



Fulchiron Samy
Internship Report

Design of an HTTP Honeytrap Server and Integration of a Large Language Model (LLM)

From the 3th of March 2025 to the 3st of August 2025

Internship tutor: Urban Sedlar
Referent teacher: Cedric Chauviere



August 2025

Résumé :

Lors de mon stage de 4e année, j'ai eu l'opportunité d'intégrer le laboratoire de la faculté d'électronique de Ljubljana, où j'ai mené, avec l'accompagnement de mon tuteur, un projet centré sur la cybersécurité. L'objectif principal était de comprendre, analyser et concevoir un honeypot, c'est-à-dire un serveur HTTP destiné à attirer les attaquants afin d'étudier leurs comportements. Le but était de développer un serveur fonctionnel capable de simuler de manière crédible un service réel, en intégrant un modèle de langage de grande taille (LLM) pour générer dynamiquement le contenu et répondre aux requêtes reçues. Le projet s'est structuré autour de plusieurs axes : l'acquisition des notions fondamentales en cybersécurité, la mise en place du serveur HTTP, puis l'intégration du LLM pour produire des réponses réalistes. Ce rapport présente les différentes phases du projet, les difficultés rencontrées ainsi que les solutions apportées.

Mots-clés : honeypot, serveur HTTP, requêtes, protocole, template HTML

Abstract :

During my fourth-year internship, I had the opportunity to join the electronics faculty laboratory in Ljubljana, where I carried out, with the guidance of my supervisor, a cybersecurity-focused project. The main objective was to understand, analyze, and design a honeypot — an HTTP server intended to attract attackers to study their behaviors. The goal was to develop a functional server capable of realistically simulating a real service by integrating a large language model (LLM) to dynamically generate content and respond to incoming requests. The project was structured around several key areas: learning fundamental cybersecurity concepts, setting up the HTTP server, and integrating the LLM to produce realistic responses. This report details the different stages of the project, the challenges encountered, and the solutions implemented.

Keywords : honeypot, HTTP server, request, protocol, template HTML

REMERCIEMENTS

This report is the result of a five-month internship that I had the pleasure and privilege to complete at the Faculty of Electrical Engineering, University of Ljubljana.

I would like to express my sincere gratitude to my tutor, Urban Sedlar, for his continuous guidance, supervision, and valuable advice throughout the internship. I also want to thank all the colleagues with whom I shared office space for their kindness and support. I am deeply grateful for the time and effort that everyone on the team dedicated to training me and helping me develop new skills. I am leaving with a wealth of knowledge and experience.

I would like to thank my academic tutor, Cédric Chauvière, for his assistance during the internship.

TABLE DE MATIÈRES

TABLE DES ILLUSTRATIONS	5
GLOSSARY	6
INTRODUCTION	7
1 INTRODUCTION TO CYBERSECURITY	8
1.1 PRESENTATION OF THE LABORATORY	8
1.2 ENVIRONMENTAL IMPACT AND SUSTAINABLE PRACTICES	8
1.3 ANALYSIS OF AN INITIAL DATASET	9
1.4 IP ADDRESS	10
1.5 NETWORK PORT	11
1.6 URL	ERREUR ! SIGNET NON DEFINI.
1.7 METHOD	14
1.8 TIMESTAMP	15
1.9 HEADERS	16
2 CREATION OF A FIRST SERVER AND GENERATION OF AN HTML PAGE USING LLM	17
2.1 FILE STRUCTURE	17
2.2 AUTHENTICATION AND PASSWORD PROTECTION	21
2.3 DYNAMIC RESPONSE GENERATION USING A LLM	22
2.4 FINAL PROMPTS	25
2.5 CHALLENGES ENCOUNTERED	26
3 IMPLEMENTING A FAKE DIRECTORY LISTING SERVICE	28
3.1 VISUAL INTERFACE	28
3.2 GENERATING CONTENT USING THE 404 ERROR	29
3.3 SAVING THE GENERATED CONTENT	35
3.4 MONITORY SERVER	35
CONCLUSION	38
ANNEXES	39

TABLE DES ILLUSTRATIONS

Figure 1: Honeypots Dataset.....	10
Figure 2: Map of IP Address Locations.....	11
Figure 3: Output of the netstat Command.....	12
Figure 4: Occurrence of network port.....	12
Figure 5: List of sensitive keywords and file extensions.....	13
Figure 6 : Dataset with searching keywords and file extensions.....	13
Figure 7: Clustering usins Levenshtein distance.....	14
Figure 8: Ocurrence of HTTP methods.....	14
Figure 9: Timestamp pic.....	15
Figure 10: File structure.....	17
Figure 11: Requests_log.jsonl.....	18
Figure 12: "/" route of the first server.....	19
Figure 13: 404 error page.....	20
Figure 14: "/login" route and inspection.....	20
Figure 15: Favicon.ico.....	21
Figure 16: Users route.....	21
Figure 17: Admin route.....	22
Figure 18: Surveillance camera page with simple prompt.....	23
Figure 19: Shodan camera.....	24
Figure 20: Extract of principal elements of the HTML page.....	25
Figure 21: Requests sent by the page.....	26
Figure 22: "/directory/" route.....	28
Figure 23: 404 error handler.....	30
Figure 24: Prompt for generate json list of files and folders.....	30
Figure 25: File extensions.....	31
Figure 26: "/directory/docs/tests/" route.....	31
Figure 27: Prompt for generate html page.....	32
Figure 28: «/directory/docs/index.html» route.....	32
Figure 29: «/directory/index.html» route.....	33
Figure 30: "/directory/data/temp2/temp/" route.....	34
Figure 31: Monitory server, received HTTP requests.....	36
Figure 32: Monitory server, directory history.....	37
Figure 33: Monitory server, files history.....	37

GLOSSARY

- **Honeypot:** A deliberately vulnerable or deceptive computer system designed to attract and analyze malicious activity.
- **Server:** A machine or application that receives and responds to requests from clients over a network.
- **Request:** A message sent by a client (a browser or automated tool) to a server, asking for a resource or service.
- **Protocol:** A set of rules that define how data is transmitted and received over a network.
- **HTML Template:** A model file containing HTML code and dynamic placeholders to generate web pages with server-side content.
- **Middleware:** A function that sits between the request and the response in a web application.
- **Handler:** A function or method responsible for processing a specific type of request. It receives the request, performs logic, and returns a response.

INTRODUCTION

In a context where data is becoming increasingly central and users spend several hours online every day, cybersecurity challenges are more critical than ever. The number of attacks continues to rise, as do the complexity and effectiveness of the techniques used by attackers. It is therefore essential to understand their methods in order to protect our servers, which are particularly vulnerable targets.

Among the most relevant approaches to studying such malicious behavior is the honeypot: a deliberately exposed server designed to simulate a real service with the goal of attracting attackers and recording their interactions. During my five-month internship at the laboratory of the Faculty of Electrical Engineering in Ljubljana, I had the opportunity to design such a server.

The project began with a study of key cybersecurity concepts and terminology. It then progressed to the implementation of a realistic honeypot. An innovative aspect of the project was the integration of an LLM (Large Language Model) to automatically generate responses to incoming requests.

This report shows the various stages of the project: the analysis of data from an existing honeypot, the progressive development of an interactive server, and finally, the design of a honeypot capable of dynamically simulating an Apache-style directory listing service using artificial intelligence.

.

1 INTRODUCTION TO CYBERSECURITY

1.1 PRESENTATION OF THE LABORATORY

I had the opportunity to spend five months at the Laboratory for Telecommunications and Multimedia (LTFE), affiliated with the Faculty of Electrical Engineering at the University of Ljubljana, Slovenia. LTFE is a leading center for applied research and technological innovation in Europe, working closely with the Multimedia Laboratory.

The laboratory conducts advanced research in various fields such as cybersecurity, the Internet of Things (IoT), 5G networks, artificial intelligence, multimedia systems, and cloud technologies. It is equipped with a Testcenter for large-scale testing of digital networks and services, and a Cyberlab, where I carried out my work, dedicated to security testing, vulnerability analysis, and cybersecurity research.

LTFE also plays a central role in numerous European and international projects, addressing key topics such as sustainable agriculture, smart mobility in rural areas, e-health, and digital inclusion for people with disabilities. In addition, the laboratory is actively involved in both academic and professional training, offering continuing education programs and hosting many internship opportunities for students.

1.2 ENVIRONMENTAL IMPACT AND SUSTAINABLE PRACTICES

As part of this project, the environmental impact of my activities was relatively limited. From a technical perspective, I primarily worked on my personal laptop, which has a moderate energy consumption. The project did not require training complex artificial intelligence models. However, I frequently relied on a language model (LLM) hosted on the laboratory's servers. These requests can be energy-intensive due to the infrastructure needed to operate such models. Aware of this impact, we made efforts to limit the number of queries by ensuring they were relevant and by grouping our needs whenever possible.

While the project itself was not specifically focused on sustainability, it indirectly contributes to a more responsible use of digital resources by enabling the detection and prevention of malicious attacks. These attacks, often carried out by bots, can generate thousands or even millions of requests, resulting in unnecessary bandwidth usage and server resource consumption.

On a personal level, I adopted an environmentally friendly lifestyle throughout my stay. I primarily used a bicycle for commuting, including trips to the university, and chose buses for travel within the country and to neighboring countries. I avoided air travel for the entire duration of my internship. Additionally, I shared a large house with Romain, Nicolas, and about fifteen other Erasmus students, which allowed us to pool resources: shared car rentals, waste sorting, composting, and more responsible collective consumption. Although I traveled more than usual, I believe my carbon footprint remained reasonable.

1.3 ANALYSIS OF AN INITIAL DATASET

The first part of the project involved becoming familiar with the technical terms used in cybersecurity, particularly those related to honeypots. A honeypot is a system designed to simulate various types of computer servers (such as web servers, databases, APIs) in order to attract attackers and analyze their behavior. In this project, we focused specifically on HTTP servers, as the ultimate goal was to build an HTTP server.

To begin the internship, we analyzed a dataset containing 254,658 HTTP requests collected by a honeypot hosted on the laboratory's servers. This dataset served as a foundation for becoming familiar with key cybersecurity concepts and terminology.

A request is a message sent by a client to a server in order to request a resource, transmit information, or trigger a specific action. For instance, when performing an online search, the browser sends an HTTP request to the web server hosting the page, asking for the data related to the search. The server processes the request and responds with the corresponding results. This client-server model is fundamental to the functioning of the Internet and is used hundred of times per day by every user.

It is important to highlight that for a request to be correctly understood and processed, it must follow a protocol, that is, a defined set of rules governing the format and steps of communication. Several protocols exist, each serving a specific purpose:

- HTTP/HTTPS: protocols used for web communication.
- FTP: protocol for file transfer.
- SMTP, IMAP: protocols for sending and reading emails.
- SSH: protocol for secure remote connections between a client and a server.

The dataset studied contains HTTP requests that comply with the HTTP protocol rules, received by the honeypot. It therefore helps to understand the main elements and concepts needed in cybersecurity, particularly in the context of HTTP servers.

Each HTTP request in the dataset is composed of several essential elements that allow us to identify its nature, its objective, and determine whether it is benign or potentially malicious.

	src_ip	dst_port	method	headers	url \
122	000.000.000	5984	GET	{'connection': 'close', 'user-agent': 'Go-http...	http://000.000.000:5984/query?q=SHOW+DIAGNOSTICS
123	000.000.000	5984	GET	{'connection': 'close', 'user-agent': 'Go-http...	http://000.000.000:5984/v2/_catalog

	timestamp
122	2025-02-24 13:36:29.738055+00:00
123	2025-02-24 13:36:29.740169+00:00

(254658, 6)

Figure 1: Honeypots Dataset

Each line corresponds to a request. We decided to retain six elements, which we will detail later, that seem the most important for identifying the nature of the request. The IP addresses in the columns `src_ip` and in the URLs have been intentionally replaced by 000.000.000 in this report for confidentiality reasons.

1.4 IP ADDRESS

The first column, `src_ip`, indicates the source IP address of the request, that is, the address of the machine (or bot) that initiated the communication with the honeypot server. This information allows tracing the geographical or network origin of the attack. The `src_ip` is thus useful for identifying whether multiple requests come from the same user or the same group of users.

After removing redundant requests originating from the same IP address, we retained only unique IP addresses (13,209 in total). Using the geographical coordinates (longitude and latitude) associated with these addresses, it was possible to represent each attack attempt as a point on a map to visualize the most targeted geographical areas.

However, displaying over 13,000 individual points would have heavily overloaded the map. To address this issue, we grouped geographically close IP addresses using the HDBSCAN clustering algorithm. This method automatically detects dense clusters of points, allowing us to identify packets of nearby IP addresses.

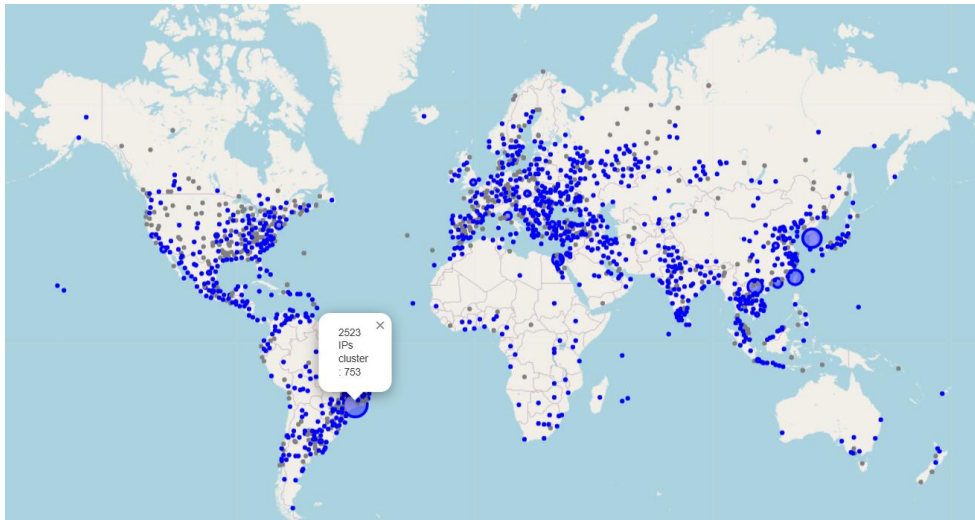


Figure 2: Map of IP Address Locations

It is observed that the IP addresses are distributed worldwide, with a strong concentration in certain regions. This reflects both the density of network infrastructures in these areas and the high Internet connectivity of local populations.

There are also notable clusters, such as in Brazil, where 2,523 IP addresses are grouped in the same area. These concentrations can be explained by several factors like the presence of data centers, use of proxies or VPNs sharing common IP ranges.

It is important to emphasize that many malicious users rely on VPNs (Virtual Private Networks) or proxy servers to hide their true IP address. These tools route their traffic through intermediate servers, making the real geolocation of the attacker more difficult or even impossible to know in some cases.

To determine whether an IP address is associated with a VPN, a proxy, or simply to obtain detailed information about it, several methods and tools can be used, such as online services like ipinfo or specialized databases.

1.5 NETWORK PORT

The analysis of targeted ports is essential because it allows identification of the services aimed at by an attacker. Indeed, attackers generally test numerous ports hoping to find misconfigured services, exposed vulnerabilities, or forgotten web applications. The study of destination ports provides valuable clues about the attackers intentions and the vulnerabilities potentially exploited.

TCP	127.0.0.1:62522	0.0.0.0:0	LISTENING
TCP	127.0.0.1:62522	127.0.0.1:49782	ESTABLISHED
TCP	10.0.0.1:139	0.0.0.0:0	LISTENING
TCP	10.0.0.1:6296	100.100.100.100:80	ESTABLISHED

Figure 3: Output of the netstat Command

We observe four active connections on different ports: two are in a listening state (LISTENING) and two are established. The first two use the IP address 127.0.0.1, which means they listen only locally, on the machine itself. The other two involve the IP address of my computer (here 10.0.0.1): one listens on port 139, used for file sharing, and the other is an established connection on port 80, which likely corresponds to communication with a website who had the address ip 100.100.100.100.

```

Occurrence of network port in the data:
dst_port
80      52791
8000    5736
44123   3151
81      3037
6666    2607
...
54370    1
53222    1
32322    1
18291    1
47222    1
Name: count, Length: 16669, dtype: int64

```

Figure 4: Occurrence of network port

It is clear that the majority of HTTP requests received by the honeypot target the standard HTTP port 80 (52,791 occurrences), although many alternative ports such as 8000 and 81 are also targeted.

1.6 URL

The third column, URL, indicates the full address targeted by the attacker. This is one of the most important components, as it helps understand what the user is trying to access or execute through their request. The URL consists of several parts, each providing valuable information about the intent of the request.

For example, in the following URL we can break it down as follows:

http://000.000.000:5984/tmpfs/auto.jpg?usr=admin114&pwd=666666

- http: the protocol used for communication, here HTTP.
- 000.000.000: this is the target IP address, here assumed to be that of our honeypot server. It can be used instead of a domain name (such as www.amazon.com), since a domain name is simply an alias pointing to an address.
- :5984: this is the targeted port.
- /tmpfs/auto.jpg: the path specifying the resource targeted on the server.
- ?usr=admin114&pwd=666666: parameters containing key=value pairs sent to the server.

Based on the path and transmitted parameters, it is often possible to identify the nature of the request, benign or potentially malicious. For example, an attacker might target sensitive files or directories such as /admin, /login, /config.php, or try to access common elements like favicon.ico (the icon displayed in the browser tab).

We attempted to classify requests based on their URLs. Several approaches were tested, though not fully developed since it was not the main focus of the internship. Like searching for keywords or key expressions using regular expression patterns.

```
# List of sensitive keywords to search for in URLs
sensitive_keywords = ['php', 'env', 'config', 'phpmyadmin', 'git', 'admin', 'exec', 'sql', 'bin', 'tmp']

# List of suspicious file extensions to search for in URLs
sensitive_extensions = ['.git', '.php', '.vscode', '.cgi', 'phpMyAdmin', '.docker', '.aws', 'favicon.ico']
```

Figure 5: List of sensitive keywords and file extensions

	url	hostname	path	parameters	has_sensitive_keywords	has_sensitive_extensions
3140	http://tmpfs/auto.jpg?usr=admin114&pwd=daniel	tmpfs	/auto.jpg	usr=admin114&pwd=daniel	1	0
3141	http://tmpfs/auto.jpg?usr=admin114&pwd=klaster	tmpfs	/auto.jpg	usr=admin114&pwd=klaster	1	0

Figure 6 : Dataset with searching keywords and file extensions

Or clustering using Levenshtein distance to group similar URLs by calculating the minimal number of modifications (insertions, deletions, or substitutions of characters) needed to transform one URL into another.

```

Similar URL Groups :
Groupe 1: ['http://favicon.ico', 'http://zabbix/favicon.ico', 'http://favicon.ico/']
Groupe 2: ['http://sitemap.xml', 'http://setup.xml']
Groupe 3: ['http://robots.txt', 'http://robots.txt/', 'http://sd/robots.txt']
Groupe 4: ['http://?format=json']
Groupe 5: ['http://wsman', 'http://a', 'http://c2', 'http://i', 'http://i.dat', 'http://i.dat/']
Groupe 6: ['http://odinhttpcall1740404188', 'http://odinhttpcall1740396027', 'http://']

```

Figure 7: Clustering usins Levenshtein distance

We can observe groups containing similar URLs, which allows us to cluster or merge them before applying other clustering methods, such as K-means. However, a problem arises in Group 5, where the URLs appear quite different semantically but are still grouped together. This is because only a few modifications are needed to transform one into the other.

Understanding how URLs work is fundamental, especially from the perspective of building an HTTP server. Indeed, when receiving a request, the server analyzes the path to determine which resource or processing should be executed.

1.7 METHOD

The method column indicates the HTTP method used in each request received by the honeypot. HTTP methods, such as GET, POST, PUT, DELETE, etc., define the type of action the client wants to perform on the server. Analyzing these methods is particularly useful for detecting cyberattacks because some methods are more often used for malicious purposes.

```

Occurrence of HTTP methods in the data:
method
GET          240471
OPTIONS      10977
POST         2985
HEAD         170
PUT          55
Name: count, dtype: int64

```

Figure 8: Ocurrance of HTTP methods

The GET method, which is predominantly used in our dataset, is generally used to retrieve resources, while the POST method, here less frequently, is used to send data to the server, which can be exploited to inject malicious code. The OPTIONS method is used to ask the server which

HTTP methods are supported or allowed, which can help attackers. The PUT and DELETE methods are rarely used by legitimate users on public websites, as they allow modifying or deleting resources. Therefore, it would be useful to inspect the associated requests.

Knowing the types of HTTP requests is important when creating a server because they indicate what the user wants to do: retrieve information, send data, modify or delete resources. Each type of request must be handled differently.

1.8 TIMESTAMP

The *timestamp* field indicates the exact date and time when each request was received by the server. Analyzing these timestamps helps detect certain types of attacks, such as brute force attacks, where an attacker often sends hundreds or even thousands of requests in a short period, trying different passwords or URL parameters to access sensitive files or resources.

Using the *timestamp*, we can identify these abnormal activity spikes. To do so, we remove all requests with identical URLs to eliminate bots that attempt the same request thousands of times in a row, which would distort the results. Then we visualize the highest peaks and analyze the URLs of these requests.

```
=== Pic #4 à 2025-02-26 18:25:00+00:00 (144 requêtes) ===
src_ip      url
218064 151.80.58.67 http://auto.jpg?usr=user112&pwd=hockey 2025-02-26 18:25:04.805496+00:00
218065 151.80.58.67 http://auto.jpg?usr=usser&pwd=000000 2025-02-26 18:25:45.564091+00:00
218066 151.80.58.67 http://auto.jpg?usr=usser&pwd=harley 2025-02-26 18:25:49.803374+00:00
218067 151.80.58.67 http://auto.jpg?usr=usser&pwd=andrew 2025-02-26 18:25:49.983736+00:00
218068 151.80.58.67 http://auto.jpg?usr=usser&pwd=sunshine 2025-02-26 18:25:50.169814+00:00
```

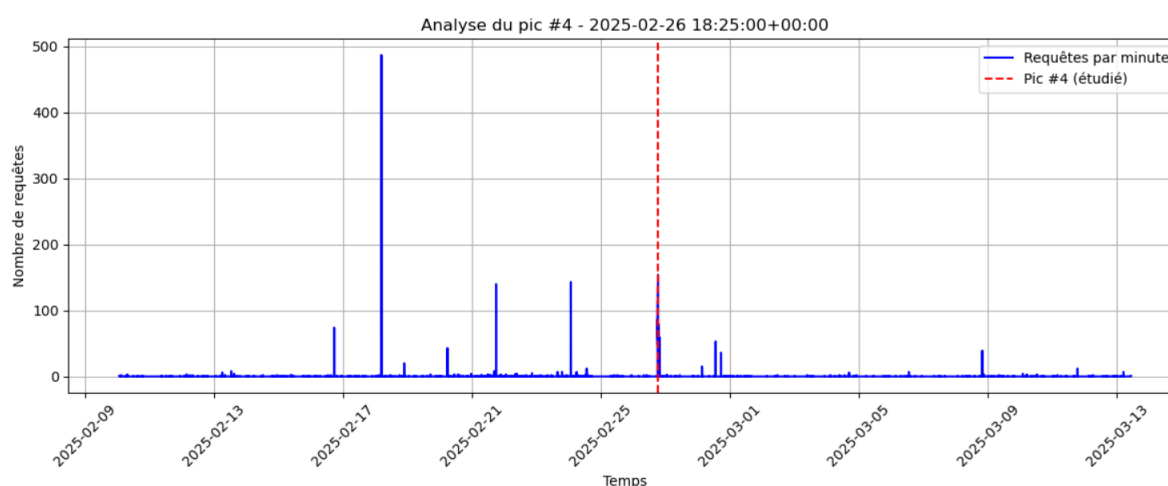


Figure 9: Timestamp pic

It can be observed that, for peak number 4, this is likely a brute force attack attempt, with URLs of the type ?usr=...&pwd=..., where the attacker tries different combinations of usernames and passwords.

1.9 HEADERS

Headers contain additional information sent by the attacker, such as the User-Agent, which can mimic a common browser, or conversely indicate an automated script. This information is valuable for identifying the tools used by attackers.

2 CREATION OF A FIRST SERVER AND GENERATION OF AN HTML PAGE USING LLM

To start the project and get familiar with how a server works, we first developed a simple HTTP server. This initial step helped us understand the basic mechanisms before implementing the core of the project: the integration of a Large Language Model (LLM) capable of dynamically generating the server content.

Today, several frameworks make it easier to create web servers in Python. Among them, FastAPI is one of the most popular due to its simplicity, speed, and modern features. It allows us to quickly set up an HTTP server that can receive and store requests. In addition, it provides great flexibility to extend its functionalities.

2.1 FILE STRUCTURE

As the project becomes more complex and the number of routes, features, and HTML pages increases, it becomes essential to organize the code in a clear and modular way. The final server will include more than twenty files to ensure its proper functioning. A well-structured file organization improves the readability, maintainability, and scalability of the project.

```
my_fastapi_app/
├── app/
│   ├── main.py           # Entry point of the application (FastAPI instance creation)
│   ├── config.py         # Application configuration (env, ports, etc.)
│   ├── routes/           # Folder for route files
│   │   ├── __init__.py
│   │   └── example.py    # Example route
│   ├── middleware/       # Custom middleware (logging, CORS, etc.)
│   │   └── __init__.py
│   └── templates/        # Jinja2 HTML files
│       └── index.html
├── static/               # Static content: CSS, JS, images...
│   ├── css/
│   └── js/
```

Figure 10: File structure

2.1.1 Middleware and Launching the FASTAPI Server

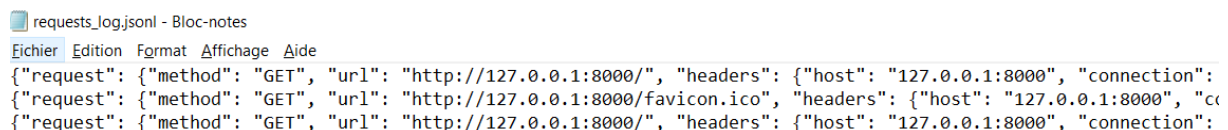
Before developing the server's features, we set up the `main.py` and `config.py` files. The `main.py` file contains the creation of the FastAPI instance, while `config.py` holds the configuration of global variables, such as the Jinja2Templates instance, the list of valid file extensions used later, and authentication keys here used to connect to the LLM.

To start the server, we simply run the following command in the terminal:

```
uvicorn app.main:serv --reload --port 8000
```

This command starts the FastAPI server defined in the `serv` variable from the `app.main` module, enables automatic reloading (useful for development), and listens on port 8000. By default, the server listens only on the local IP address (127.0.0.1), meaning it is accessible only from the local machine.

We also create a `middleware/` folder to store a middleware that intercepts all requests sent to the server. This middleware will log each request in a JSONL file (one JSON object per line). This allows us to keep a record (IP address, timestamp, URL, etc.) of the requests received by the honeypot for further analysis, as explained in the first part.



```
requests_log.jsonl - Bloc-notes
Fichier Edition Format Affichage Aide
{"request": {"method": "GET", "url": "http://127.0.0.1:8000/", "headers": {"host": "127.0.0.1:8000", "connection":
{"request": {"method": "GET", "url": "http://127.0.0.1:8000/favicon.ico", "headers": {"host": "127.0.0.1:8000", "c
{"request": {"method": "GET", "url": "http://127.0.0.1:8000/", "headers": {"host": "127.0.0.1:8000", "connection":
```

Figure 11: Requests_log.jsonl

2.1.2 Routes

The `routes/` folder contains all the routes that our server can handle. Each Python file inside it corresponds to a coherent group of related routes. We started by creating basic routes commonly found in a web server, such as the root “/” and “/login” to handle login logic.

Requests sent to “/” and “/login” are intercepted and processed in the `home.py` and `login.py` files, respectively. These files include the necessary logic to process the request (credential verification, rendering HTML pages, redirection...) and then return an appropriate response to the client.

```
@router.get("/", response_class=FileResponse)
async def read_root():
    filepath = os.path.join("static", "home.html")
    return FileResponse(filepath, media_type="text/html")
```

Figure 12: Fonction for the route «/»

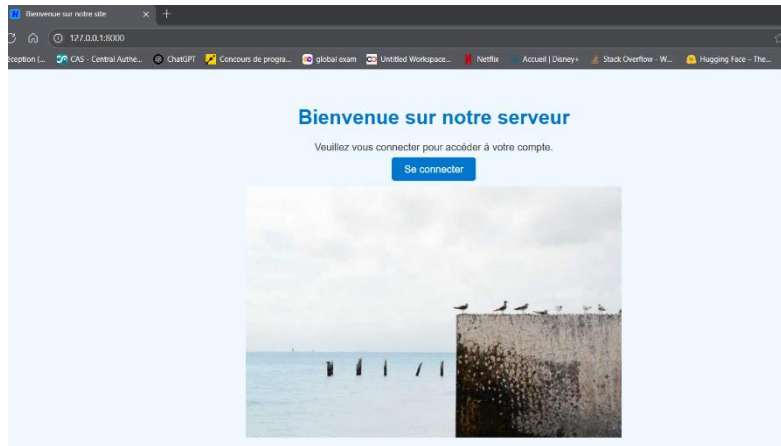


Figure 12: "/" route of the first server

We will mainly use three types of responses in this project:

1. **Sending raw files** (such as .txt or .js), which allows us to directly serve files with the correct type.
2. **Dynamic rendering of HTML pages** using templates, to generate enriched interfaces based on server-side data.
3. **Redirecting users** to another route. For example, the "Log In" button on the "/" page redirects to the "/login" page.

FastAPI also provides a built-in error handling system. The one we're particularly interested in is the 404 error, triggered when a user accesses a non-existent route. For this first server, we simply returned a basic 404 error page.

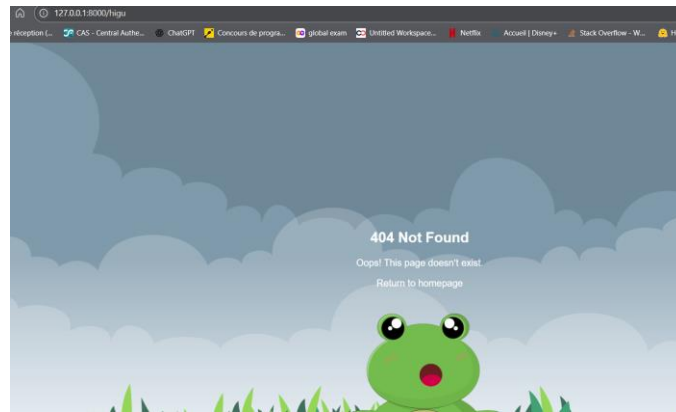


Figure 13: 404 error page

This will be especially important later, as we will use a LLM (Large Language Model) to dynamically generate responses. The requested routes will not exist ahead of time, so all the logic will need to be handled in the 404 error handler.

As previously mentioned, there are several HTTP methods used to send requests to a server. In this project, we will mainly handle the GET and POST methods, which are used to retrieve and send information, respectively.

For our /login route, the GET method will return the HTML page containing the login form to the user. The POST method will be used to process the data entered by the user (username and password) when submitting the form.

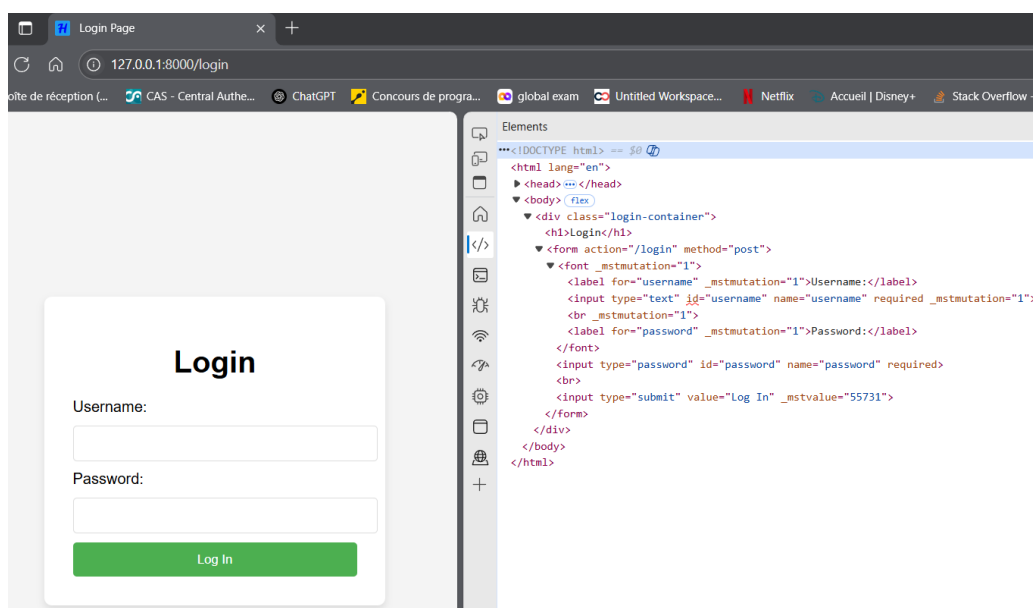


Figure 14: "/login" route and inspection

On the left side, we can see the content of the page at the /login address. On the right side, we observe the HTML code of this page. We notice that the form correctly sends a POST request (`<form action="/login" method="post">`) to the /login endpoint with the fields username and password. This request will then be held by the function `@router.post("/login", response_class=HTMLResponse)` in the login.py file.

We noticed in the request_log.jsonl file that a request to /favicon.ico appears every time a web page is visited on the server. This request is made to retrieve the icon displayed in the browser tab. To make our server more realistic and avoid unnecessary 404 errors, we created a /favicon.ico route that returns a .ico file generated using an online tool.

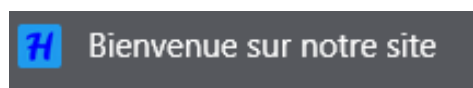


Figure 15: Favicon.ico

2.1.3 Static Files and Templates Structure

The static/ folder contains static files directly accessible by the client through the URL /static/filename, such as CSS stylesheets, images, JavaScript scripts, or the favicon. These files are served as-is, without any server-side processing.

Meanwhile, the templates/ folder holds the HTML files. These templates contain HTML code with dynamic variables that will be replaced at runtime with data provided by the server. In our case, for the login handler, this allows displaying information about the user who logged in.



Figure 16: Users route

2.2 AUTHENTICATION AND PASSWORD PROTECTION

In the URL shown in the previous image, we can see a token (a unique and random sequence of characters) passed as a parameter (Mathilde?token=eyJ...), which is one of the common

methods to secure access during a form-based login. Upon her first login with her password, a unique token is assigned to Mathilde and stored both in her cookies and on the server side. If she tries to access her account again, the token stored in her cookies is compared to the one stored on the server to validate her session.

To protect user passwords and manage login logic, we created a SQLite database where passwords are not stored in plain text but hashed using a cryptographic algorithm that irreversibly transforms the password into a string of characters. This way, if an attacker manages to access the database, they cannot directly retrieve the passwords. When Mathilde logs in, her entered password is hashed and compared to the hashed password stored in the database.

We also created an `/admin` route accessible via a token (in our example, `abc123`). This admin page allows viewing the list of users, their usernames, and their hashed passwords.

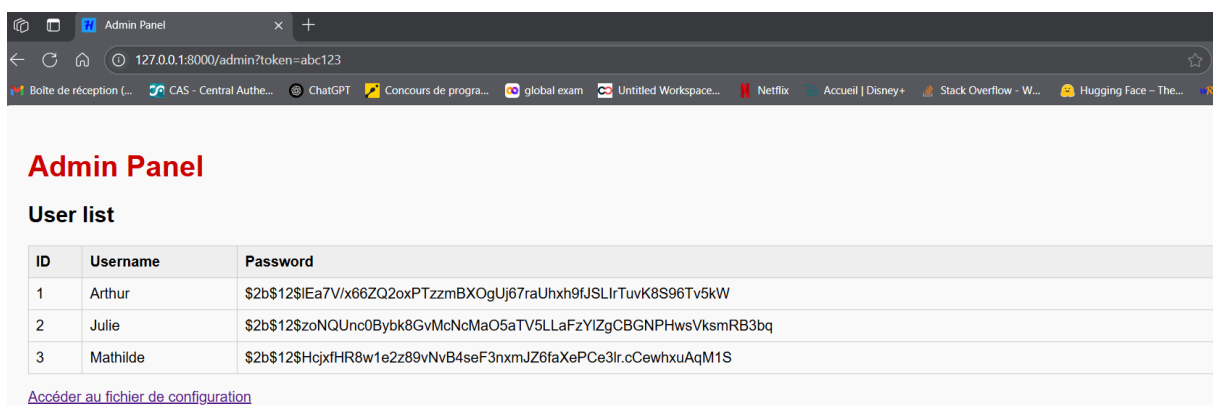


Figure 17: Admin route

To enhance security, we could have implemented more robust measures, such as replacing the token system with more secure cookie settings (like `HttpOnly` and `Secure` flags) or incorporating a salt when hashing passwords. Salting ensures that even if two users have the same password, their stored hashes will be different. Since these aspects were not central to our project, we did not explore them further.

2.3 DYNAMIC RESPONSE GENERATION USING A LLM

After setting up a basic functional server capable of logging incoming HTTP requests to a file, the next step was to integrate a Large Language Model (LLM) to generate the server's content dynamically. The goal was to simulate a simple fake web service, such as a surveillance camera

interface or a directory listing, intended to trap attackers. The LLM enables automatic generation of the server content, whether HTML code, JavaScript, or other elements necessary for the fake service to function, without requiring manual coding, which is not the main focus of this project.

To achieve this, we used models provided by our laboratory. Using a VPN and an authentication token, we implemented a function to send prompts to one of the available models and receive responses.

The final objective was for the LLM to automatically generate certain responses to HTTP requests. We therefore needed to select a model powerful enough to produce realistic answers, while remaining fast enough to ensure an acceptable response time. We chose the *LLaMA 3.1 8B* model, which has around 8 billion parameters. For comparison, the most powerful *LLaMA 3.1 400B* model has over 400 billion parameters.

To ensure coherent and reproducible responses, we set the temperature parameter to 0, meaning the model always chooses the most likely word. This way, the same request will consistently yield the same response.

2.3.1 Choice of the web service

The first web service we chose to simulate was a surveillance camera interface, a relatively simple interface and a common target for attackers. After prompting our LLM to generate a web page mimicking a surveillance camera feed, we quickly realized that detailed and specific instructions were required in the prompt to obtain a realistic and coherent result.

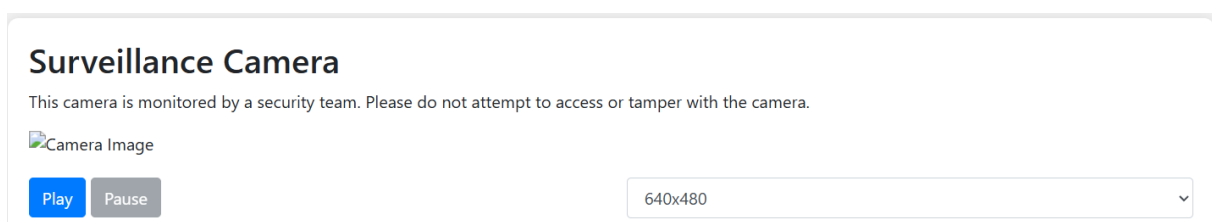


Figure 18: Surveillance camera page with a simple prompt

To address this issue, we used Shodan, a specialized search engine that indexes internet-connected devices worldwide, including surveillance camera interfaces. Our goal was to find a surveillance camera in order to create a realistic replica of it. To avoid overloading or compromising a remote server, which could belong to a third party, we downloaded the entire

HTML page locally for analysis. We then designed a prompt tailored for the LLM to reproduce a similar interface automatically. This process was carried out strictly for private and experimental purposes. If the server were ever to be made public, it would be necessary to obtain the appropriate rights to use that HTML content.

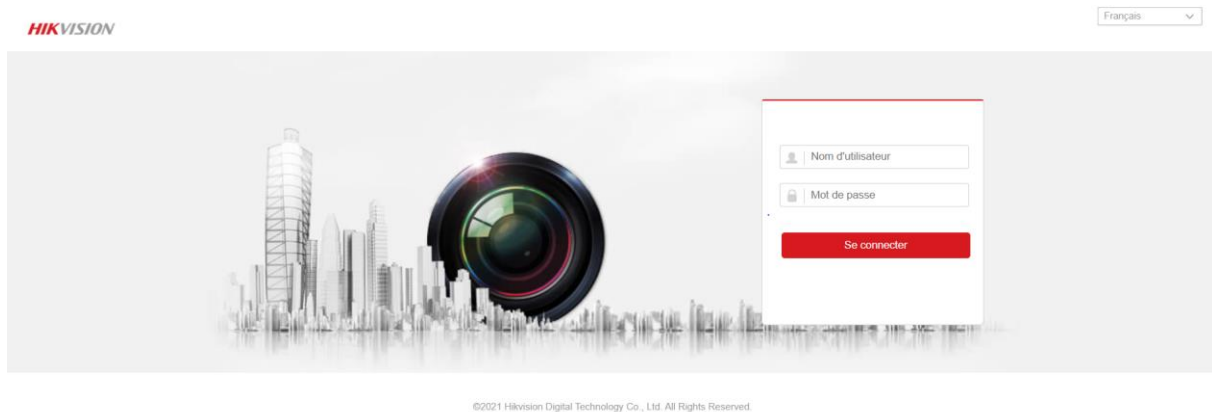


Figure 19: Shodan camera

We then analyzed the page to extract its visual and structural components, such as HTML tables, images, fonts, CSS styles, and layout elements. The goal was to craft a clear and well-structured prompt that would enable the LLM to generate a realistic and faithful reproduction of the interface.

2.3.2 Web Page Analysis Using Selenium

Using Selenium, a Python library commonly used for web scraping, we developed a function to iterate through all elements on a given web page, in this case, our `camera.html` file. This function extracts, for each element, its position, size, text content, and the URL of images included via CSS.

This approach allowed us to capture both the images and the visible components of the page. However, since the script collects all HTML elements, it often includes duplicates (due to nested structures) as well as irrelevant elements. To clean up the results, we applied a first filter to exclude elements with a width, height, or position equal to zero. Then, a second filter removed nested elements with identical text, keeping only one visible instance per content block.

The goal was to later apply these same techniques to more complex pages, in order to automate and simplify the replication process with the help of a language model.


```

Number of elements before filtering: 221
Number of elements after first filtering: 34
Number of elements after second filtering:: 6
div - 'English
  Login
  ©2021 Hikvision Digital Technology Co., Ltd. All Rights Reserved.' at (0, 0) [1028x574]
span - 'English' at (868, 65) [98x23]
input - 'User Name' at (721, 237) [236x30]
input - 'Password' at (721, 284) [236x30]
label - 'Login' at (824, 354) [33x15]
div - '©2021 Hikvision Digital Technology Co., Ltd. All Rights Reserved.' at (0, 520) [1028x54]

```

Figure 20: Extract of principal elements of the HTML page

2.4 FINAL PROMPTS

Once we had extracted all the necessary elements, we were able to start crafting prompts aimed at reconstructing a functional version of the surveillance camera web page. After numerous attempts, we quickly realized that generating the entire page with a single prompt was difficult, if not impossible, for two main reasons.

To be convincing, the page doesn't just need to look realistic, it must also be functional. This means that all interactive elements, such as the language selector or login form, must be properly implemented. These typically require JavaScript, which adds another layer of complexity.

Even small changes in the prompt can result in drastically different HTML output, making it difficult to control the page's structure, appearance, and behavior consistently.

As a result, we adopted a different strategy: Instead of relying on a single prompt, we started with one prompt to generate the basic visual layout of the page, and then submitted the resulting HTML to additional follow-up prompts designed to fix or complete any missing elements. After many iterations, we finalized a pipeline based on three distinct prompts.

The first prompt generate a basic visual version of the page, relying on the information obtained through web scraping. The second prompt was written to make the language selector functional. Finally, a third prompt was used to create a realistic login manager. Prompts and output are in the annexes.

However, starting with the third prompt, we noticed that the LLM began modifying unrelated parts of the HTML page, for example, changing the color of the buttons. Even the slightest

change in the prompt would result in a completely different output, making it nearly impossible to iteratively refine the page by stacking prompts to improve specific components.

2.5 CHALLENGES ENCOUNTERED

This initial approach, replicating an existing web service, presented several challenges. First, crafting a prompt capable of producing a realistic and convincing result proved to be difficult. There were also potential copyright concerns related to duplicating a real-world service. But most importantly, we faced the inherent complexity of recreating an online service, even a relatively simple one.

For example, the surveillance camera interface we attempted to reproduce using the LLM turned out to be far more complex than it appeared. Upon inspection, we discovered that the original web page triggered dozens of requests to load JavaScript scripts and other external resources. By checking the Network tab in the browser's developer tools, we observed over 70 separate HTTP requests initiated as soon as the page loaded

Name	Status	Type
login.asp?_1752483317563	304	docu...
sea.min.js	200	script
ui.css?version=17524867...	200	styles...
sea-config.js?version=175...	200	script
nocache.js	200	script
jquery.min.js	200	script
polyfill2.js	200	script
common.js?version=V4.0....	200	script
jquery.layout.js?version=...	200	script
jquery.cookie.js?version=...	200	script
json2.js?version=V4.0.1bu...	200	script
angular.min.js?version=V...	200	script
websdk.js?version=V4.0.1...	200	script
cryptico.min.js?version=V...	200	script
jsVideoPlugin-1.0.0.min.js...	200	script
base64.js?version=V4.0.1...	200	script

All these elements were essential to the proper functioning of the camera interface, and reproducing nearly all of them would have been both unrealistic and outside the goal of the project.

72 requests 186 kB transferred 2.1 MB resources

Figure 21: Requests sent by the page

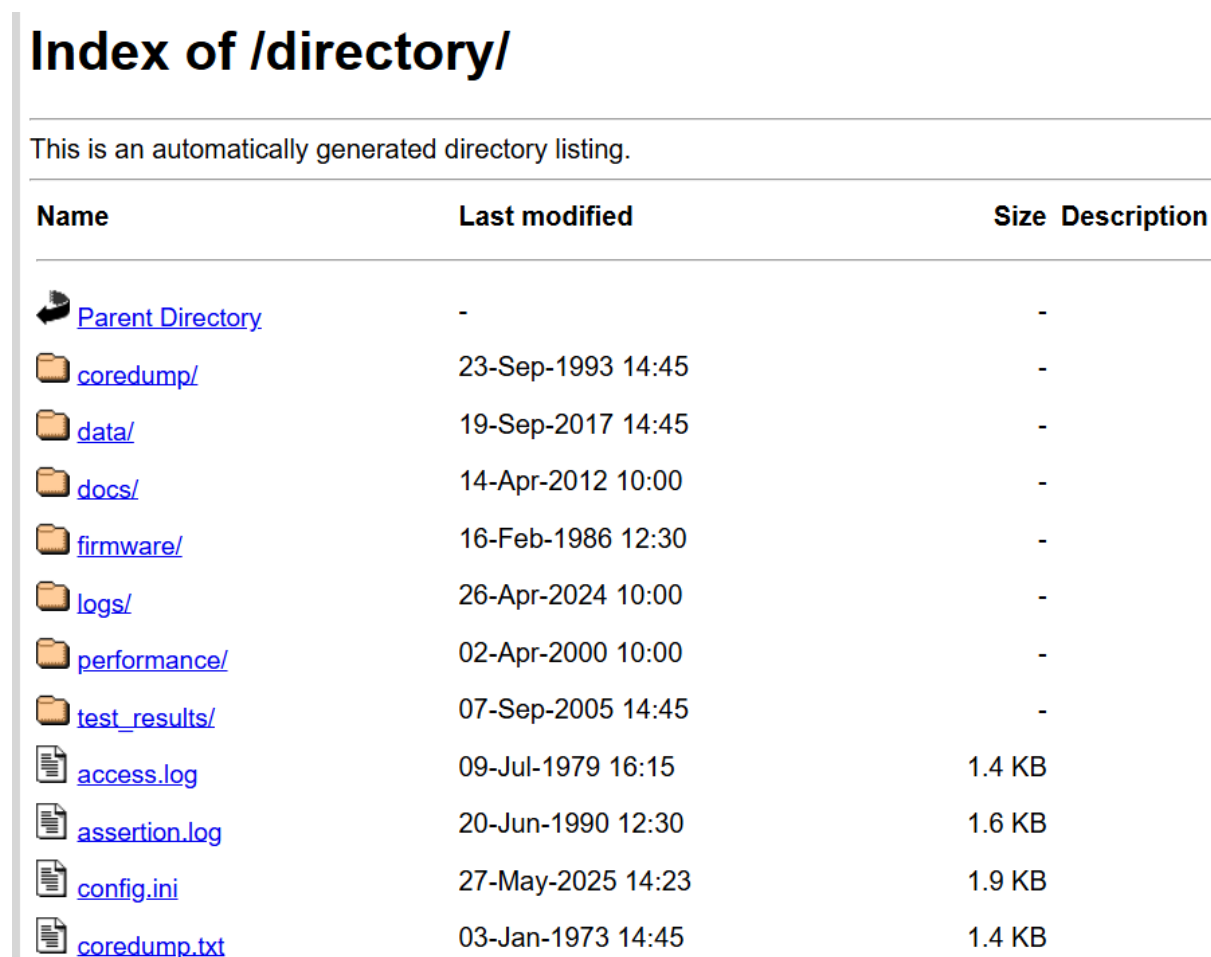
We therefore decided to switch to simulating a directory listing service, which is much simpler, lightweight, easy to implement, and a common target for attackers. Additionally, the LLM

could dynamically generate fake files and folders necessary for the directory listing, making it a more practical and adaptable option.

3 IMPLEMENTING A FAKE DIRECTORY LISTING SERVICE

3.1 VISUAL INTERFACE

The first step in creating this new server was to design the visual interface of the directory listing. We chose to replicate the classic appearance of Apache directory listings to ensure a familiar and credible rendering for attackers or bots. To do this, we retrieved the HTML code of an authentic Apache directory listing and adapted it to our needs in the form of an HTML template. We then replaced the listed elements, folders and files, with those generated dynamically by our LLM, as explained in the previous section.












This is an automatically generated directory listing.		
Name	Last modified	Size Description
 Parent Directory	-	-
 coredump/	23-Sep-1993 14:45	-
 data/	19-Sep-2017 14:45	-
 docs/	14-Apr-2012 10:00	-
 firmware/	16-Feb-1986 12:30	-
 logs/	26-Apr-2024 10:00	-
 performance/	02-Apr-2000 10:00	-
 test_results/	07-Sep-2005 14:45	-
 access.log	09-Jul-1979 16:15	1.4 KB
 assertion.log	20-Jun-1990 12:30	1.6 KB
 config.ini	27-May-2025 14:23	1.9 KB
 coredump.txt	03-Jan-1973 14:45	1.4 KB

Figure 22: "/directory/" route

The entry point to our directory listing is located at the `/directory/` root path. The initial set of files and folders displayed at this location was designed to attract the attention of bots or

attackers. These are stored on our server as a Python list. We selected names that are likely to appear sensitive or valuable, such as `coredump/`, `logs/`, `firmware/`, and for files, `config.ini`, `access.log`.

Each file or folder appears as a clickable link. For instance, clicking on `coredump.txt` redirects the user to `http://127.0.0.1:8000/directory/coredump.txt`, while clicking on the `firmware/` folder leads to `http://127.0.0.1:8000/directory/firmware/`.

3.2 GENERATING CONTENT USING THE 404 ERROR

Once the directory listing interface was working, we needed to make it interactive. The goal was to dynamically generate the content of subfolders and files when a user clicks on them. For example, if the user clicks on a folder like `performance/`, a new page should display a fake list of files and subfolders. On the other hand, if the user clicks on a file, a page with realistic content matching the file type (HTML, JavaScript, Python, etc.) should be shown.

The idea is to create a potentially infinite structure of folders and subfolders to trap automated bots that explore websites.

At this point, only the `/directory/` route is explicitly defined in the server. Any other URL entered by the user returns a 404 error. We decided to take advantage of this 404 error to generate fake content. Instead of displaying a classic error page, like we did with the first server, we customized the 404 response to return a fake file or folder page that matches the requested URL. This is done using handlers, special Python functions triggered when certain events happen, like an error. In our case, the server uses a custom handler that is automatically called by FastAPI when a 404 error is detected.

```

@serv.exception_handler(404)
async def custom_404_handler(request: Request, exc: Exception):
    method = request.method
    path = request.url.path
    requete = (
        f"path = \"{path}\"\\n"
    )
    ext = os.path.splitext(path)[1].lower()
    if method == "GET":
        # 1. In the case of a directory
        if path.endswith('/'):
            return await honeypot_handlers.handle_fake_directory(request, requete)
        # 2. In the case of a file
        elif ext in text_file_extensions:
            if ext == ".html":
                return await honeypot_handlers.handle_html(request, requete)
            elif ext == ".txt":
                return await honeypot_handlers.handle_txt(request, requete)
            elif ext == ".js":
                return await honeypot_handlers.handle_js(request, requete)
            elif ext == ".py":
                return await honeypot_handlers.handle_py(request, requete)
            else:
                return await honeypot_handlers.handle_other_extensions(request, requete)
        else:
            return RedirectResponse(url="/directory/")
    elif method == "POST":
        return await honeypot_handlers.handle_default_404(request)
        #return await honeypot_handlers.handle_post(request, requete)
    # 3. Else
    return await honeypot_handlers.handle_default_404(request)

```

Figure 23: 404 error handler

When a request is received by our custom 404 error handler, we extract the HTTP method, the path of the URL, and the file extension if it exists. Then, depending on the method and the requested URL, we call the appropriate handler function and pass the request to it. This function allows the LLM to generate a realistic response that matches the type of resource being requested.

3.2.1 Directory case

When a user clicks on a folder (when a GET request is sent to a URL ending with /), the `handler_fake_directory` function is called. This function sends a prompt to the LLM, asking it to generate a JSON containing a list of files and subfolders, with their size and last modified date.

```

content = [
    f'Generate a JSON list containing between 25 and 40 fictitious files or directories (no duplicates) to display in an Apache directory listing.\n',
    f'The current path is: \"{current_path}\".\n',
    'Each list item is an object with the following keys:\n',
    '- "name": a realistic file name (e.g. "config.ini") or folder name (e.g. "logs/"), without digits and invented realistically.\n',
    '- Directory names must be a single name followed immediately by a slash ("/"), with no additional slashes inside the name.\n',
    '- "size": readable size in KB (between "0.7KB" and "1.7KB", "-" if it is a directory)\n',
    '- "modified": date and time of last modification, format "27-May-2025 14:23"\n',
    '- "is_dir": true if it is a directory, false otherwise\n',
    '\n',
    'Do not generate image files (e.g. .png, .jpg, .gif, .bmp) or executable binary files (e.g. .exe, .bin, .dll, .so).\n',
    '\n',
    'Respond only with this complete JSON, starting with "[" and ending with "]"'.
    'Do not cut your response and do not add any other text, explanation, or code block tags.'
]

```

Figure 24: Prompt for generate json list of files and folders

To avoid displaying unusable or hard-to-generate files (such as binary, executable, or video files), we defined a list of allowed extensions. Any file name generated by the LLM that doesn't match one of these extensions is discarded.

```
text_file_extensions = [
    ".txt", ".md", ".rtf", ".html", ".css", ".js", ".py", ".php", ".java", ".cpp", ".c",
    ".cs", ".sh", ".bat", ".sql", ".json", ".xml", ".yaml", ".yml", ".ini", ".cfg",
    ".conf", ".log", ".tex", ".csv", ".toml", ".env", ".properties", ".asm", ".go",
    ".rb", ".pl", ".vb", ".r", ".swift", ".kt", ".scss", ".less", ".vue", ".ts", ".tsx",
    ".jsx", ".gradle", ".gradle.kts", "Dockerfile", "Makefile", ".adoc", ".nfo", ".cmake",
    ".podspec", ".proto", ".gitignore", ".gitattributes", ".podspec", ".ld", ".d"
]
```

Figure 25: File extensions

The server then returns the `directory.html` template, populated with the elements generated by the LLM.

Index of /directory/docs/tests/

This is an automatically generated directory listing.

Name	Last modified	Size	Description
 Parent Directory	-	-	
 cache/	20-Apr-2024 13:30	-	
 data/	15-Jan-2024 09:30	-	
 dataframes/	15-Apr-2024 13:00	-	
 docs/	27-Feb-2024 09:00	-	
 fixtures/	18-Jan-2024 09:00	-	
 logs/	25-Apr-2024 10:45	-	
 results/	20-Mar-2024 10:00	-	
 schemas/	18-Apr-2024 13:15	-	
 templates/	15-Mar-2024 10:45	-	
 tests/	18-Feb-2024 13:20	-	
 testsuite/	25-Feb-2024 10:00	-	
 cache.json	28-Apr-2025 14:15	0.7KB	
 config.ini	27-May-2025 14:23	0.7KB	

Figure 26: `"/directory/docs/tests/"` route

3.2.2 File Case

When a user clicks on a file (when a GET request is sent to a URL ending with a file extension), we extract the extension from the URL. Based on this extension, we call the appropriate handler. In our implementation, we have defined specific handlers for the .txt, .html, .js, and .py extensions.

For other extensions:

- If the extension is included in our allowed list, the response is handled by a generic function using a tailored prompt.
- If the extension is not allowed, the user is redirected to /directory/.

Using a different prompt depending on the file type allows us to provide more specific instructions. For example, for HTML files, we explicitly tell the LLM in the prompt to use a known CSS framework like Milligram, which is available in the static folder of our server.

```
content = `
f'Generate a JSON list containing between 25 and 40 fictitious files or directories (no duplicates) to display in an Apache directory listing.\n'
f'The current path is: "{current_path}".\n'
'Each list item is an object with the following keys:\n'
'- "name": a realistic file name (e.g. "config.ini") or folder name (e.g. "logs/"), without digits and invented realistically.\n'
'- Directory names must be a single name followed immediately by a slash ("/"), with no additional slashes inside the name.\n'
'- "size": readable size in KB (between "0.7KB" and "1.7KB", "-" if it is a directory)\n'
'- "modified": date and time of last modification, format "27-May-2025 14:23"\n'
'- "is_dir": true if it is a directory, false otherwise\n'
'\n'
'Do not generate image files (e.g. .png, .jpg, .gif, .bmp) or executable binary files (e.g. .exe, .bin, .dll, .so).\n'
'\n'
'Respond only with this complete JSON, starting with "[" and ending with "]"'.
'Do not cut your response and do not add any other text, explanation, or code block tags.'
`
```

Figure 27: Prompt for generate html page

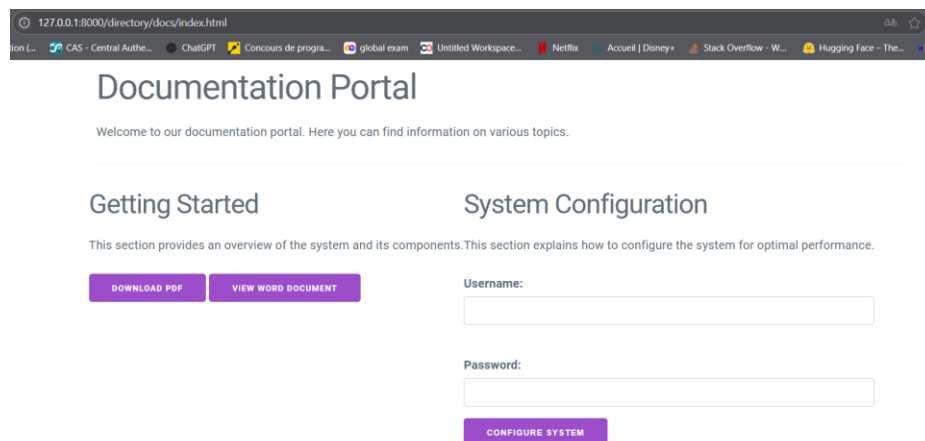


Figure 28: «/directory/docs/index.html» route

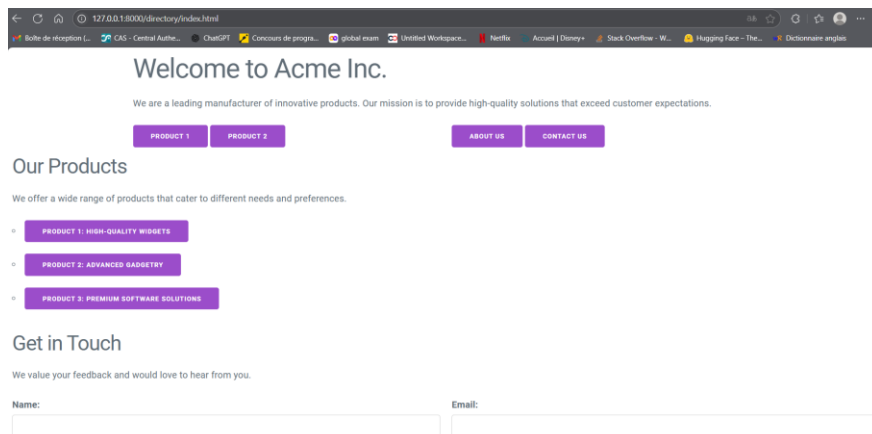


Figure 29: «/directory/index.html» route

We can see that thanks to the CSS file, the generated HTML files have the same layout and look quite realistic. We also notice that even though both files are named `index.html`, they are different because they come from different folders, `/directory/index.html` for the first one, and `/directory/docs/index.html` for the second. Examples of page generate for other extensions in the annexe.

We can even go further and generate prompts not only based on the file extension but also based on the file name, allowing for even more precise and context-aware content generation.

3.2.3 Improvements and Issues

When the request we receive is not using the HTTP GET method, we redirect the user to a real error page, just like we did in the first server. We tried to handle POST requests in particular, because when the LLM generates HTML pages, it often includes forms like the pictures before. These forms usually send POST requests with the user's input. So, we created a dedicated handler for this case, but the results were not satisfying.

One of the main problems is how files and folders are created: everything is generated dynamically. This means that if a user types a random URL ending with a slash `/` or a valid extension, the LLM will still generate content, even for strange URLs like `asabdk.html` or `gag/dje/`. This behavior can feel unrealistic or even suspicious.

To avoid this issue, one solution could be to generate content only for files and folders that have already been created. However, this approach brings two main limitations:

- The response time might get slower as the number of stored files and folders increases.
- A user might first see a real 404 error when typing a URL like `directory/data/data.sql`, but if the LLM later generates a file named `data.sql`, the same URL could return a valid page. This creates inconsistent server behavior.

Finally, after a few directory explorations, we noticed that the generated file names and content start to look less realistic or logical. This weakens the illusion and reduces the overall effectiveness of the honeypot.

Index of /directory/data/temp2/temp/

This is an automatically generated directory listing.

Name	Last modified	Size	Description
 Parent Directory	-	-	
 backup/	20-Apr-2024 13:45	-	
 backup2/	15-Feb-2024 09:05	-	
 cache/	15-Feb-2024 09:05	-	
 cache/	20-Apr-2024 13:45	-	
 cache2/	15-Feb-2024 09:05	-	
 data/	15-Feb-2024 09:05	-	
 logs/	25-Apr-2024 10:11	-	
 logs/2024/	20-Apr-2024 13:45	-	
 temp/	15-Feb-2024 09:05	-	
 temp/	20-Apr-2024 13:45	-	
 temp/	15-Feb-2024 09:05	-	
 temp2/	25-May-2025 14:50	-	
 temp3/	20-Apr-2024 13:45	-	
 config.ini	27-May-2025 14:23	0.7KB	

Figure 30: `"/directory/data/temp2/temp/"` route

We can observe redundant directories like `temp/`, as well as composed paths such as `logs/2024/`, despite having specified in the prompt to avoid duplicates and to generate file names containing only a single slash.

3.3 SAVING THE GENERATED CONTENT

To improve both performance and realism, it was important to save the content generated by the LLM locally. Since the model uses a temperature of 0, its output is deterministic. That means there is no need to call the LLM again for the same file or folder. To do this, we use two dictionaries:

- The first one maps each folder path (e.g. /directory/data/) to the list of files and subfolders it contains.
- The second one maps each file path (e.g. /directory/data/archive.ini) to its generated content.

Before each LLM call, we check if the requested path (file or folder) is already in the dictionaries. If it is, we return the saved content directly, without asking the LLM again. To limit memory usage, we keep only the last 100 files and last 15 folders in memory.

At the same time, we store a persistent trace of everything the LLM generates in two .jsonl files:

- file_contents.jsonl: stores file content,
- directory_history.jsonl: stores folder structures.

Indeed, dictionaries are part of the Python server process. So when the server stops, their data is lost unless it has been written to disk.

3.4 MONITORY SERVER

Once the directory_server was running, the next goal was to visualize attacker behavior and the LLM-generated responses in real time. For that, we had three .jsonl files that are updated continuously:

- request_logs.jsonl: contains all incoming requests to the server,
- directory_history.jsonl: contains folder structure,
- file_contents.jsonl: contains file content.

We built a second server called monitor_server, which displays these elements live on an interactive HTML page. We used a WebSocket connection, a communication protocol that allows real-time data exchange without sending a new HTTP request each time. The WebSocket

reads the .jsonl files line by line and sends new data to an HTML template, ensuring a clear and dynamic visualization.

We could have used other methods to share data between the two servers, like REST API. However, that would mean directly connecting `directory_server` and `monitory_server`, which is a security risk. The main server (`directory_server`) is meant to be attacked. Linking it directly to the monitoring interface could expose the dashboard to potential threats. By fully separating the two servers and using .jsonl files as an intermediary, we maintain strong isolation between the honeypot and the dashboard.

One of the main challenges was detecting new entries in the files. At first, we checked every second if a file had changed, but this method reloaded the whole file every time. This overloaded the HTML page and caused errors. To fix this, we implemented a more efficient approach: we track the last read position in each file and only read the lines added after that point. This way, only the new data is sent, keeping updates smooth and lightweight.

Monitoring Honeypot HTTP

Received HTTP requests

MMethod	URL	Client IP	Status	Processing Time	Detail
GET	http://127.0.0.1:8000/directory/data/temp2/tem	127.0.0.1	307	0.00 ms	Show details
GET	http://127.0.0.1:8000/directory/	127.0.0.1	200	28.90 ms	Show details
GET	http://127.0.0.1:8000/directory/config.ini	127.0.0.1	200	7114.40 ms	Show details
GET	http://127.0.0.1:8000/directory/data/	127.0.0.1	200	5810.49 ms	Show details
GET	http://127.0.0.1:8000/directory/data/temp/	127.0.0.1	200	7785.29 ms	Show details
GET	http://127.0.0.1:8000/directory/data/temp/history.bat	127.0.0.1	200	3479.29 ms	Show details
GET	http://127.0.0.1:8000/directory/data/temp/report.xml	127.0.0.1	200	2303.61 ms	Show details
GET	http://127.0.0.1:8000/directory/data/	127.0.0.1	200	3.07 ms	Show details
GET	http://127.0.0.1:8000/directory/data/results/	127.0.0.1	200	5970.98 ms	Show details
GET	http://127.0.0.1:8000/directory/data/	127.0.0.1	200	0.00 ms	Show details
GET	http://127.0.0.1:8000/directory/	127.0.0.1	200	4.66 ms	Show details
GET	http://127.0.0.1:8000/directory/config.ini	127.0.0.1	200	0.00 ms	Show details

Previous page Page 1 / 1 Next page

Figure 31: Monitory server, received HTTP requests

We can see the processing times on the right. The higher times correspond to calls to the LLM, while the times of a few milliseconds correspond to resources already present on the server, such as the directory folder or files previously generated by the LLM, like `/directory/config.ini` which was created earlier. The *Show Details* button displays the full JSON file with all the request details (headers, IP, query parameters, etc.).

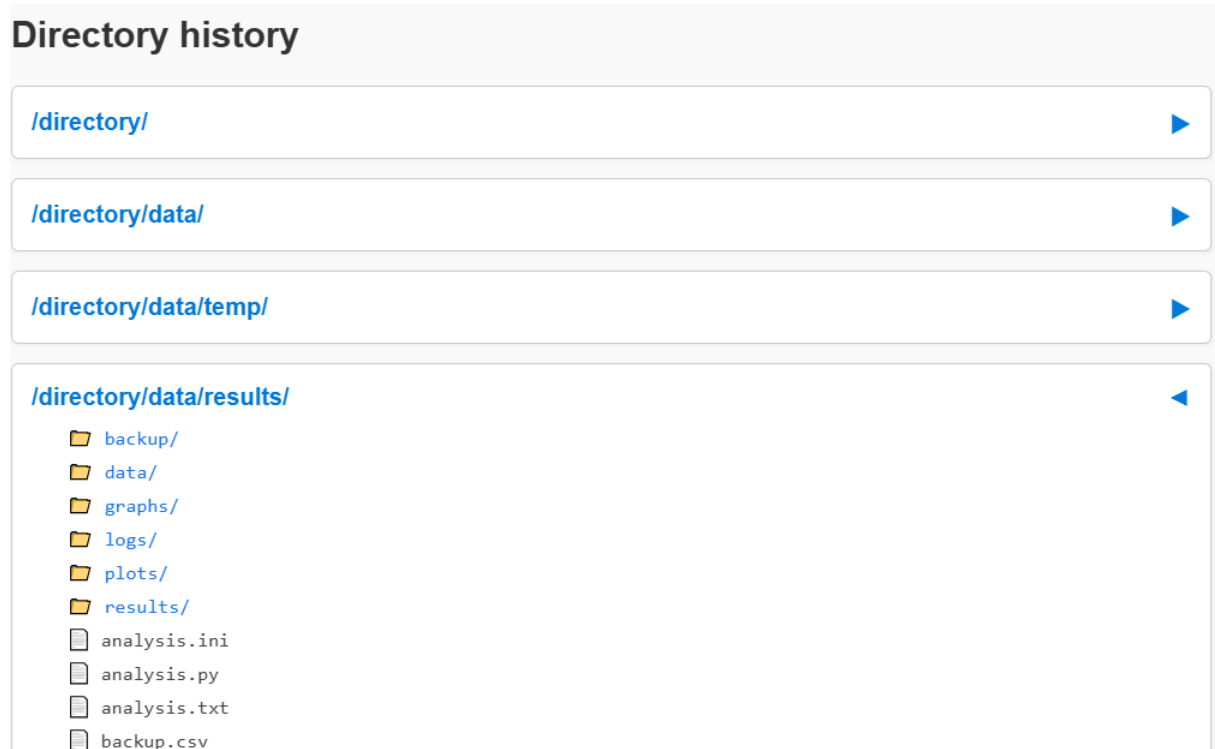


Figure 32: Monitory server, directory history



Figure 33: Monitory server, files history

CONCLUSION

During these five months of internship, I had the opportunity to work, with the support of my supervisor, on the creation of an HTTP server powered by a large language model (LLM), with the goal of building a honeypot. This project, which is highly relevant in today's cybersecurity landscape, allowed me to learn important concepts and terminology in the field, while improving my technical skills in Python, especially using the FastAPI framework.

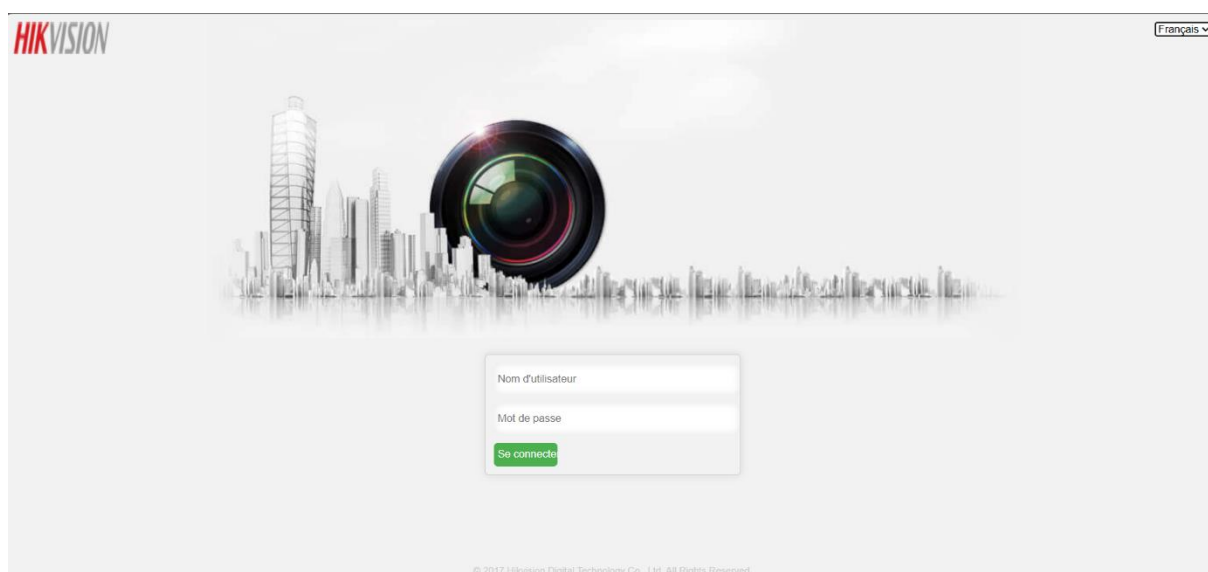
Although the server is functional, it still has some issues and areas for improvement. Unfortunately, I didn't have enough time or expertise to fully address them. The project still has great potential. The testing phase was not conducted, as my main task was to build the server itself, and I wasn't able to make the honeypot publicly available or expose it to real attackers, mainly for security reasons.

This project gave me a clear introduction to the world of cybersecurity and its many possibilities in fighting against increasingly advanced attacks. This internship also gave me the chance to discover Slovenia, a beautiful country, as well as neighboring countries that I had the chance to visit.

ANNEXES

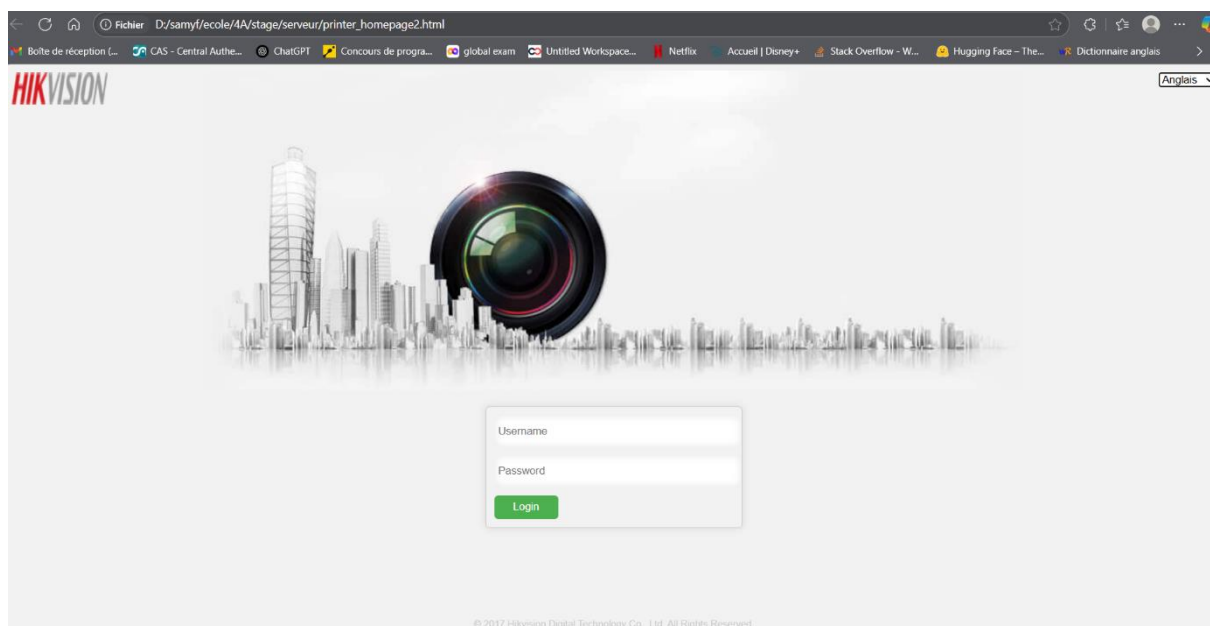
Annexe 1 : First prompt and output HTML for recreating the Shodan camera

```
content1 = (  
  "Act as if you are the login homepage of a surveillance camera.\n"  
  "Generate only the full HTML code of this page, without any explanation or comment.\n"  
  "The HTML code must include all necessary CSS and JavaScript directly embedded in the file (no external files).\n"  
  "The page must contain:\n"  
  "- A logo displayed at the top left of the page.\n"  
  "  Image: http://127.0.0.1:8000/static/logo.png, size 120x40.\n"  
  "- A background image (or banner) displayed below the logo, horizontally centered.\n"  
  "  Image: http://127.0.0.1:8000/static/login.jpg, fixed size 1021x400.\n"  
  "- A login form centered horizontally on the page, positioned just below the main image.\n"  
  "  It must include:\n"  
  "    - A text field with placeholder 'Username', size 236x30\n"  
  "    - A password field with placeholder 'Password', size 236x30\n"  
  "    - A 'Login' button below the password field, with an appropriate width (e.g., 80px)\n"  
  "- A dropdown menu to select the language, aligned at the top right of the page.\n"  
  "  The default language is 'French'.\n"  
  "- A copyright notice centered at the very bottom of the page, in small gray text:\n"  
  "  '©2017 Hikvision Digital Technology Co., Ltd. All Rights Reserved.'\n"  
)
```



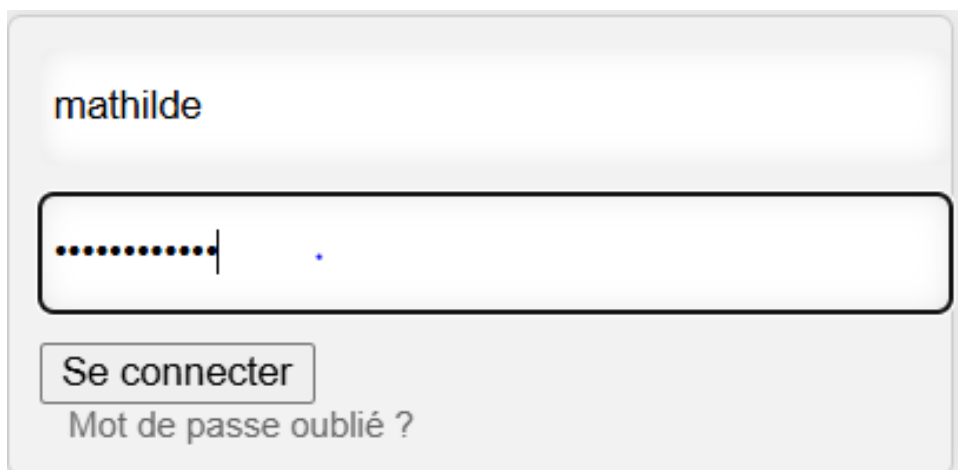
Annexe 2 : Second prompt and output HTML for recreating the Shodan camera

```
"model": model,
"messages": [
  {
    "role": "user",
    "content": content1
  },
  {
    "role": "assistant",
    "content": html1_code
  },
  {
    "role": "user",
    "content": (
      "Improve the language selector by adding JavaScript so that when the user selects a language, "
      "all the text on the page dynamically changes (placeholders, buttons). "
      "Include the JavaScript directly within the HTML code."
    )
  }
],
```



Annexe 3 : Third prompt and output HTML for recreating the Shodan camera

```
"messages": [
  {
    "role": "user",
    "content": content1
  },
  {
    "role": "assistant",
    "content": html2_code
  },
  {
    "role": "user",
    "content": (
      "Generate only the JavaScript code that modifies the behavior of an existing login form with the class 'form-container'.\n"
      "This script must:\n"
      "- Intercept the click on the 'Login' button to prevent the form from being submitted.\n"
      "- Display an error message 'Incorrect password' on each attempt.\n"
      "- Dynamically add a 'Forgot password?' link next to the login button, but only after the first failed attempt.\n"
      "- Never display this link again if it is already visible.\n"
      "The JavaScript code must be self-contained and ready to be embedded within a <script> tag in the same HTML page.\n"
    )
  }
],
```



mathilde

.....

Se connecter

Mot de passe oublié ?

Annexe 4 : « /directory/config.ini » page

```
← ↻ 🏠 ⓘ 127.0.0.1:8000/directory/config.ini
🇧🇴 Boîte de réception (...) 🔄 CAS - Central Authe... 🌀 ChatGPT 🟡 C

[config]

; Database settings
database_host = localhost
database_user = admin
database_password = password123
database_name = mydb

; API settings
api_key = 1234567890abcdef
api_secret = 0987654321fedcba

; System settings
system_timezone = UTC
system_language = en_US.UTF-8

; Logging settings
log_level = INFO
log_file = /var/log/myapp.log
log_rotation = daily

; Security settings
security_firewall_enabled = yes
security_firewall_rules = 192.168.1.0/24, 10.0.0.0/16

; Backup settings
backup_interval = weekly
backup_location = /mnt/backups
backup_retention = 4 weeks
```

Annexe 5 : « /directory/setup.py » page

```
← ↻ 🏠 ⓘ 127.0.0.1:8000/directory/setup.py
🇧🇴 Boîte de réception (...) 🔄 CAS - Central Authe... 🌀 ChatGPT 🟡 Concours de

from setuptools import setup

setup(
    name='My Project',
    version='1.0',
    author='John Doe',
    author_email='john@example.com',
    description='A brief description of my project.',
    long_description='This is a longer description of my project.',
    packages=['my_project'],
    install_requires=[
        'requests>=2.25.1',
        'beautifulsoup4>=4.9.0'
    ],
    extras_require={
        'python_version=="3.8"': ['f-strings==1.5.0']
    },
    classifiers=[
        'Development Status :: 5 - Production/Stable',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: MIT License',
        'Programming Language :: Python :: 3'
    ],
    python_requires='>=3.6',
    project_urls={
        'Bug Reports': 'https://example.com/bugs',
        'Source Code': 'https://example.com/src'
    }
)
```

Annexe 6 : « /directory/docs/analysis.log » page

```
127.0.0.1:8000/directory/docs/analysis.log
Boîte de réception (...) CAS - Central Auth... ChatGPT Concours de progra... global exam Untitled Workspace... Netflix Accueil | Disney+ Stack Overflow - W...

Analysis Log
-----
2022-02-15 14:30:00 - User attempted to access restricted directory "/sensitive-data"
2022-02-16 10:45:00 - System alert triggered due to high CPU usage (85%)
2022-02-17 12:00:00 - Anomalous network traffic detected from IP address 192.168.1.100
2022-02-18 09:15:00 - User logged in with invalid credentials (username: admin, password: password123)
2022-02-19 11:30:00 - System crash reported due to overheating (GPU temperature reached 95°C)

Security Incidents
-----
* Date: 2022-01-01
* Time: 12:00:00
* Description: Unauthorized access to sensitive data was detected. The incident was contained and the affected system was isolated.
* Resolution: The vulnerability was patched, and additional security measures were implemented to prevent similar incidents in the future.

System Performance Issues
-----
* Date: 2022-02-10
* Time: 14:00:00
* Description: System performance issues were reported due to high CPU usage. The issue was resolved by restarting the system and updating the operating system.
* Resolution: The system was upgraded with additional RAM, and a monitoring tool was implemented to detect similar issues in the future.

Network Anomalies
-----
* Date: 2022-02-05
* Time: 10:00:00
* Description: Anomalous network traffic was detected from an unknown IP address. The incident was contained, and the affected system was isolated.
* Resolution: The vulnerability was patched, and additional security measures were implemented to prevent similar incidents in the future.

User Activity
-----
* Date: 2022-02-20
* Time: 09:00:00
* Description: User logged in with valid credentials (username: user123, password: password456)
* Resolution: The login was successful, and the user was granted access to the system.
```

Annexe 7 : « /directory/performance/results/data.csv » page

```
127.0.0.1:8000/directory/performance/results/data.csv
Boîte de réception (...) CAS - Central Auth... ChatGPT Concours de prog

Server load averages for the past week:

Date,1min,5min,15min
2022-09-01,0.12,0.05,0.03
2022-09-02,0.11,0.04,0.02
2022-09-03,0.13,0.06,0.04
2022-09-04,0.10,0.03,0.01
2022-09-05,0.12,0.05,0.03
2022-09-06,0.11,0.04,0.02
2022-09-07,0.13,0.06,0.04

Average CPU usage for the past week:

Date,CPU avg
2022-09-01,12%
2022-09-02,11%
2022-09-03,15%
2022-09-04,10%
2022-09-05,12%
2022-09-06,11%
2022-09-07,13%

Memory usage for the past week:

Date,Mem avg
2022-09-01,45%
2022-09-02,44%
2022-09-03,50%
2022-09-04,42%
2022-09-05,46%
2022-09-06,43%
2022-09-07,48%

Disk usage for the past week:

Date,Used %
2022-09-01,65%
2022-09-02,64%
2022-09-03,70%
2022-09-04,62%
2022-09-05,66%
2022-09-06,63%
2022-09-07,68%
```