which we didn't have. For hackboxes, network analysis algorithms are also new. Similarly, we had never interfaced hardware and software before, except for a robot project, nor had we created any microarchitecture, etc. This project therefore required a significant adaptation and self-training phase. Luckily, our academic tutor was competent in these areas, so we were able to organise regular meetings to make progress together.

One of the major difficulties encountered during the design phase of the project was the wide range of possible technologies. As the project was complex and covered many technical areas that were new to us, it took us a few weeks to fully grasp the project and to be able to make proposals on the technologies to be used.

The advantage of working in agile mode was therefore to optimise this learning curve in order to gain practical skills as quickly as possible. Indeed, if we had trained ourselves in all the languages used before the start of the project, it would have taken a considerable amount of time and we wouldn't have been able to get as far with the project. The disadvantage of this method is that we have to look for the precise information we need, and we don't have enough expertise in the technologies to develop more appropriate solutions ourselves.

Despite having to deal with an extensive and complex subject that required considerable training time, we were able to adapt and produce work that we feel is satisfactory

given the time and resources at our disposal. The broad outlines of the project were completed and, more importantly, a functional deliverable was produced, in accordance with the principles of the agile method. Although certain functionalities were too specific to be implemented on time, we remain satisfied with the final deliverable.

6 Technical production

6.1 Web application

An initial development phase focused solely on the front-end (HTML, CSS, JavaScript) and enabled the integration of functions for creating attack scenarios using drag & drop. In simple terms, HTML and CSS are used to display and style text, while JavaScript is used to manage animations and the movement of boxes. Unsuccessful attempts to integrate a JavaScript-only back-end convinced us to use an existing framework. In fact, it is vital for the web application to orchestrate the back-end with the front-end in a fluid manner. Given the small size of the database and the number of potential users of the application, we decided to use the open-source micro framework for web development in Python called Flask. Its advantage over its main competitor Django is its ease of use and light weight.

6.1.1 Organisation of files

After migrating the existing code and adapting it to Flask, we were able to integrate and manage the database efficiently. To organise the files in the most professional way, we used a popular software architecture pattern for graphical user interfaces, the Model-View-Controller (MVC) pattern. In this pattern, the Model designates the database, the View designates the graphical interface, visible to the operator, and the Controller designates the module that handles user actions, modifies the data in the model and the view.

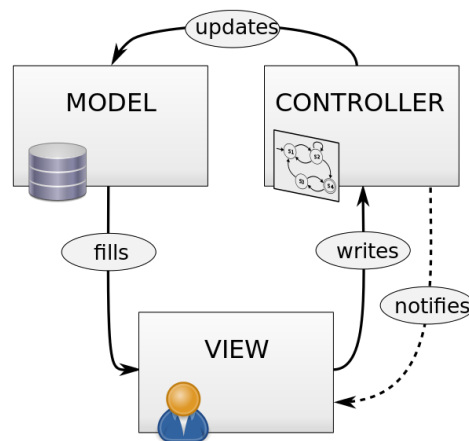


Figure 4: Model-View-Controller

From a global point of view, the tree structure is presented in figure 5. The main folder is called PenMaker, and contains the "instance" and "project" folders as well as the "README.md" and "res.txt" files. These last two files are the user manual for the application and the result of a hackbox attack, stored as a text file. The 'instance' folder contains the database and the Python script used to create it. The "project" folder contains all the files used to process the web application. The Python files and the HTML files are stored in the "template" subfolder. The "static" sub-folder stores the immutable

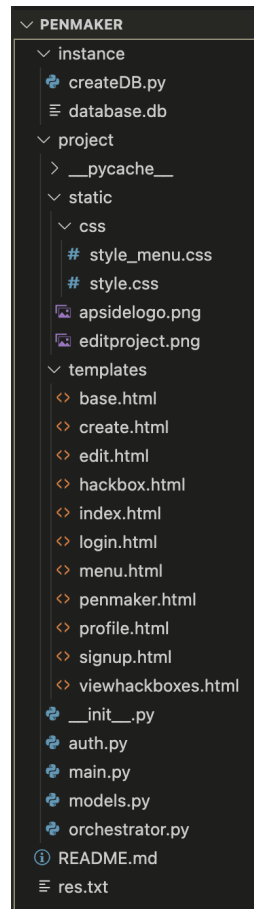


Figure 5: File tree

files, i.e. the images and CSS files. Finally, the "__pycache__" sub-folder stores the session caches, generated automatically by Flask.

Flask allows the use of Python to organise pages written in HTML, CSS and JavaScript, and to implement Python functions in these same pages. In our case, we split the Python files as follows:

- `init.py`: to initialise the application itself, import the database and initialise the user manager
- `auth.py`: lists all the endpoints relating to the authentication system (`/login`, `/signup`, and `/logout`) as well as their respective functions
- `main.py`: contains the default endpoint (`/`) and the `/profile` endpoint, which displays no relevant information
- `orchestrator.py`: lists all endpoints relating to hackboxes and attack scenarios
- `models.py`: this is the file in which the `User` and `Hackbox` classes are created

The `User` and `Hackbox` classes are created using the following constructor:

```
1 class User(UserMixin, db.Model): # UserMixin provides default
2     # implementations for the methods that Flask-Login expects user
```

```

3     # objects to have
4     # primary keys are required by SQLAlchemy
5     id = db.Column(db.Integer, primary_key=True)
6     email = db.Column(db.String(100), unique=True)
7     password = db.Column(db.String(100))
8     name = db.Column(db.String(1000))
9
10
11 class Hackbox(db.Model):
12     id = db.Column(db.Integer, primary_key=True)
13     function = db.Column(db.String(100), nullable=False)
14     inputs = db.Column(db.String(100), nullable=False)
15     outputs = db.Column(db.String(100), nullable=False)
16     mac_address = db.Column(db.String(17), nullable=False)
17
18     def __repr__(self): # how the object is printed
19         return f"Hackbox('{self.function}',
20             '{self.inputs}', '{self.outputs}', '{self.mac_address}')"

```

6.1.2 Drag & drop

Developing a window in which it is possible to create an attack scenario by displaying boxes, moving them around and linking them together was one of the most time-consuming parts of the project. Indeed, in order to achieve a simple, intuitive and no-code interface, a great deal of programming effort is required. The result achieved to date is shown in Figure 6.

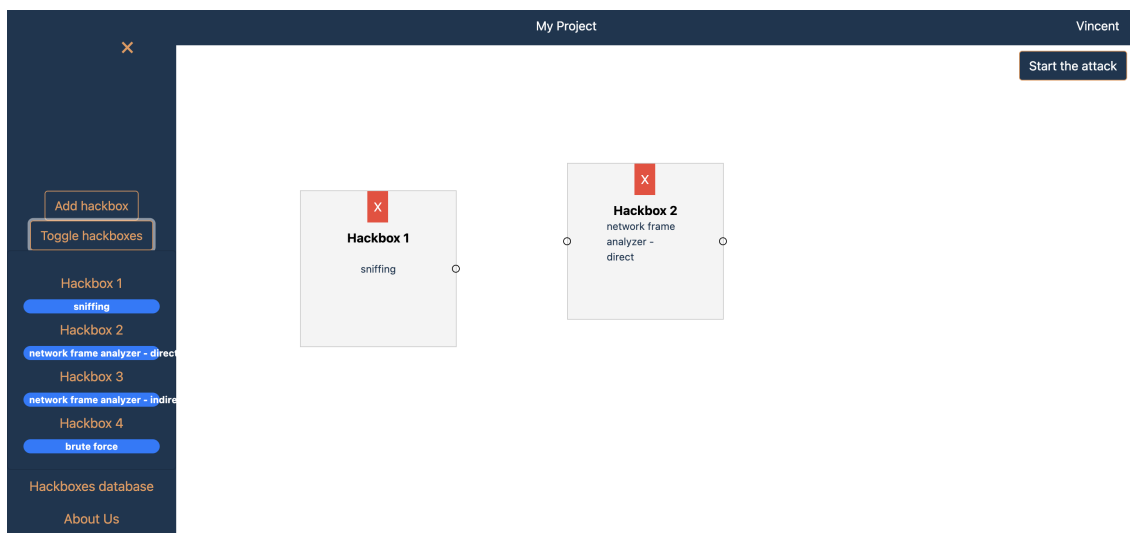


Figure 6: Attack scenario creation window

Hackboxes are represented by boxes called **draggable-box** in the code. We've added event listeners, which allow an action to be carried out as soon as a trigger action is performed. Here, the trigger action is a click on a **draggable-box**, which will modify the coordinates of the box. To do this, we've initialised several variables to track the location

and state of the `draggable-box`: `isDragging` to indicate whether the box is currently dragging, `currentX` and `currentY` to store the coordinates of the current position of the box, `initialX` and `initialY` to store the coordinates of the initial position of the box at the time of click, and `xOffset` and `yOffset` to store the offsets between the current position and the initial position.

A "mousedown" EventListener is added to the `draggable-box`. When this event is triggered, the initial position of the box is recorded by subtracting the coordinates `xOffset` and `yOffset` from the click position. If the target of the event is the box itself, the `isDragging` variable is set to `true` to indicate that the box is being moved. A "mouseup" event listener (left-click release) is added to the box. When this event is triggered, the `isDragging` variable is set to `false` to indicate that the box has stopped moving.

We've also added a mousemove event listener. When this event is triggered, we check whether the box is being moved (`isDragging` is `true`). If it is, it prevents the default behaviour of the event to avoid any unwanted interaction. The coordinates `currentX` and `currentY` are then updated by subtracting the coordinates `initialX` and `initialY` from the current mouse position (`e.clientX` and `e.clientY`). The offsets `xOffset` and `yOffset` are updated with the new coordinates `currentX` and `currentY`.

The function `setTranslate` is called to apply a CSS transformation to the box and move it visually using the `transform` property. This function sets the `transform` property of the `el` element (the box) using the `xPos` and `yPos` values to move the box horizontally and vertically. This is done using the method `translate3d(xPos, yPos, 0)`.

6.1.3 Inputs and outputs

We then added elements to the starting class `draggable-box` to add inputs and outputs which will be used to connect the hackboxes together. To do this, we divided the `draggable-box` into 4 sub-classes: `top`, `bottom`, `left` and `right` in order to make the placement of the inputs and outputs on the sides of the box simpler. So on the "left" side of the box we have an input container, which will contain the inputs, and on the "right" side we have an output container.

When we generate a hackbox we can, by retrieving the number of inputs and outputs in the database for the hackbox in question, use 2 `for` loops to create the inputs and outputs and place them in the appropriate containers. Finally, we used the `space-around` argument to the `justify-content` property to place our inputs and outputs correctly on the hackbox.

6.1.4 Indexing HTML elements

Once these elements had been added to our `draggable-box`, we modified our JavaScript classes for inputs and outputs to implement functionalities enabling an input to be connected to an output both visually and in the code. To do this, we had to add an index to each `draggable-box`, input and output, to distinguish them from each other.

Each box therefore has its own index, called `boxIndex`, and a counter `count` `db` is incremented each time a `draggable-box` is generated and is used as the index for the box

generated. When generating inputs and outputs, we use the for loop to modify the index of the generated input or output.

We then added an attribute called **entry-index** or **exit-index** to the JavaScript classes for inputs and outputs. This attribute is written in the form **a.b** where "a" is the index of the box and "b" is the index of the input or output.

Finally, we added the arguments **connected** and **connectedTo** to the input and output classes, these two arguments being initialised to **false** and **null** respectively, so they can be used to find out whether the input or output is connected and to have the index **entry-index** or **exit-index** of the input or output to which it is connected.

6.1.5 Connection in code

To connect an output to an input, we added a click listener to the outputs when they were generated. To do this, we used the function **addEventListener()**. So, each time an exit that is not connected is clicked, the **confirmationclick1** function is executed. This confirms that it is the first click and adds a click listener to the document. **confirmationclick2**. If this is the case, the **connectElements()** function is executed, thus removing the click listener from the document.

The function **connectElements()** takes as arguments an input and an output and will connect them by changing their attribute **connected** to **true** and modifying **connectedTo** of the entry and exit so that it takes the value of the index of the output or the input to which it is connected. The 2 elements corresponding to the desired entry and exit have now been connected in the code, so they must also be connected visually by drawing an arrow.

6.1.6 Visual connection

To draw the lines, we've created a line controller that connects elements on an HTML canvas. To begin with, we initialise several variables:

- **lines**: array storing all the lines created using the **LineController** object. Each line is represented by an object containing various properties, such as the start and end nodes, colour, thickness, etc.
- **canvas**: element used as a canvas for drawing lines.
- **ctx**: 2D context of the canvas
- **ele1, ele2**: These are variables used to store the HTML elements corresponding to the start and end nodes of a specific line.
- **linemap**: This is an empty object which will be used to map the combinations of start and end nodes to their corresponding indices in the **lines** array. This will allow quick search for a line based on its nodes.
- **left, right**: These are empty objects which will be used to store the offsets of the left and right elements respectively. These offsets will be used to calculate the coordinates of the start and end points of the lines.

- **dax, day:** These are variables which store the horizontal and vertical differences between the coordinates of the start and end nodes respectively.
- **dangle:** This is a variable which stores the angle of the line in relation to the horizontal axis, calculated using the horizontal and vertical differences (**dax** and **day**).
- **rightx, righty, leftx, lefty:** These are variables which store the coordinates of the start and end points.

We then use a `connect()` function to generate the line controller by initialising the line controller and defining internal functions for managing the lines. The `connect()` function works as follows:

The function starts by initialising a variable `me` with the value `this`. This stores a reference to the current instance of the `LineController` object, which will be used within the internal functions.

Next, it initialises the `canvas` object by defining its ID, width and height so that it occupies the entire browser window. The `canvas` object is then added to the body of the HTML document using `document.body.appendChild(canvas)`.

The variable `ctx` is initialised with the 2D context of the canvas, which will be used to draw the lines. The `drawLine` function is defined. It takes as a parameter `option` which represents the options of the line to be drawn, the line is then added to the `lines` array using `linemap` to store the index of the line in the array. Finally, the `draw` function is called to draw the line.

The `draw` function takes `option` as a parameter. Within this function, several steps are performed to draw the line between the two elements:

- Preliminary check to ensure that the left and right nodes are defined.
- Determines the line colour (here, it is set to `black`).
- Search for HTML elements corresponding to the left and right nodes using their indexes.
- Calculates the co-ordinates of the centres of the left and right elements.
- Calculates the direction and angle between the two centres.
- Calculates the co-ordinates of the start (`leftx, lefty`) and end (`rightx, righty`) points of the line.
- Storage of the coordinates of the start and end points in the `left` and `right` objects.
- Adjust the y coordinates to align the line correctly with the elements.
- Draw the line on the canvas using `beginPath()`, `moveTo()`, `lineTo()`, `lineWidth`, `strokeStyle` and `stroke()`.
- Added "mouseup" event listeners on the `draggable-box` affected by the line to redraw the arrows when they are moved.

We then define 2 functions to manage the lines: `changecolor()` and `redrawLines()`. `changecolor()` is a function which allows changing the colour of a line and `redrawLines()` allows the deletion of all the lines on the canvas and redraw them one by one using the `draw()` function.

Finally, we define the `removeLine()` function. It takes a parameter `exit` which represents the output element of the line to be deleted. This function traverses the `lines` array to find the matching line by comparing the identifiers of the output nodes with the `exit-Index` attribute of the output element. When a match is found, it performs the following actions:

- Retrieve the values of the exit and entry nodes of the line.
- Reset the `connected` and `connectedTo` properties of the entry and exit elements.
- Use the `changecolor` function to make the line white.
- Removal of the line from the array `lines` and `linemap`.
- Use the `redrawLines` function to redraw the lines without the deleted line.

We had to use the `changecolor` function to make the line white because we found that the first call to the `redrawLines` function did not erase the line, in fact the line was only erased once another line had been deleted, i.e. on the second call to the `removeLines` function. We therefore decided to make the line 'invisible' until it was deleted. This is only a visual problem, as we have checked that when a line is deleted the input and output are no longer connected by their `connected` and `connectedTo` arguments.

We also use the `redrawLines()` function to manage the movement of boxes. Using the event listeners generated in the `draw()` function, each time a box connected by a line is moved, we call the `redrawLines()` function to redraw the line after the box has been moved.

Finally, we have created a function which allows us to obtain the order in which the hackbox attack scenarios will be executed. This function is called `getExecutionOrder()`, and is a recursive function which takes as its parameter a `startBox` (generally the one connected to the start button), representing the start box from which the execution order will be determined, and which traverses the boxes using their connections.

The function initialises two empty arrays: `executionOrder` and `stack`. `executionOrder` will be used to store the execution order of the boxes, while `stack` will be used as a stack to browse the boxes. The function starts by adding the `startBox` to the `stack`. This box will be the starting point for the execution order search.

Next, the function enters a `while` loop which runs until the `stack` is empty. On each iteration of the loop, the function removes the last box from the stack using `stack.pop()` and stores it in the variable `currentBox`. This means that we explore the boxes starting with the last one added. The `currentBox` is then added to the end of the `executionOrder` array, representing the current execution order.

Next, the function runs through the exits of the `currentBox` using a `for` loop. For each exit, the function checks whether it is connected to another box. If it is, it retrieves the index of the connected box from `exit.connectedTo`, using `split('.')` to extract the index before the dot. It then accesses this connected box in the `draggableBoxes` array using the adjusted index of 1 (because indices in `connectedTo` start at 1 whereas indices in the array start at 0). The connected box is then added to the `stack` to be explored later.

Once all the outputs of the `currentBox` have been processed, the `while` loop continues until the `stack` is empty. Finally, the function returns the array `executionOrder` which contains the execution order of the boxes.

This is an example of an algorithm that can be used to obtain an execution order and is not necessarily the most efficient.

6.1.7 Write and display the database

To create the database of hackboxes and users, we chose to write the Python script below. When this script is run, a file `database.db` is created in the `instance` folder. It is not readable by a text editor application; a specialist software or an application to see what the database contains would be needed.

```
1  import sqlite3
2
3      # Create a connection to the database
4      conn = sqlite3.connect('database.db')
5
6      # Create the 'users' table
7      conn.execute('''CREATE TABLE user
8                      (id INTEGER PRIMARY KEY AUTOINCREMENT,
9                       email TEXT,
10                      name TEXT,
11                      password TEXT);''')
12
13      # Create the 'hackboxes' table
14      conn.execute('''CREATE TABLE hackbox
15                      (id INTEGER PRIMARY KEY AUTOINCREMENT,
16                       function TEXT,
17                       inputs INTEGER,
18                       outputs INTEGER,
19                       mac_address TEXT);''')
20
21      # initialise the 'hackboxes' table with 4 hackboxes
22      conn.execute("INSERT INTO hackbox VALUES (1, 'sniffing', 0, 1,
23          'E4:5F:01:F5:EF:15')")
24      conn.execute("INSERT INTO hackbox VALUES (2, 'network frame
25          analyzer - direct', 1, 1, 'E4:5F:01:F5:E5:66')")
26      conn.execute("INSERT INTO hackbox VALUES (3, 'network frame
```

```
27 analyzer - indirect', 1, 1, '22:33:44:55:66:77')")
28 conn.execute("INSERT INTO hackbox VALUES (4, 'brute force', 2, 1,
29 '33:44:55:66:77:88')")
30
31 # Commit the changes and close the connection
32 conn.commit()
33 conn.close()
```

To integrate the database with the web application, we can use the sqlite3 module or its more advanced and powerful version, SQLAlchemy. In our case, we decided to initialise the database with SQLAlchemy, then perform the necessary manipulations with sqlite3. This way, we only need to import the SQLAlchemy module once, when we initialise the application (see code below).

```
1 from flask_sqlalchemy import SQLAlchemy
2
3 app = Flask(__name__)
4 app.config['SECRET_KEY'] = os.urandom(32)
5 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'
6 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
7 db = SQLAlchemy(app)
```

In order to be able to display the DB data directly on a page and thus have an overview of the hackboxes available, it is necessary to write suitable code. The page dedicated to this can be accessed via the endpoint /viewhackboxes and is shown in figure 7. As the menu bar is managed by a separate file common to all the application's pages, its code does not appear as is in the viewhackboxes.html file. In fact, this code file only contains a few lines:

- One line for the page title
- A line to display the number of hackboxes available and a line to display the hackbox creation button
- A for loop to display the information for each hackbox, i.e. its identifier and its function (its identifier is a clickable link which redirects to the specific hackbox page, where more information is available).
- The for loop also contains the "Edit" button, which redirects to the hackbox-specific edit page, and the "Delete" button, which removes the hackbox from the DB (the hackbox is indeed deleted, not just removed from display)

The corresponding code is as follows:

```
1 {% block content %}
2
3 <h1>{% block title %} List of all hackboxes {% endblock %}</h1>
4
```

```

5 <p class="font-italic">{{ hackboxes|length }} hackboxes available.
6     <a type="button" class="btn btn-outline-success"
7       href="{{ url_for('orchestrator.create') }}">Create hackbox</a>
8 </p>
9
10
11 <!-- Hackbox id -->
12 {% for hackbox in hackboxes %}
13 <hr>
14 <!-- URL for the specific hackbox page -->
15 <a href="{{ url_for('orchestrator.hackbox', hackbox_id=hackbox.id) }}">
16     <h2>Hackbox {{ hackbox.id }}</h2>
17 </a>
18
19 <!-- Hackbox function -->
20 <span class="badge badge-pill badge-primary">{{ hackbox.function }}</span>
21
22 <!-- Edit button -->
23 <a href="{{ url_for('orchestrator.edit', hackbox_id=hackbox['id']) }}">
24     <span class="btn btn-outline-warning">Edit</span>
25 </a>
26
27 <!-- Delete button -->
28 <form method="POST"
29     action="{{ url_for('orchestrator.delete',
30         hackbox_id=hackbox.id) }}">
31     <input type="submit" value="Delete" class="btn btn-outline-danger"
32         onclick="return confirm('Are you sure you want
33         to delete this hackbox?')">
34 </form>
35
36 {% endfor %} {% endblock %}

```

In this code, the "Delete" button is not a button as such, but rather a form. In fact, if it were a simple button, it would redirect to the `/hackbox_id/delete` endpoint, whose sole functionality would be to delete the hackbox whose id appears in the endpoint. However, by sounding this way, any malicious user could enter this URL and delete all the hackboxes. Putting in a form helps to guard against this security issue, as the URL never becomes accessible as such, but only via the form. Anyone entering the hackbox 1 deletion endpoint directly into the search bar, for example, would be redirected to the page shown in figure 8. However, this solution does not provide complete protection against malicious use: if a hacker were to use a penetration testing tool such as BurpSuite, it would be possible to intercept the request and modify it in order to delete hackboxes. The only more effective way of protecting against this attack (whether via URL or form - GET or POST) is to authenticate requests (for example via the implementation of user accounts and the use of session objects, etc.).

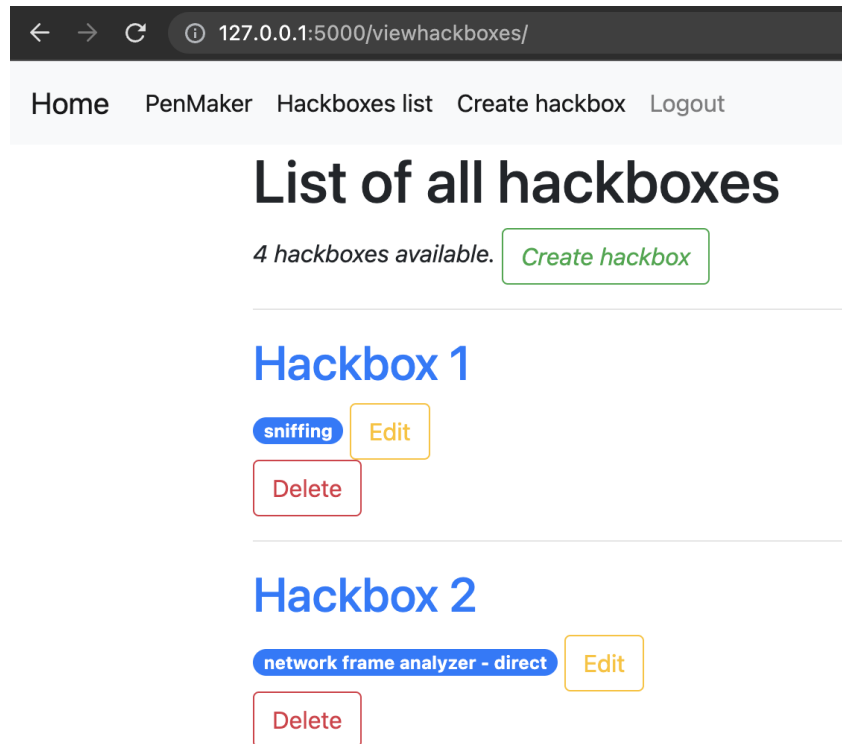


Figure 7: Hackboxes inventory page



Figure 8: Attempting to delete a hackbox using a direct URL

6.1.8 Creating, editing, and deleting the database

There are four possible operations to perform on a database, often summarised by the acronym CRUD: create, read, update, delete. In the previous sub-section, we detailed how the database was initialised and displayed (read). In this sub-section, we will detail the lines of code used to modify, create and delete an element in the database. These operations are performed in the `/hackbox_id/edit/`, `/create` and `/hackbox_id/delete` endpoints respectively.

To create a hackbox, a form must be filled in by the user. The fields requested by this form are: the function of the hackbox, its number of inputs, its number of outputs, and its MAC address. The id is not requested, as it is a value which is auto-incremented each time a hackbox is added. Figure 9 shows the page seen by the user when creating a new hackbox.

The code corresponding to the page is a form of the type:

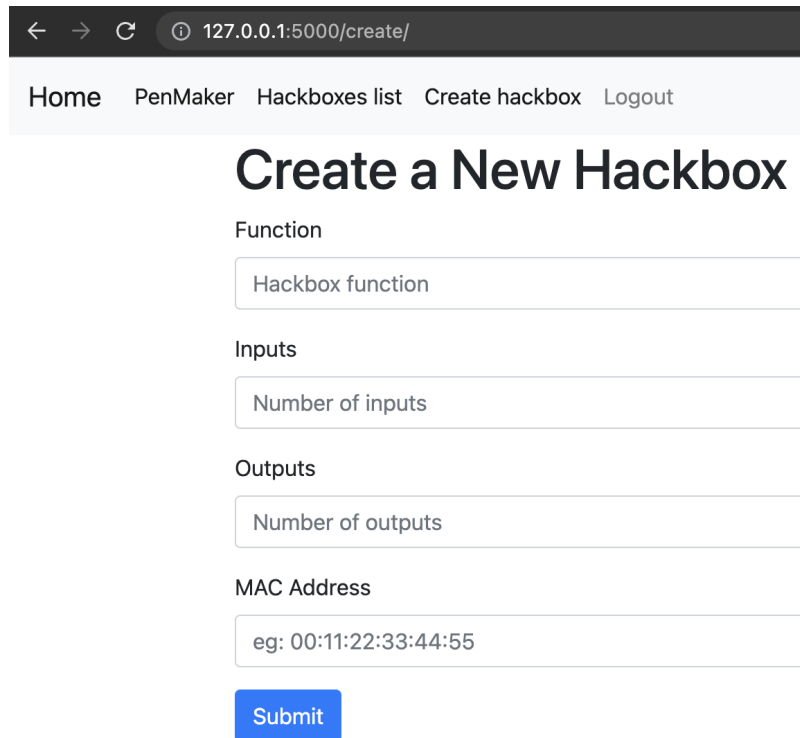


Figure 9: Hackbox creation page

```

1 <div class="form-group">
2     <label for="function">Function</label>
3     <input type="text" name="function"
4         placeholder="Hackbox function" class="form-control" required>
5     </input>
6 </div>

```

This part corresponds to the function field, the other fields being similar, except for the type, which becomes `number` for inputs and outputs, and `submit` for the form submit button. Once the form has been submitted, it is processed by the application's back-end, i.e. by the Python file `orchestrator.py`:

```

1 @orchestrator.route('/create/', methods=('GET', 'POST'))
2 @login_required # Only accessible if the user is logged in
3 def create():
4     if request.method == 'POST': # If the form is submitted
5         function = request.form['function']
6         inputs = request.form['inputs']
7         outputs = request.form['outputs']
8         mac_address = request.form['mac_address']
9         hackbox = Hackbox(function=function,
10                            inputs=inputs,
11                            outputs=outputs,
12                            mac_address=mac_address) # Creation of
13                                                    # the hackbox

```



```

14         db.session.add(hackbox) # Add the hackbox to the database
15         db.session.commit() # Commit the changes
16         return redirect(url_for('orchestrator.viewhackboxes'))
17         return render_template('create.html', name=current_user.name)

```

In this code fragment, the data entered by the user is retrieved when the "Submit" button is pressed, then stored in the corresponding variables. These variables are then used to initialise a new instance of the Hackbox class.

For the "hackbox editing" part, the principle is the same: the user fills in a form, whose data is extracted and then used to overwrite the data in the existing hackbox. If the user enters a URL with a non-existent hackbox identifier, they are redirected to a page displaying a 404 error, "Not Found". Finally, for the "hackbox deletion" part, we have seen that it is possible to secure this operation by making the URL directly inaccessible. When the delete form is submitted via the "Delete" button, two lines of code in particular are called:

```

1  hackbox = Hackbox.query.get_or_404(hackbox_id)
2  db.session.delete(hackbox)

```

These lines are used to isolate the hackbox chosen by the user (if the user selects a hackbox with a non-existent identifier, a 404 error is displayed), and then to delete it from the database. Note that this operation does not change the identifier of existing hackboxes. The HTTP methods corresponding to CRUD operations on a database are POST, GET, PUT and DELETE respectively. However, only the POST and GET methods are really necessary to perform CRUD operations.

6.1.9 Database transfer in JavaScript

In our case, the difficulties were in transferring the variables we were working with from one language to another, in particular from JavaScript to Python and from Python to JavaScript. In the last case in particular, we were faced with the impossibility of passing a DB variable from Python to JavaScript. To solve this problem, we created a 'mirror' of the database in the form of a Python dictionary. This mirror is automatically generated from the database, so that a change to the original DB also changes the mirror. We then made this mirror accessible to the /data/fullhackboxes endpoint (figure 10). The code used to generate this mirror is as follows:

```

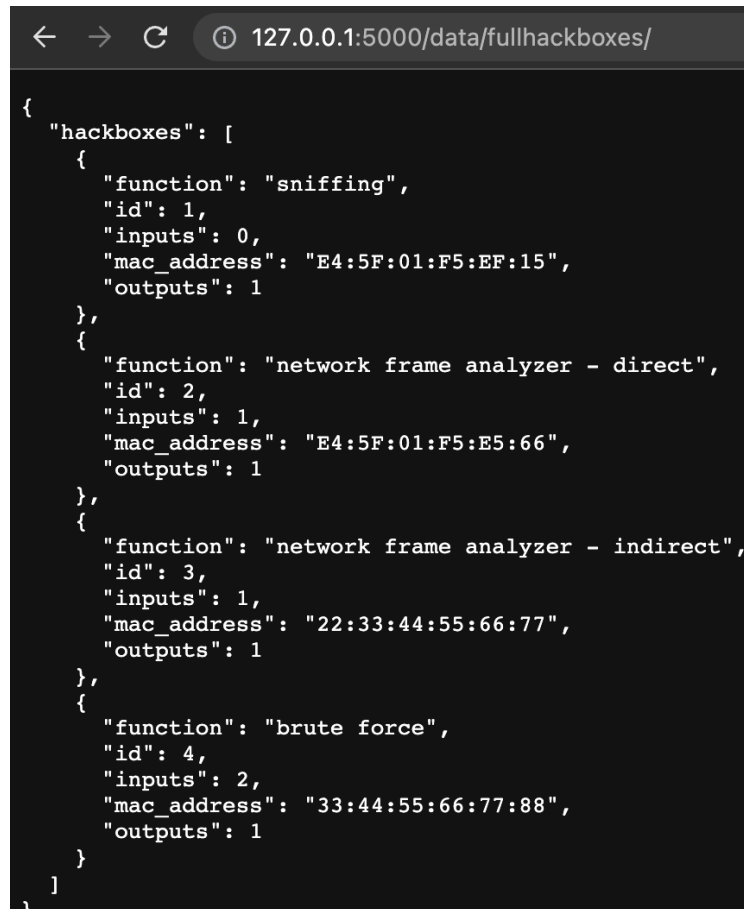
1  @orchestrator.route('/data/fullhackboxes/')
2  def fullDataHackboxes():
3      hackboxes = {"hackboxes": []}
4      # Connection to the database
5      conn = sqlite3.connect('instance/database.db')
6      c = conn.cursor() # Creation of the cursor
7      c.execute("SELECT * FROM hackbox") # Select all hackboxes
8      rows = c.fetchall() # Fetch of the results
9      for row in rows:
10         hackboxes["hackboxes"].append({ # Add the hackbox to the list
11             "id": row[0],

```

```

12         "function": row[1],
13         "inputs": row[2],
14         "outputs": row[3],
15         "mac_address": row[4]
16     })
17 conn.close()
18 return hackboxes

```



```

{
  "hackboxes": [
    {
      "function": "sniffing",
      "id": 1,
      "inputs": 0,
      "mac_address": "E4:5F:01:F5:EF:15",
      "outputs": 1
    },
    {
      "function": "network frame analyzer - direct",
      "id": 2,
      "inputs": 1,
      "mac_address": "E4:5F:01:F5:E5:66",
      "outputs": 1
    },
    {
      "function": "network frame analyzer - indirect",
      "id": 3,
      "inputs": 1,
      "mac_address": "22:33:44:55:66:77",
      "outputs": 1
    },
    {
      "function": "brute force",
      "id": 4,
      "inputs": 2,
      "mac_address": "33:44:55:66:77:88",
      "outputs": 1
    }
  ]
}

```

Figure 10: Database in Python dictionary format

We then wrote some JavaScript code to access this endpoint using the `fetch` function. The method used is similar to that used when working with programming interfaces, also known as APIs. In this way, we were able to use the database directly in JavaScript, which was a crucial step for the project, as the specification stipulated that the web application had to be able to interact with the database.

```

1 fetch("/data/fullhackboxes/")
2   .then((response) => response.json())
3   .then((data) => {
4     * manipulations avec le miroir de la BDD * }

```

6.1.10 Security and authentication

In order to meet the security requirements for users, we have implemented a basic authentication system, based on HTML forms and methods integrated into Flask. Users are stored in the database in the form of email addresses, names and passwords. In order to guarantee the integrity of passwords, they are not stored unencrypted in the database. Before being stored, they are encrypted (or 'hashed') using the sha256 algorithm (see code fragment below). In this way, a malicious user with access to the database cannot extract passwords, only unusable strings of characters. However, passwords remain vulnerable to a dictionary or brute force attack.

```
1 new_user = User(email=email, name=name,  
2 password=generate_password_hash(password, method='sha256'))
```

If a malicious user compares a user's encrypted password with the list of the most commonly used hashed passwords in the world (azerty, password, 123456, etc.), it is possible to find a match and therefore to connect to a user's account without knowing their clear password. To avoid this problem, one effective method is to add a random character string to the beginning of the password when a user registers. However, we have not found a method to ensure rigorous implementation of this solution, as generating and saving a random string for each user is a complex process. An alternative solution was to add a fixed random character string, generated beforehand, and common to all users. Another security measure would have been to make the password field less flexible, by forcing the user to use at least one number, one capital letter, one special character and a minimum length. This is an option that can be easily implemented, but for the time of development we preferred to be able to choose a one-character password, to save time.

6.2 Hackboxes

As far as the technical aspects are concerned, we decided to use Raspberry Pi 4 model B electronic boards as hackboxes. We made this choice because the Raspberry Pi meets all the requirements of our hackboxes, with a wi-fi or ethernet internet connection, and remote communications (using SSH or ARP protocol) possible. What is more, it is one of the most widely used electronic boards in the sector, which makes it easier to implement new hackboxes later on. Finally, the company already has attack modules implemented in Raspberry Pi.

So we configured the hackboxes, firstly by installing the Raspberry Pi OS, which is a version of Linux adapted for Raspberry computers. This OS allows remote manipulation of the Raspberry, as if it were a small computer. We then configured the SSH port to be used for communication with the web application, and finally activated the Internet.

6.3 Hackbox 1 - sniffing

6.3.1 Packet interception

The packet interception phase uses Tshark, the tool behind the popular Wireshark network analysis software. We first thought of using Wireshark to intercept packets, but it is not

easily manipulated by a script. On the contrary, it is possible to use Tshark to intercept packets via command lines, i.e. via a script. It is therefore preferable to use Tshark rather than Wireshark. It has also been thought to use Python libraries for this packet interception phase instead of Tshark. However, no library with sufficiently comprehensive capabilities has been found to replace Tshark. For this reason, it is preferable to use a script with Tshark rather than using Python code. It was therefore decided to use a Bash script that uses Tshark.

The language used to write this script is Bash. However, it is a Python code that automatically writes the Bash script. There are two reasons for this. Firstly, it was decided that the orchestrator would use mainly Python for its back-end. However, the original plan was for the orchestrator to do most of the calculations, leaving as little as possible to the hackbox. This idea therefore required using Python to write a sniffing script. However, this idea was abandoned because it was not of significant interest. As a result, it is now the hackbox itself that manages the sniffing, so the orchestrator has fewer calculations to make. This simplifies integration, as it allows the orchestrator to easily manipulate the sniffing script without having to enter too low a level. The second reason is simply to simplify the programming of the sniffing functionality: the code can be written, updated and debugged more quickly. The following architecture has therefore been chosen:

1. The orchestrator executes a Python file on the Hackbox via SSH
2. The HACKBOX captures packets, analyses them and prepares a report file
3. The orchestrator retrieves the report file via SSH

The sniffing code that may be present on hackboxes during a basic operating demonstration is based on the functions of the code that will be sent to A..... The functions developed have been designed so that a code based on them would be written as follows:

- initialisation of variables
- call a Bash script writing function using Tshark
- execute the Bash script
- call a file processing function (JSON) to highlight the information of interest.

6.3.2 Packet capture

The functions have been developed with the aim of leaving plenty of scope for customising the sniffing script. The user chooses the network interface on which to perform packet capture. In particular, they can choose the **any** interface to request packet capture on all interfaces. This capture only ends when one of the two stop conditions is met (unless the user wishes to end the capture early):

- the number of packets captured is equal to the limit set by the user
- the capture time is equal to the limit set by the user

The user must give at least 1 stop condition.

Packets can be filtered. The function can be told to consider only packets coming from a certain IP or MAC address; going to a certain IP or MAC address; using a certain protocol, or using a protocol from a list of authorised protocols; coming from a certain input port or going to a certain output port. Capture is therefore highly customisable. The packet data is stored in a file with a JSON extension. This file can be used by an analysis function included in the code supplied to A..... In addition, a .pcap file is generated, summarising packet transmissions.

6.3.3 JSON file analysis and report file generation

The JSON file produced by a packet capture function is analysed. The information extracted from this file is useful for answering the two questions set out in the specifications: 1) How many packets are circulating per second, on what interface and with what protocol? and 2) Is there any inconsistency in the IP/MAC address mapping? In other words, are there any packet exchanges that might suggest that the same MAC address has several IP addresses, or that the same IP address has several MAC addresses?

To answer question 1), a function measures the time that elapses between the start and the end of the capture (i.e. the duration of the execution of the script, so, barring a negligible error, the duration of the capture, given the content of the scripts written). This function divides the number of packets captured by this duration to return the answer to question 1). Here too, it is possible to customise the answer to this question. Of all the packets whose information is contained in the JSON file, it is possible to take only certain packets into account. For example, it is possible to ask for the number of packets per second only by considering packets characterised by the DNS protocol. It is possible to filter by protocol, source or destination IP/MAC address and interface.

To answer question 2), a function draws up an inventory of the IP/MAC correspondence. Any inconsistencies are flagged up. This option offers the same customisation as the previous one: it is possible to filter by source or destination IP/MAC address, protocol or interface. A report file is then generated. This contains the results on the consistency of the IP address/MAC address correspondence. For a basic demonstration, the hackbox may contain a code that calls up a special function that adds the report from question 1) to the report file that must answer question 2). In this way, both questions are answered in the same report file, to simplify the work of the orchestrator for such a demonstration. For more details, A... will have the code in its entirety as well as documentation to explain how to use it to incorporate it into future projects.

The basic principle of IP/MAC correspondence checking is as follows:

```
Let L be an empty list
For any packet:(
    If source IP does not exist in any pair of L:(
        If source MAC exists in a pair of L:(
            Alert of the problem))
    If destination IP does not exist in any pair of L:(
        If destination MAC exists in an L pair:(
```

```

        Alert of the problem))
    If source MAC does not exist in any L pair:(
        If source IP exists in an L pair:(
            Alert of the problem))
    If destination MAC does not exist in any L pair:(
        If destination IP exists in an L:(
            Alert about the problem))
    If the pair (source IP, source MAC) is not in L, add it
    If the (destination IP, destination MAC) pair is not in L, add it.
)

```

6.4 Hackbox 2 - HTTP frame analysis

Several Python libraries were considered, starting with the `socket` module, which allows HTTP requests to be sent but not analysed. We opted for Scapy, which does a good job of retrieving network frames and then analysing them. Unfortunately, we encountered problems when implementing it in the hackbox, due to insufficient permissions and closed ports. We therefore decided to use the Pyshark library and restrict ourselves to analysing frames rather than retrieving them. This hackbox retrieves the frame file in pcap format and extracts all the information it contains about HTTP requests.

An HTTP (Hypertext Transfer Protocol) packet is a data unit transporting information between a client and a web server. It can contain information about a request or a response. An HTTP request generally contains the following elements:

- Request line: includes the HTTP method (GET, POST, PUT, DELETE, etc.), the URI (Uniform Resource Identifier) of the requested resource and the HTTP protocol version (IPv4 or IPv6).
- Headers: additional information such as authentication headers, content headers or cache control headers.
- Body: optional and used to send additional data with the request, typically for POST or PUT requests.

An HTTP response generally contains the following elements:

- Status line: includes the HTTP protocol version, a status code and an associated status message.
- Headers: additional information such as content headers, cache headers or cookie headers.
- Body: contains the data returned by the server, such as images, files or HTML content.

The code below displays the information contained in an HTTP packet in clear text. Note that the pyshark library allows us to break down the network packet into several

distinct pieces of information that we can then display. To do this, we insert the .pcap file containing all the packets retrieved previously into a pyshark file. The file will then identify each of the packets it contains and apply our HTTP packet display function. This function will first make sure that it is actually processing an HTTP packet before displaying the information.

```
1  import pyshark
2
3  def http_packet(packet):
4      try:
5          if packet.http:
6              print("Source IP:", packet.ip.src)
7              print("Destination IP:", packet.ip.dst)
8              print("Request Full URI:", packet.http.request_full_uri)
9              print("Request Method:", packet.http.request_method)
10             print("Request Version:", packet.http.request_version)
11             print("Host:", packet.http.host)
12             print("User Agent:", packet.http.user_agent)
13             print("Response Code:", packet.http.response_code)
14             print("Response Phrase:", packet.http.response_phrase)
15             print("Server:", packet.http.server)
16             print("/n")
17         except AttributeError:
18             pass
19
20 capture = pyshark.FileCapture("/home/arnaud/Document
21 /intercepted_traffic.pcap")
22 capture.apply_on_packets(http_packet, packet_count=10)
```

This hackbox is destined to evolve into an HTTPS frame analyser, as we have not yet succeeded in processing this type of protocol. HTTPS traffic is encrypted via SSL/TLS, so we need to be able to decrypt this traffic in order to analyse it. To do this one needs access to the encryption key, which can be acquired in several ways. The simplest and most accessible is to be the administrator of the HTTPS server, where it is then possible to extract and use the key for decryption. Another possibility is if one has access to a certification authority (CA) that has issued a certificate for the HTTPS server one wishes to analyse. It is then possible to deploy an SSL/TLS proxy such as "mitmproxy" or "Charles Proxy" to intercept HTTPS traffic and decrypt the SSL/TLS session key on the fly. This SSL/TLS proxy simulates the HTTPS server, making it possible to intercept and decrypt HTTPS traffic. Or use third-party tools such as "ssldump", "tcpdump", "Wireshark" or "Fiddler"; these can be used to capture SSL/TLS traffic between the client and the HTTPS server, and extract the SSL/TLS encryption key from the capture.

6.5 Orchestrator / hackboxes communication

6.5.1 Design

This sub-section looks at the theoretical operation of the attack scenario and, more specifically, the way in which the different systems - the orchestrator and the hackboxes - ex-

change data. It is assumed that a coherent attack scenario has already been created by a user from the orchestrator.

Assuming such a scenario has been created, the user can execute the attack scenario via a start button on the same page as the one used to create the scenario. Pressing this button executes an initial script contained in the site's back-end. This script gathers the information required for the attack scenario via the database. In particular, this script identifies the various hackboxes involved in the scenario, the links between them, i.e. which hackbox is linked to which hackbox, the type of attack carried out and their respective MAC addresses. A second script, carrying out the corresponding attack, is run on the associated Raspberry. This script provides an attack report file, enabling the user to obtain information about how the attack was carried out. This report file is then uploaded to the orchestrator. At the end of the attack process, the user is automatically redirected to a page summarising the information relating to the attack.

Two scenarios are of interest today. A simple scenario: a sniffing hackbox, with the result accessible from the orchestrator, and a more complex scenario: a sniffing hackbox sends its result to a network frame analyser hackbox, which itself sends its result back to the orchestrator.

The various data exchanges pass through an SSH tunnel. We didn't have a protocol imposed by the A... team, so we initially relied on APIs to carry out this communication. However, as this method of communication is not secure and is fairly complex, we chose to use the SSH protocol, which already offers secure communication between two systems connected to the same network. The SSH protocol is used to open a communication tunnel between these systems, in this case the host PC on which the user is using the web application and a hackbox.

As its name suggests, the orchestrator is the only master, so all data exchange goes through it. There is no direct communication between two hackboxes. This choice was made in agreement with the A.... team.

6.5.2 Realisation

The first IT object linked to the execution of the scenario is the "Start the attack" button. When it is clicked, it acts as a form that activates the execution of a complex Python function that extracts the data needed for the attack from the Attack scenario creation window (figure 6), and connects to the hackboxes to launch the attack(s). This Python function has not yet been finalised: extracting the useful data from the scenario creation window involves transferring a number of variables from JavaScript to Python, which we have not yet managed to do at the time of writing.

Two Python libraries are used in this code: sqlite3 and Paramiko. The sqlite3 library for processing the database, and the Paramiko library for simply setting up an SSH channel via Python. Firstly, this function accesses the database.db database and extracts all the information relating to the hackboxes. As a reminder, a hackbox contains the following fields: identifier, function, linked inputs, linked outputs and MAC address. Once extracted, it is stored in the form of a table containing all the information needed

to carry out an attack. This table can then be processed to retrace the scenario created by the user.

First we look at the number of hackboxes contained in the scenario. In our case, assuming we have created a coherent project, this can only be a sniffing hackbox or a frame analyser not linked to an upstream sniffing hackbox, and therefore taking a PCAP file as input.

Next, check whether the hackbox is a sniffing or analysis hackbox. We access the hackbox functions in the table extracted from the database. Index 0 of this table contains information about the first hackbox, including its function. This information is itself contained in an array. The function of a hackbox is placed in the second index of this sub-array, because it represents the second field of a hackbox in the database.

Sniffing: This is where a sniffing function is executed. The first step is to connect to the SSH tunnel. Before doing so, the user of the web application must ensure that the hackbox(es) being used are connected to the same Wi-Fi network. The first step is to establish a connection between the computer and the hackbox. To do this, we use the `connect` method from the Paramiko library. This method requires connection information such as the username and password used when configuring the SSH tunnel on the Raspberry.

The second step is to run the attack script on the identified Raspberry. `exec_command` method is used to execute a Linux command on another device, with the command passing through the SSH channel. The absolute path (on the Raspberry Pi) of the text file summarising the attack is also specified. It is necessary to know whether the attack is complete before downloading the file. The `recv_exit_status` method in the Paramiko library is used to ensure that a command has been completed, in this case ensuring that the attack launched on the Raspberry has been completed. Finally, the `get` method in the same library is used to download the balance file contained in the hackbox to the user's computer.

Frame analyser: If a frame analyser is used, the same procedure is applied for detecting the type of attack as for connecting to the hackbox and executing the attack script. What changes afterwards is only the addition of a PCAP file, containing a network frame, if the frame analysis hackbox is not placed downstream of a sniffing hackbox. If this analyser hackbox follows a sniffing hackbox, the sniffing and then the frame analysis are carried out, the sniffing PCAP file being downloaded to the orchestrator and then sent to the hackbox carrying out the analysis. The analysis report is also downloaded via SSH to the orchestrator.

6.5.3 Improvement suggestions

The communication method designed in this way is functional but could be improved for ergonomic reasons. Firstly, the SSH channel connection identifier and password are hard-coded in the code. Obviously, this is not good practice. We've tried hashing this information, but it's not easy because of the way the `connect` function works, which doesn't allow a hashed password to be compared with its unhashed counterpart.

In addition, it might be interesting to create a user-friendly interface for adding new attacks. This could be done via a new tab called "Configure a new attack". This tab would allow the creation of a hackbox corresponding to a new attack. In particular, this tab would make it possible to deposit code on a hackbox without intervening directly with the hackbox's Linux interface; everything would be done via the orchestrator.

6.6 Detecting hackboxes

Before we can communicate with the hackboxes, we need to detect them so that we can extend our project to future hackboxes and know which ones can be used when we launch the script. To do this, we used the hackbox database and, more specifically, all the MAC addresses present in it. We stored them all in a list of addresses that we will use later. We then created a Python code using the `subprocess` library, which will allow us to execute command lines in the PC terminal via our Python code. Using this library, we can open the ARP table on our PC and retrieve all the pairs of IP addresses and MAC addresses present on the network. We then test each MAC address to check whether it is present in the list of MAC addresses in the database, and if it is present we add this pair of IP/MAC addresses to our list of matching IP/MAC addresses. This code is placed before the script is run, so we know which IP addresses we need for SSH communication. This method does not restrict us to the hackboxes we have implemented, but is open to new hackboxes on the sole condition that they are added to the database.

We have encountered a few problems with ARP table responses. Depending on the operating system, the response format is different. We therefore added a test of the OS used via the Python library `platform` and the method `platform.system()`, which allows us to differentiate the analysis of responses depending on the operating system. One problem that persisted for a considerable part of the project was ARP responses, as hardware connected to a network only responds to ARP requests if it has previously communicated with the sender of the request. However, we believe that with the ARP table, this should no longer happen.

7 Conclusion

In conclusion, the industrial project carried out over several months as a group was an enriching experience both technically and in terms of teamwork.

From a technical point of view, this project enabled us to deepen our knowledge and skills in different areas. We gained an in-depth understanding of web technologies such as HTML, CSS and JavaScript, popular frameworks such as Flask, and ways of communicating between hardware and software. We also explored advanced concepts such as database management, web application security and performance optimisation. Throughout the process, we had to face technical challenges and find appropriate solutions to overcome them, which enabled us to broaden our expertise and develop our ability to solve complex problems.

However, the project wasn't just about the technical aspect. Working in a team of 6 students was a valuable experience that taught us many lessons about collaboration and communication. We had to learn how to coordinate our efforts, distribute tasks evenly and meet deadlines. Managing time and resources is a crucial skill if the project is to be completed on time. We also had to learn how to resolve conflicts and make collective decisions, taking into account the different perspectives and opinions within the team. These teamwork skills are invaluable and will stand us in good stead in our future professional careers.

Finally, this project also allowed us to develop our project management skills. We had to draw up a detailed plan, monitor the progress of the project and adjust our strategies when necessary. We learned how to manage priorities, anticipate potential problems and find effective solutions. These project management skills will be invaluable in the future, as we will undoubtedly be working as part of a team on complex, multidisciplinary projects. All in all, we're proud of the final result we've achieved together, and we're convinced that the lessons we've learned throughout this project will serve us well in our future careers.