University of Piraeus

SCHOOL OF INFORMATION AND TELECOMMUNICATIONS TECHNOLOGIES

Department of Digital Systems

THESIS

Development of Capture the Flag Challenges in the Web Application Category

Kouretas Panagiotis

Supervising Professor:

Christos Xenakis, (Professor)

PIRAEUS

APRIL 2024

FINAL YEAR PROJECT

Capture the Flag web exploitation challenges

Kouretas Panagiotis
A.M.: e19072

ABSTRACT

This thesis details the creation and development of web exploitation Capture the Flag (CTF) challenges and it is structured to guide participants through the process of solving them. It begins with an overview of CTF challenges that sets the stage for a deeper exploration into seven specifically designed challenges across a spectrum of three difficulty levels: easy, medium, and hard. The complexity of each challenge increases depending on its difficulty level. For each challenge, the thesis outlines the theoretical background that is necessary for understanding and solving the problem, the skills participants can expect to develop, a detailed solution process, and insights into the challenge's creation.

SUBJECT AREA: Web exploitation

KEYWORDS: CTF, Cybersecurity, Web

ACKNOWLEDGMENTS

At this point, I would like to extend my deepest gratitude to my supervisor, Professor Christos Xenakis, whose expertise and encouragement inspired me to delve deeper into the field of cybersecurity. I am also profoundly grateful to the CTFLib team for their dedication and the resources they provided, which were crucial to the success of my thesis. Special appreciation to Thanasis Grammatopoulos, whose support and insights were invaluable at every stage of my thesis and his contribution went beyond guidance to truly shaping the outcome of this research. Each of you has played a pivotal role in this journey, and for that, I will always be thankful.

Contents

1. Introduction

1.1 Capture-The-Flag (CTF) challenges

In the rapidly evolving digital age, cybersecurity has emerged as a critical area of concern for individuals, organizations, and nations worldwide. The need for qualified experts who can safeguard and defend against these threats is greater than ever as cyber threats become more complex. An inventive method of instruction and training is offered by integrating Capture-The-Flag (CTF) challenges into the cybersecurity learning landscape.

Capture-The-Flag (CTF) challenges are competitions that simulate real-world cybersecurity problems within a controlled and properly configured environment. These challenges, which combine training and education, give participants the chance to interact directly with a range of cyber threats and gain practical experience in identifying, analyzing, and mitigating them.

CTFs are distinguished into two broad formats as 'Jeopardy-style' and ‘Attack-Defend’ challenges. The first one, which is the most common, involves solving discrete challenges across different categories. In the second one, teams must penetrate in real time other teams’ systems while protecting their own.

Some of the most popular 'Jeopardy-style' challenges’ categories are Digital Forensics, Web Exploitation, Cryptography, Reverse Engineering and Binary Exploitation. Furthermore, they are also separated according to their difficulty level as easy, medium and hard challenges. Corresponding to the complexity of each challenge, there are points assigned that reflect its difficulty.

Because of their game-like approach, CTFs have spurred the creation of many platforms that are hosting these challenges from multiple categories. These platforms involve participants’ ranking scoreboards. This not only makes learning more fun and immersive, but also more interactive and competitive.

Figure 1 1 - Example of HackTheBox platform's scoreboard ranking. [1]

Each one of the developed challenges will have a specific format for the corresponding hidden flag and it will be the following: “CTFLIB{example-flag}”. Moreover, these challenges can be found on CTFLib’s website [2] as depicted in Figure 1.2. which is a basic part of the Cybersecurity Master’s Degree Program of the University of Piraeus.

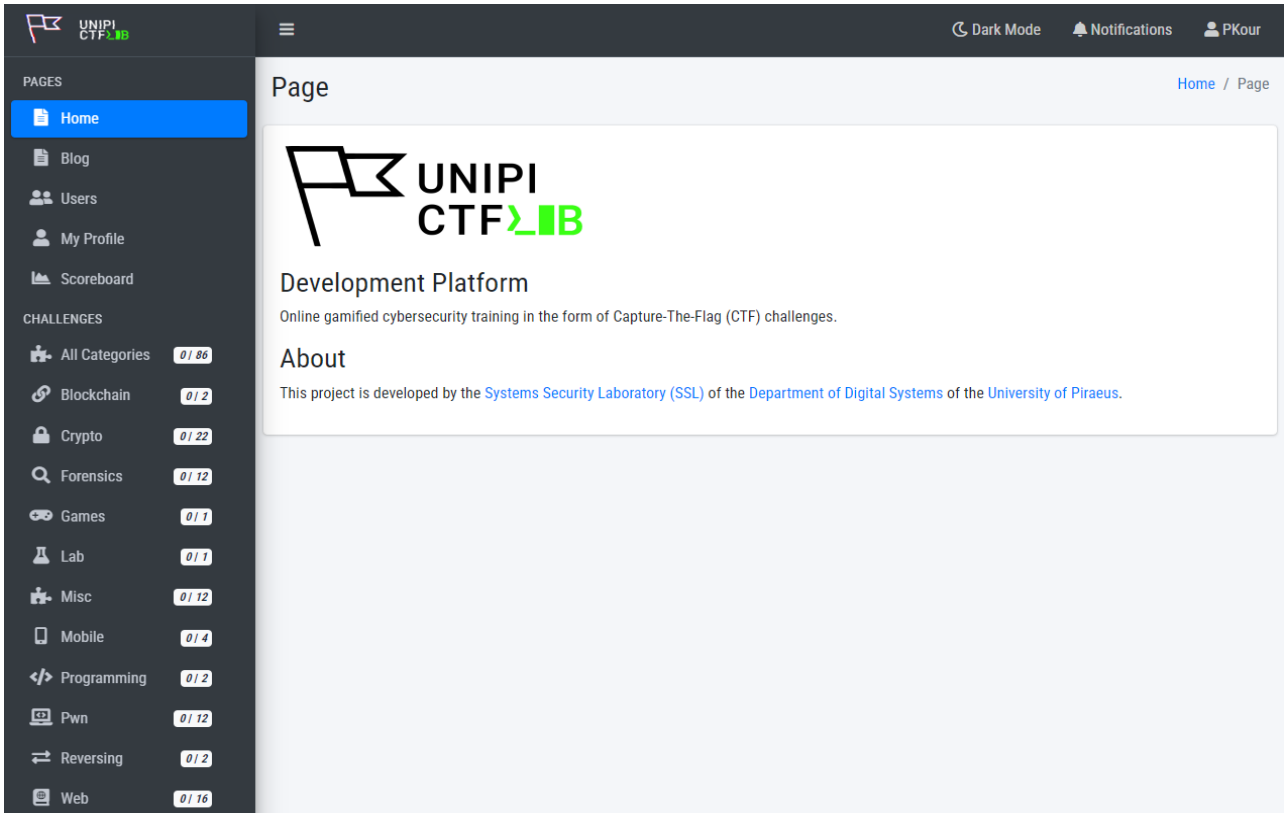


Figure 1 2 – Website of CTFLib’s platform.

1.2 Structure of the Thesis

This thesis outlines the creation and deployment of seven Web Capture-The-Flag (CTF) challenges across a spectrum of three difficulty levels easy, medium and hard. The following chapters are properly constructed to introduce, analyze, and draw the appropriate conclusions about each challenge and its respective security threats that demonstrates.

The Chapters of the Thesis are the following:

- Chapter 2: Easy Challenges (NoCrypt Keys, Mystic Ensign Emporium)
- Chapter 3: Medium Challenges (Digital Art Gallery, V4ln Design, Sci-Fi NameGen)
- Chapter 4: Hard Challenges (Virtual Invitation Creator, Jedi Archives)
- Chapter 5: Conclusion
- Chapter 6: References

2. Easy Challenges

This chapter explores the basic principles underlying the easy difficulty level challenges. It provides an in depth look at the necessary theoretical background, demonstrates the methodologies the participants need to follow to solve each challenge and showcases its creation process.

There are two easy challenges that demonstrate concepts such as the following:

- NoSQL injection
- Access control vulnerabilities with Insecure Direct Object References (IDOR)

2.1 NoCrypt Keys

2.1.1 Challenge Overview

This “Easy” level challenge presents a practical demonstration of a NoSQL injection vulnerability within a web application's authentication mechanism. Participants are tasked with bypassing a login page and signing in without having an account to acquire the hidden flag. The vulnerability of the challenge lies in the application’s improper validation and sanitization of the user input since the NoSQL query uses the provided input directly. This oversight allows the attackers to manipulate this query by injecting malicious inputs and getting unauthorized access.

2.1.2 Skills Acquired

This challenge will help the participants acquire several key skills.

- In-depth understanding of NoSQL injection vulnerabilities
- Practical Security Assessment
- Critical Thinking and Problem-Solving
- Awareness of secure code practices (e.g. input validation and sanitization)
- Mitigate NoSQL injection vulnerabilities
- Exploit NoSQL injection vulnerabilities

2.1.3 Background Theory

The background theory for this challenge encompasses a basic understanding of how the NoSQL databases and queries work.

NoSQL databases are a broad category of database management systems. Compared to traditional relational databases, they have different designs, querying mechanisms, and data storage approaches. These databases frequently compromise the tight data consistency offered by SQL databases in order to achieve their high performance, scalability, and flexibility goals. Also, NoSQL databases are particularly popular for handling large volumes of unstructured or semi-structured data.

There are four common types of NoSQL databases as demonstrated in Figure 2.1.1.

- Document databases
- Key-Value stores
- Wide-column stores
- Graph databases

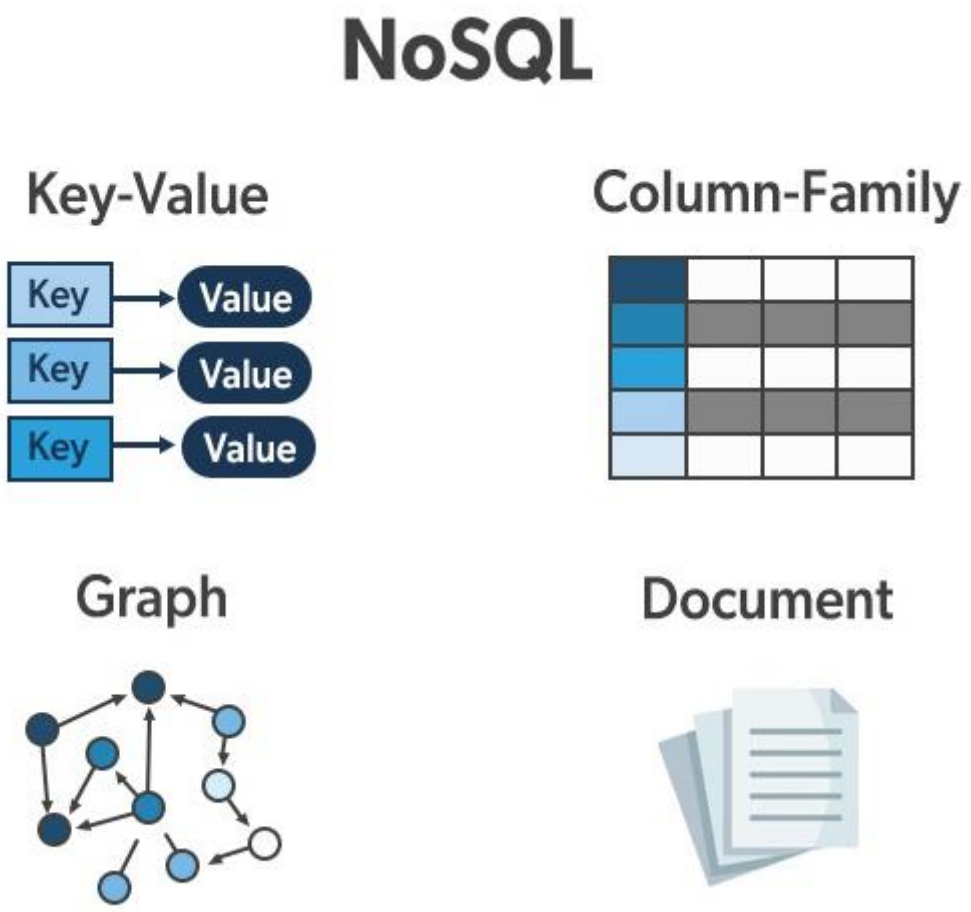


Figure 2.1.1 – Types of NoSQL databases

Depending on which type of database our application is using, the querying of the database varies significantly. Document and key-value stores often use simple key-based retrieval or JSON-like queries. Wide-column stores use a table-like structure but with more flexibility in the column family design, and graph databases use graph-specific queries to explore relationships between nodes.

NoSQL injection vulnerabilities arise when an application improperly validates and sanitizes the user input before passing it to a NoSQL database query. Exploiting these vulnerabilities by an attacker can lead to unauthorized access, bypass authentication mechanisms, or perform malicious actions on a web application. Since NoSQL databases do not use a standard querying language and their syntax varies widely among different types of databases, the injection techniques and payloads can be quite diverse.

There are two different types of NoSQL injection. The first one is the Syntax injection, in which you can break the NoSQL query syntax, enabling you to inject your own payload. This type is very similar to SQL injection vulnerability. The second one is the Operator injection, in which you can use NoSQL query operators (e.g. $ne, $gt, $where, etc.) to manipulate queries as depicted in Figure 2.1.2.

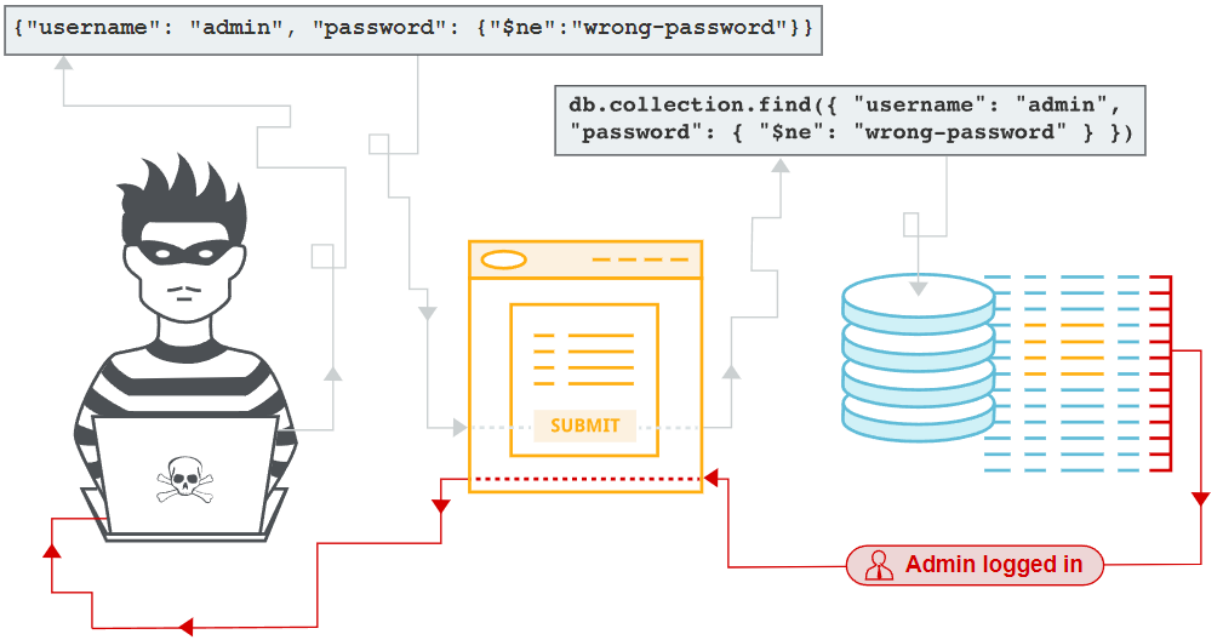


Figure 2.1.2 - Example of a NoSQL operator injection to bypass an authentication. [3]

The appropriate way to prevent NoSQL injection attacks depends on the specific NoSQL technology that is in use. However, you can substantially mitigate the risk of such attacks by following these recommended broad guidelines:

- Sanitize and validate all user inputs.
- Insert user input using parameterized queries instead of concatenating user input directly into the query.
- To prevent operator injection, apply an allowlist of accepted keys

2.1.4 Solution

First of all, we deploy the challenge from CTFLib.

Then, as seen in Figure 2.1.3., we are sent to the challenge's home page at http://localhost:4242/ which looks like a common login page.

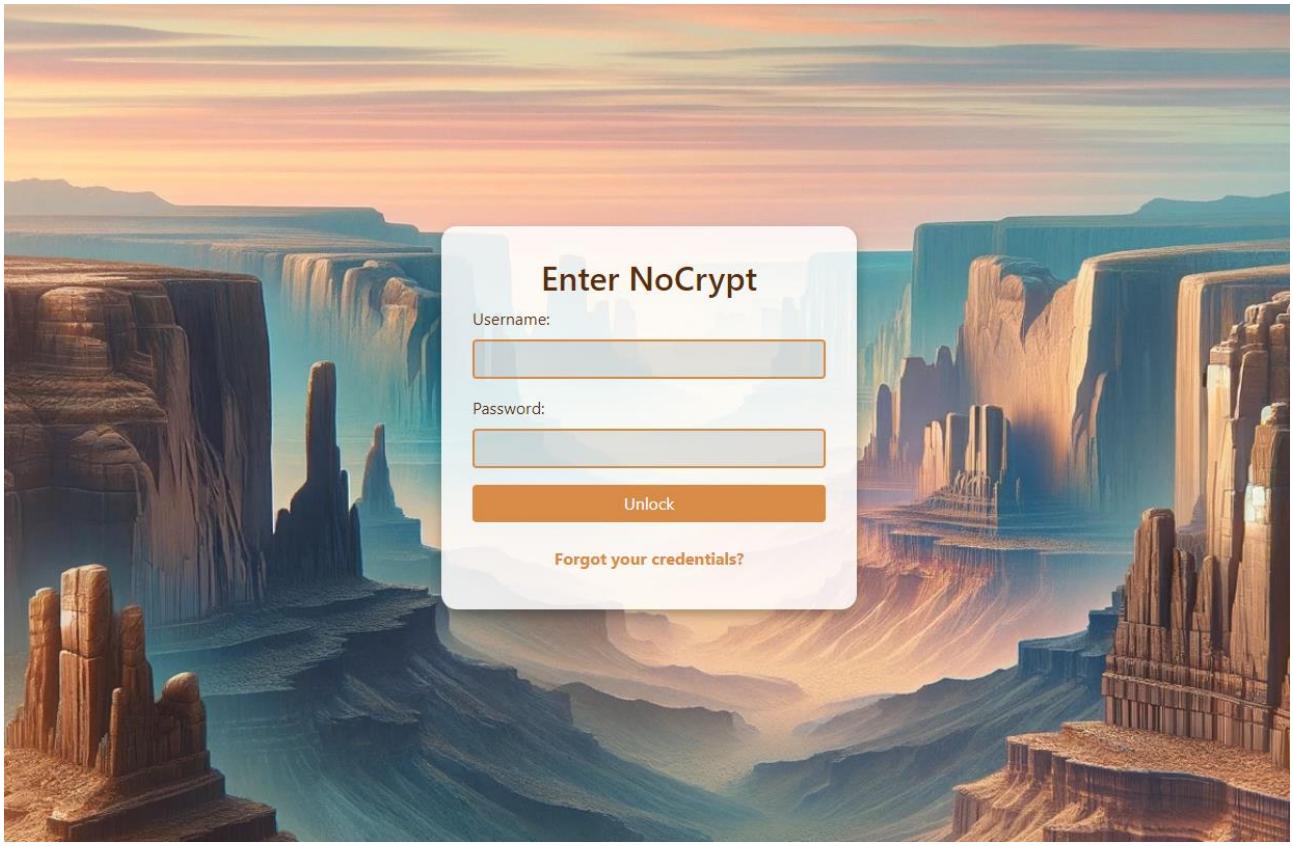


Figure 2.1.3 - Challenge's Home Page / Login Page.

With a quick look around at the page's source code (pressing "CTRL + U" on the page) I couldn't find any helpful information or hints.

So, I tried testing some possibly valid credentials to bypass this login page. I used many combinations (“Username: admin” and “Password: admin” etc.) and every time I got the same error response as depicted in Figure 2.1.4.

Figure 2.1.4 - Trying to login without an account.

When I tried clicking the “Forgot your credentials?” button, this is the response I got as seen in Figure 2.1.5.

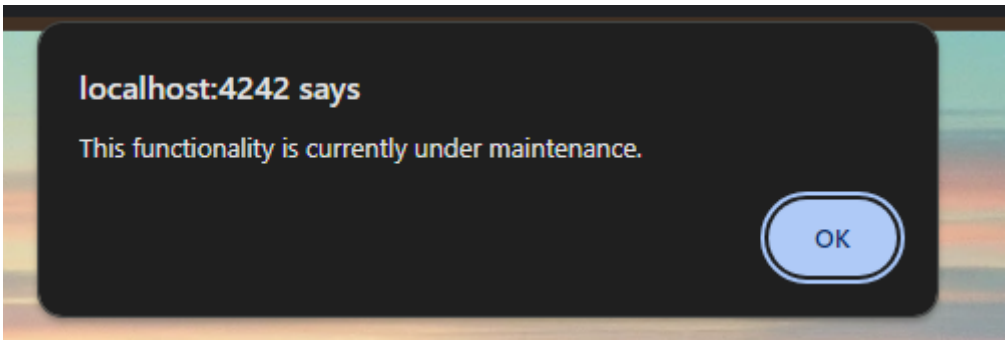


Figure 2.1.5 - Functionality error message

So far, we have no guidelines to move forward. However, upon closer inspection of the challenge description, we can notice that a few keywords seem to be implying something helpful.

Challenge description:

"Traverse the enigmatic world of NoCrypt, where Jason's Legacy holds the key to unlocking arcane data vaults. Compose the sequel to this cryptic tale by decoding the messages concealed in plain sight."

There are three keywords in this description “NoCrypt”, “Jason” and “sequel”. Further consideration of these terms leads us to the NoSQL injection vulnerability since NoSQL databases like MongoDB or other NoSQL databases interpret JSON-style queries.

Let's now look for NoSQL injection payloads we can use to bypass this login page. I discovered a GitHub repository with a variety of payloads that can be useful in our situation.

- https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/NoSQL%20Injection

After running a few tests, I ended up with the following payload:

{"username": {"$ne": null}, "password": {"$ne": null}}

Let's break it down a little bit.

{"username": {"$ne": null}} is a query that can be used in NoSQL databases to specify a condition where the username field should not be equal to null ($ne which stands for "not equal"). When this is part of a login query, it essentially says "find me a user whose username is anything but null". Since most usernames in a database are indeed not null, this condition will be true for all users.

{"password": {"$ne": null}} applies the same logic to the password field. This means "find me a user whose password is anything but null". Again, this will be true for all users with a password set.

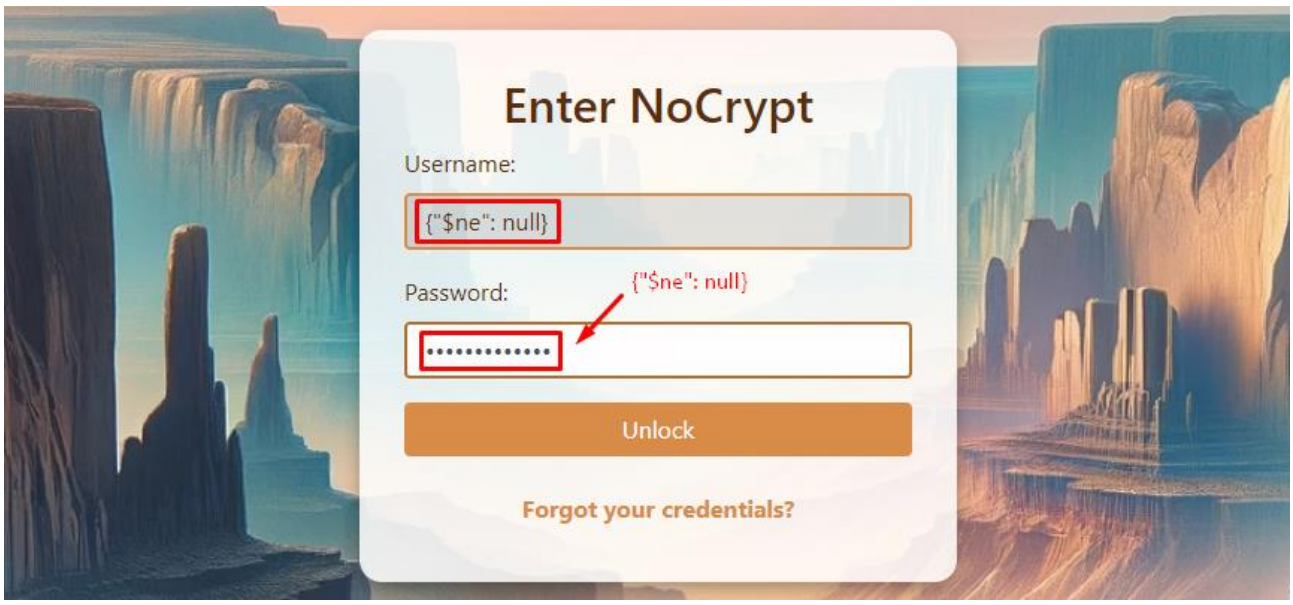


Figure 2.1.6 - Payload execution.

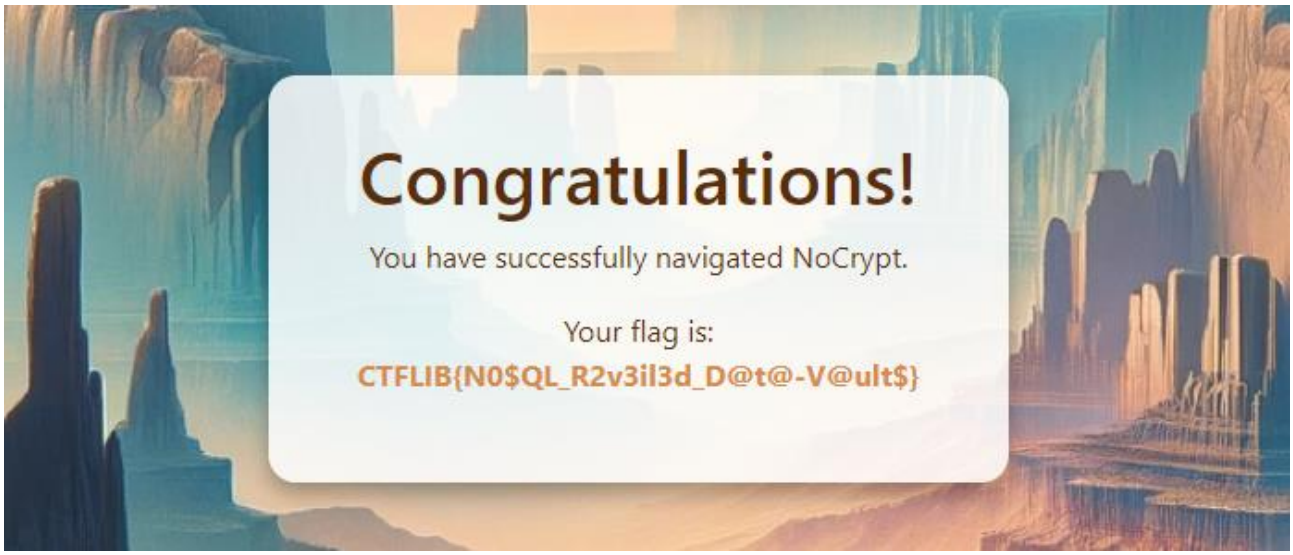


Figure 2.1.7 - Hidden Flag Successfully Retrieved.

GOT IT, we successfully bypassed the login page and retrieved the hidden FLAG:

CTFLIB{N0$QL_R2v3il3d_D@t@-V@ult$}

2.1.5 Creation

This challenge as it is referred to in chapter 2.1.1. Challenge Overview is just a simple authentication page that the participant needs to perform a NoSQL operator injection to bypass it.

Here is the Dockerfile of the challenge as seen in Figure 2.1.8.

Dockerfile

```
FROM python:3.8-slim-buster

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
ENV APP_PORT 4242

# Install MongoDB
RUN apt-get update && \
    apt-get install -y gnupg wget && \
    wget -qO - https://www.mongodb.org/static/pgp/server-4.4.asc | apt-key add - && \
    echo "deb http://repo.mongodb.org/apt/debian buster/mongodb-org/4.4 main" | tee /etc/apt/sources.list.d/mongodb-org-4.4.list && \
    apt-get update && \
    apt-get install -y mongodb-org && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Create MongoDB data directory
RUN mkdir -p /data/db

# Setup the application directory
RUN mkdir -p /opt/app
# Copy the application files to the container
COPY app/ /opt/app/
WORKDIR /opt/app

# Install dependencies defined in the requirements.txt file
RUN pip install -r requirements.txt --no-cache-dir

# Expose the application's port
EXPOSE 4242

# Define the command to run MongoDB and then start the Flask application
CMD mongod --fork --logpath /var/log/mongod.log; cat /var/log/mongod.log; /bin/sh /opt/app/serve.sh
```

Figure 2.1.8 - Challenge Dockerfile.

This Dockerfile defines python:3.8-slim buster as the container's base image which is a Debian Buster based image with Python 3.8. installed. Then the Python environment is configured, MongoDB is installed and a directory to store the data is created. Application files are copied into container's working directory, and the necessary dependencies (for Python & MongoDB) are installed. Port 4242 is exposed for the application. The image is configured to start MongoDB and then Flask application using a script (serve.sh).

```
serve.sh
#!/bin/sh

# Generate global variables
export APP_SECRET_KEY=$(python3 -c 'import secrets;print(secrets.token_hex(32))' 2>&1)

# Serve application
python3 -m gunicorn --bind 0.0.0.0:$APP_PORT app:app --workers 4
```

Figure 2.1.9 - serve.sh script for initializing the gunicorn service.

The serve.sh script is initializing the gunicorn service that deploys the flask application as depicted in Figure 2.1.9. Gunicorn is a web server that is used to serve Python web applications (including Flask) efficiently and easily by handling multiple requests simultaneously with multiple worker processes.

```
# Create a Flask app instance
app = Flask(__name__)

# Retrieve MongoDB URI from the environment variable or use default if not set
mongo_uri = os.environ.get('MONGO_URI', 'mongodb://localhost:27017/')
# Connect to MongoDB using the URI
client = MongoClient(mongo_uri)
# Select the database for the CTF challenge
db = client.ctf
# Select the collection within the database
users = db.users

1 usage
def initialize_db():
    # Initialize the database with a test user if the users collection is empty
    if users.count_documents({}) == 0:
        users.insert_one({"username": "testuser", "password": "password123"})

# Initialize the database on startup
initialize_db()
```

Figure 2.1.10 - Setup connection with the database and create a test user if users collection is empty.

Figure 2.1.10 illustrates the part MongoDB database is initialized with a test user if the users collection is empty. This setup phase ensures that there's at least one user in the database for participants to interact with, since we have no register functionality in the challenge.

```python
@app.route('/')
def index():
    return render_template('index.html')

@app.route( rule: '/login', methods=['POST'])
def login():
    # Extract username and password from form data
    username_input = request.form['username']
    password_input = request.form['password']

    try:
        # Attempt to parse input as JSON (for NoSQL injection possibility)
        username = json.loads(username_input)
        password = json.loads(password_input)
    except json.JSONDecodeError:
        # If not JSON-like, use the raw input
        username = username_input
        password = password_input

    # Vulnerable NoSQL query using user-provided input directly
    user = users.find_one({"username": username, "password": password})

    if user:
        # If a user is found, read the flag from a file and display it
        with open('flag.txt', 'r') as file:
            flag = file.read()
        return render_template( template_name_or_list: 'success.html', flag=flag)
    else:
        # If login fails, reload the index page with an error message
        return render_template( template_name_or_list: 'index.html', error="Login failed! Please try again.")
```

Figure 2.1.11 - Main routes of the challenge / Login route demonstrates the vulnerable NoSQL query.

The appropriate front-end pages are created (index.html, success.html). The backend of the application defines two main routes as depicted in Figure 2.1.11. The “/login” route implements the login functionality. Additionally, it demonstrates the NoSQL vulnerability, passing the user input straight into the query without any proper input validation or sanitization.

2.2 Mystic Ensign Emporium

2.2.1 Challenge Overview

This “Easy” level challenge introduces participants to the Mystic Ensign Emporium, a virtual flag shop specifically designed to highlight vulnerabilities in Access Control, with a particular focus on Insecure Direct Object References (IDOR). Participants will engage in an interactive learning experience where they must cleverly manipulate client-side controls, such as cookies and URL parameters, to gain unauthorized access to sensitive content. Through this hands-on approach, the challenge aims to foster a deeper understanding of common web vulnerabilities and the importance of robust security practices.

2.2.2 Skills Acquired

This challenge will help the participants acquire several key skills.

- In-depth understanding of Access Control mechanisms
- Identify and explore IDOR vulnerabilities
- Client-side control manipulation
- Critical Thinking and Problem-Solving
- Awareness of secure code practices (e.g. server-side validation and authorization checks)
- Prevent / Mitigate IDOR vulnerabilities
- Exploit IDOR vulnerabilities

2.2.3 Background Theory

The background theory for this challenge encompasses a fundamental understanding of web application security, especially focusing on Access Control vulnerabilities and Insecure Direct Object References (IDOR).

Access control, at its core, is a security technique that regulates who or what can view or use resources in a computing environment. In web applications, access control mechanisms are crucial for ensuring that users can only access the resources and perform the actions for which they are authorized. Effective access control systems prevent unauthorized access to sensitive information and protect against malicious actions. There are several models of access control and many different ways to apply them depending on the web application’s requirements. [4]

Insecure Direct Object Reference (IDOR) is a type of access control vulnerability that is demonstrated when an application exposes a reference to an internal implementation object (e.g. file, directory, or database key) without carrying out sufficient authorization checks. IDOR is

typically met in web applications that use predictable or easily guessable object identifiers. Because of this vulnerability, attackers can manipulate these references (e.g., changing the ID in a URL) to gain unauthorized access to sensitive data or perform unauthorized actions.

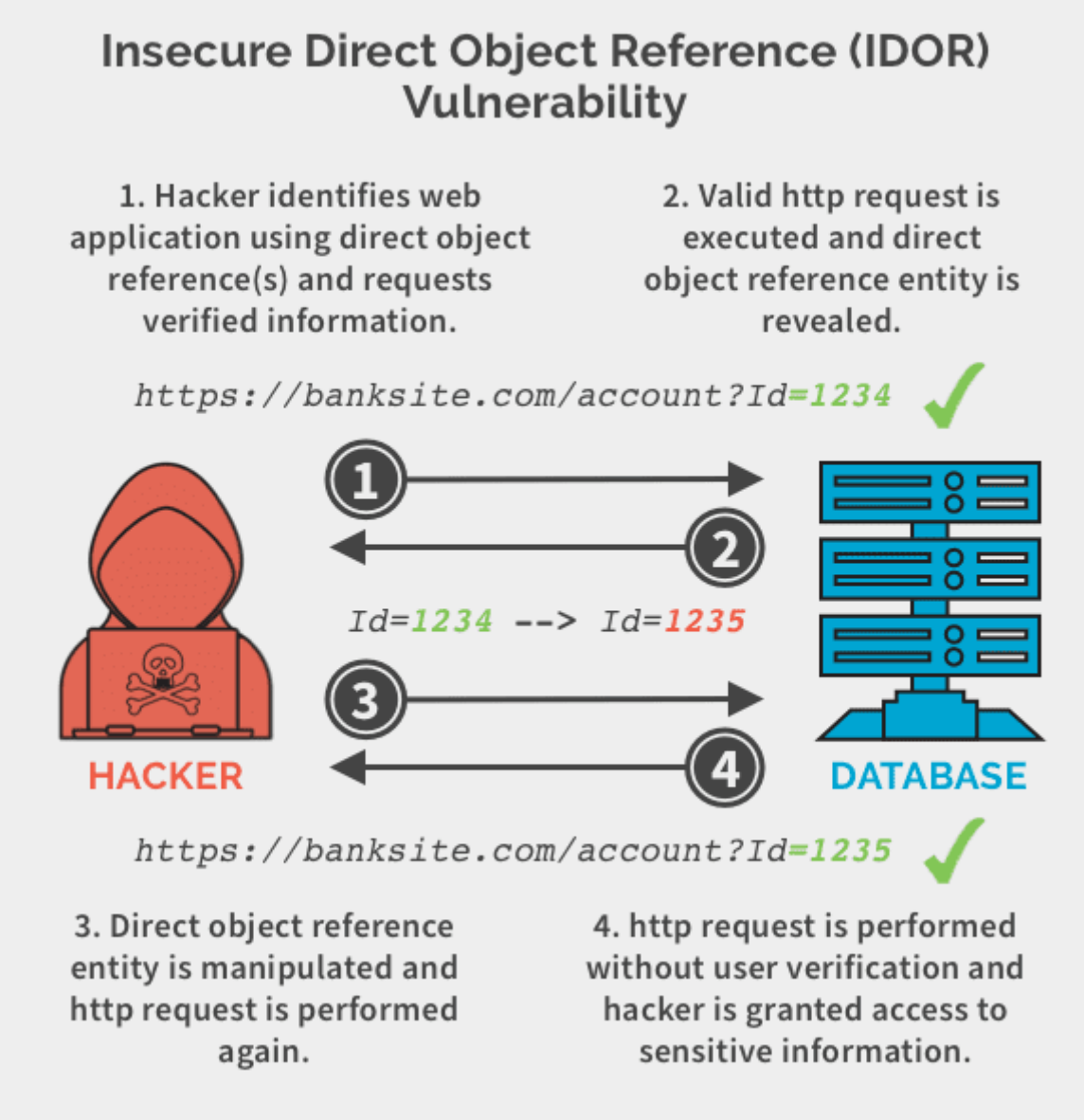


Figure 2.2.1 - Example of an IDOR vulnerability / Vulnerable account Id parameter on URL. [5]

Figure 2.2.1. depicts an example of an IDOR vulnerability. In this case, we notice that there is a vulnerable account Id parameter on the URL that can be modified by an attacker to gain unauthorized access to different users' accounts.

Mitigating IDOR vulnerabilities involves implementing robust access control checks and avoiding the exposure of direct object references to users. Also here are some good principles that would significantly help reduce these vulnerabilities.

- Using Indirect Object References: Replace direct object references with indirect references mapped server-side to the actual objects.
- Implementing Proper Authorization Checks: Ensure that each access to a resource involves a check to confirm the user has the authorization to access the requested resource.
- Adopting Least Privilege Principle: Limit user access rights and permissions to the minimum necessary to perform their functions.

2.2.4 Solution

First of all, we deploy the challenge from CTFLib.

Then we are redirected at http://localhost:4242/ which is the home page of the challenge as seen in Figure 2.2.2. It looks like a website that is selling “Mystic Ensigns”.

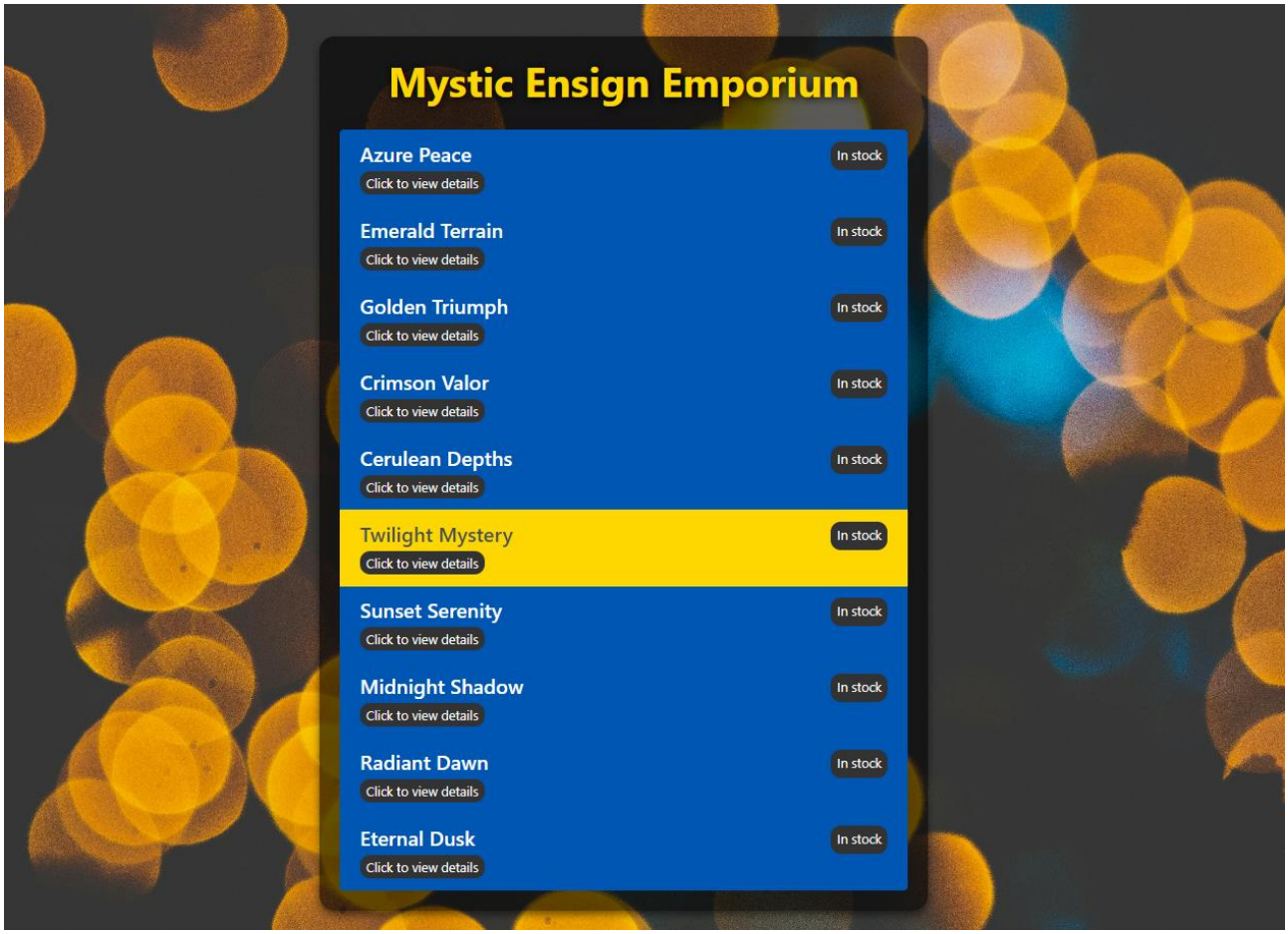


Figure 2.2.2 – Challenge’s Home Page / Mystic Ensign Shop.

When we click on one of the products we are redirected to a “flag-description” page that contains some information about the chosen flag.

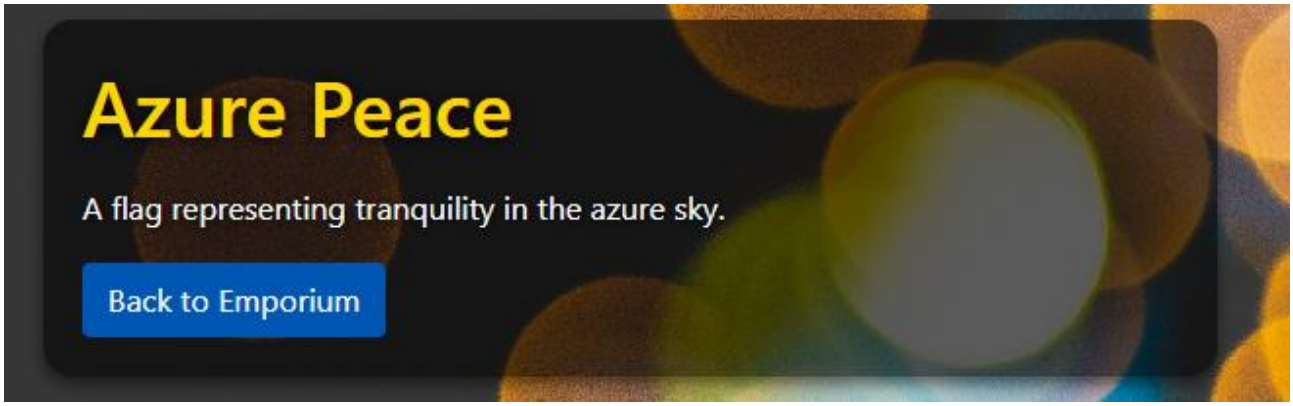


Figure 2.2.3 - Flag description page.

With a quick look around at the page's source code I couldn't find something helpful for the next step of the challenge. Also, the description of the challenge looks like it does not give any practical hints.

However, if we examine the "URL" of the "flag-description" page closer, we can see that there are two parameters. One for the id of the flag page "/flag/1" and one more for the stock of the corresponding flag "in_stock=True" as depicted in Figure 2.2.4.

When we try to change the "id" of the "flag-description" page from "1" to a number up to "10" we see that we can access the descriptions of all the available flags in the store.

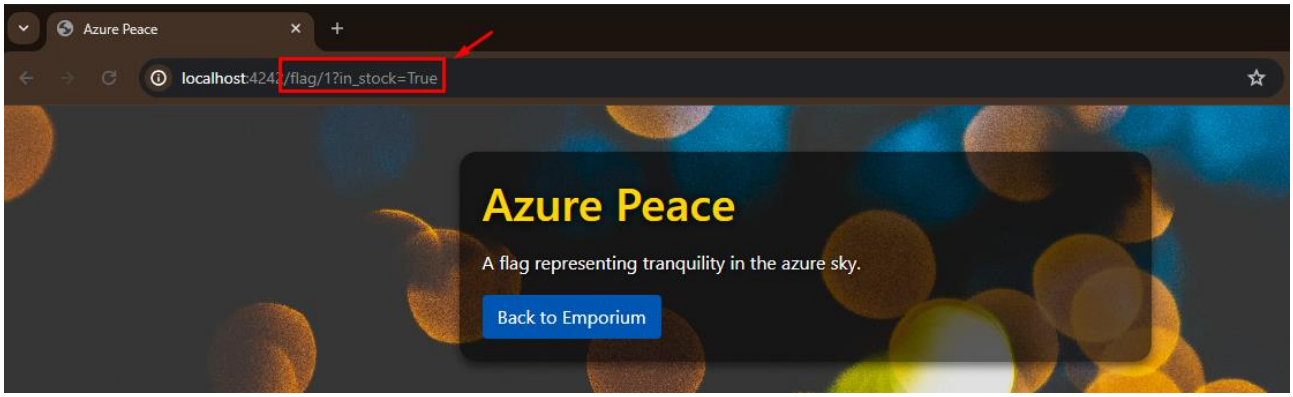


Figure 2.2.4 - Flag description page URL parameters.

So, let's experiment a little bit with the "id" parameter. I tried some numbers (>10) because 10 is the number of the currently available flags that are in stock, in case I could access any possible flag that may be unavailable and I got this response as seen in Figure 2.2.5.

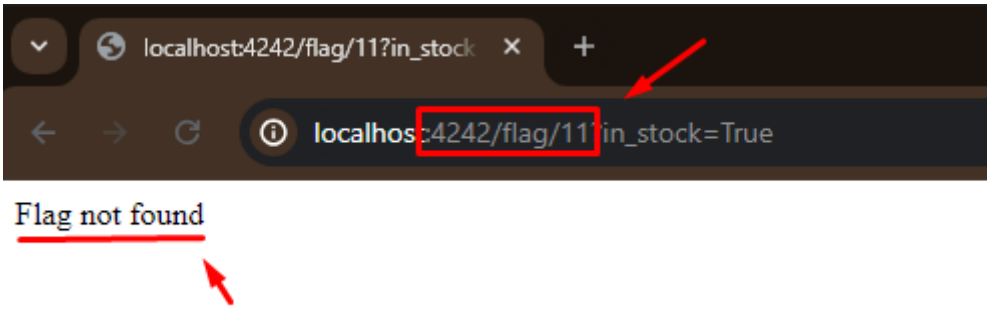


Figure 2.2.5 - Flag not found error message.

Upon further investigation, I noticed through the browser developer tools there is a cookie for this website named "show_hidden" that has the value "false".

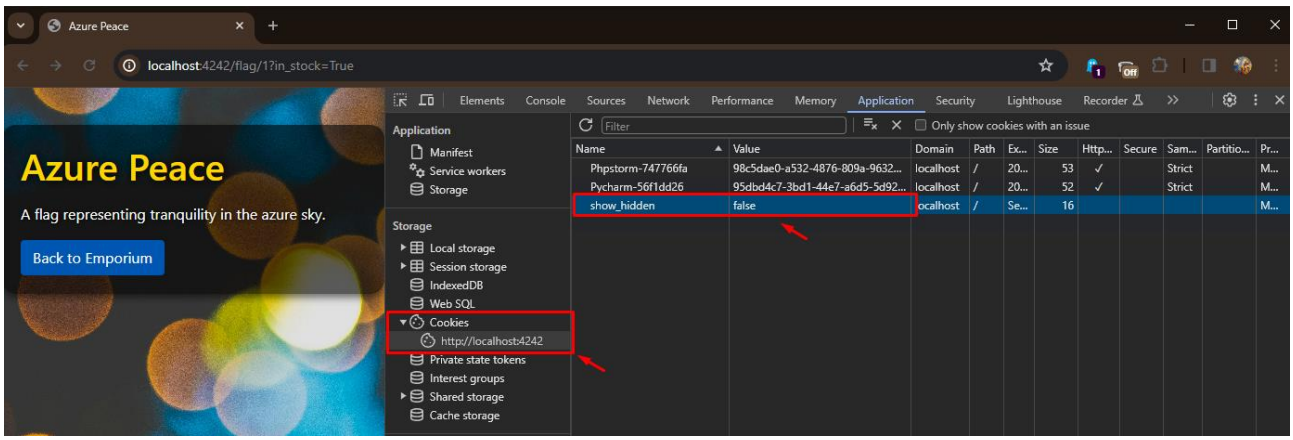


Figure 2.2.6 - Cookie value in Browser Developer tools.

I tried to manipulate this cookie, change its value and see what happens. After updating "show_hidden" to "true" and refreshing the page this is what was displayed on the home page of the store as depicted in Figure 2.2.7.

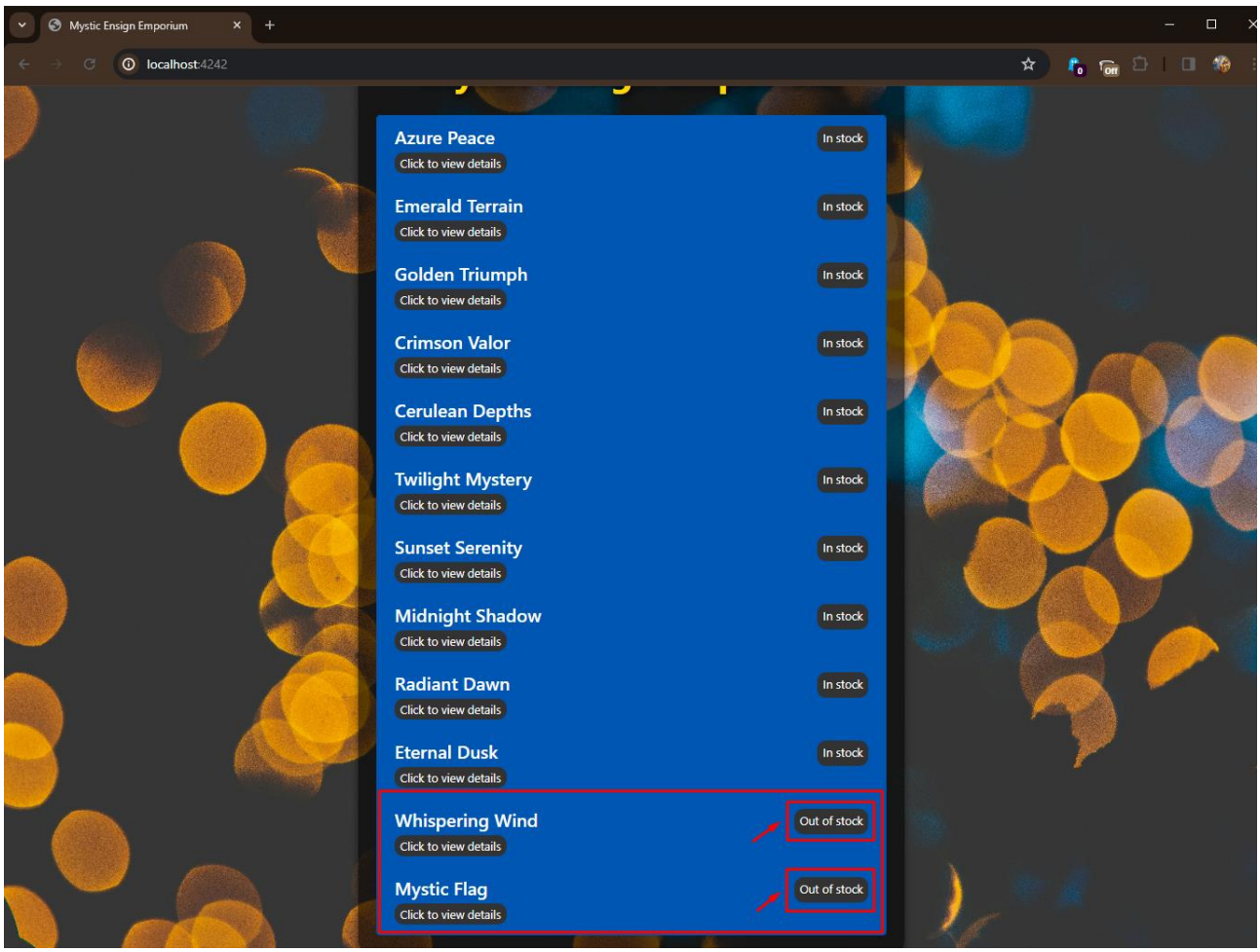


Figure 2.2.7 - Out of stock flags appeared in the shop.

Two new flags that are out of stock appeared in the shop. If we click on each one of them, we get the response that “This flag is not in stock” on the “flag-description” page as seen in the Figure 2.2.8.

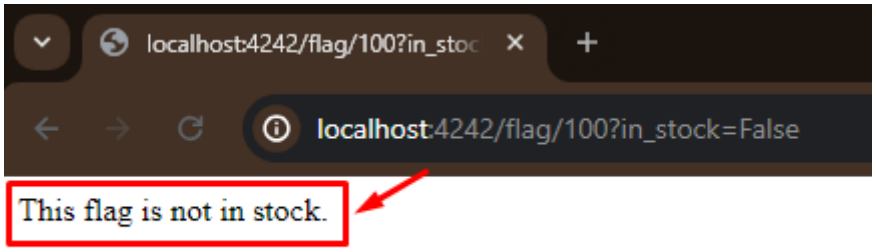


Figure 2.2.8 - Flag not in stock error message.

What if we try to manipulate the second parameter now?

Changing the stock parameter from “in_stock=False” to “in_stock=True”, we are getting back the description of the flag that was out of stock and not available for sale to the customers.

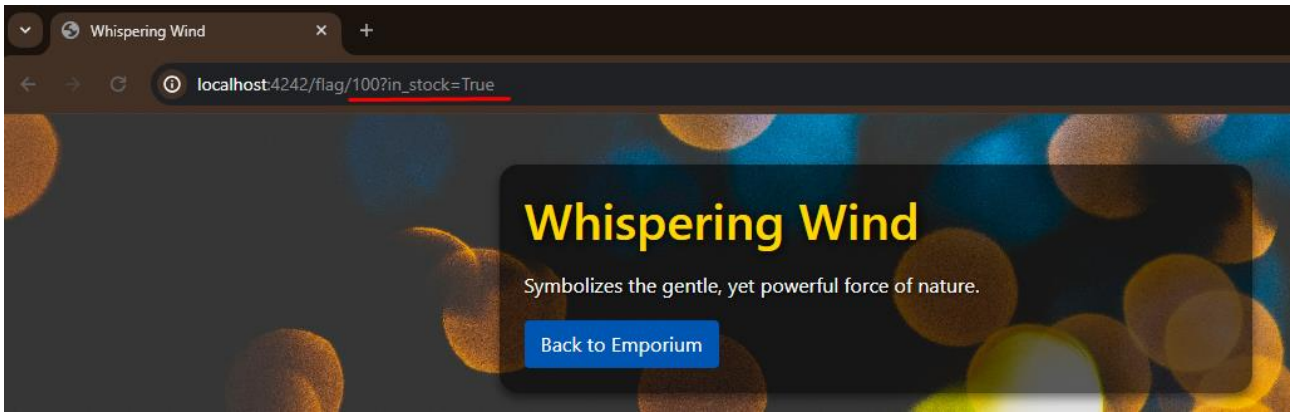


Figure 2.2.9 - Manipulating "in_stock" parameter.

Then we notice that this flag’s page id is “/flag/100”. Now let's try to see what the other unavailable flag displays on the description page if we update its stock parameter.

Figure 2.2.10 - Hidden flag successfully retrieved.

BANG, we found the hidden FLAG in the “Mystic Ensign Emporium”.

CTFLIB{Acc3ss_C0ntr0l_V4ln_Expl01t3d}

2.2.5 Creation

This challenge as it is referred in chapter 2.2.1. Challenge Overview is a flag shop website.

Here is the Dockerfile of the challenge as seen in Figure 1.8.

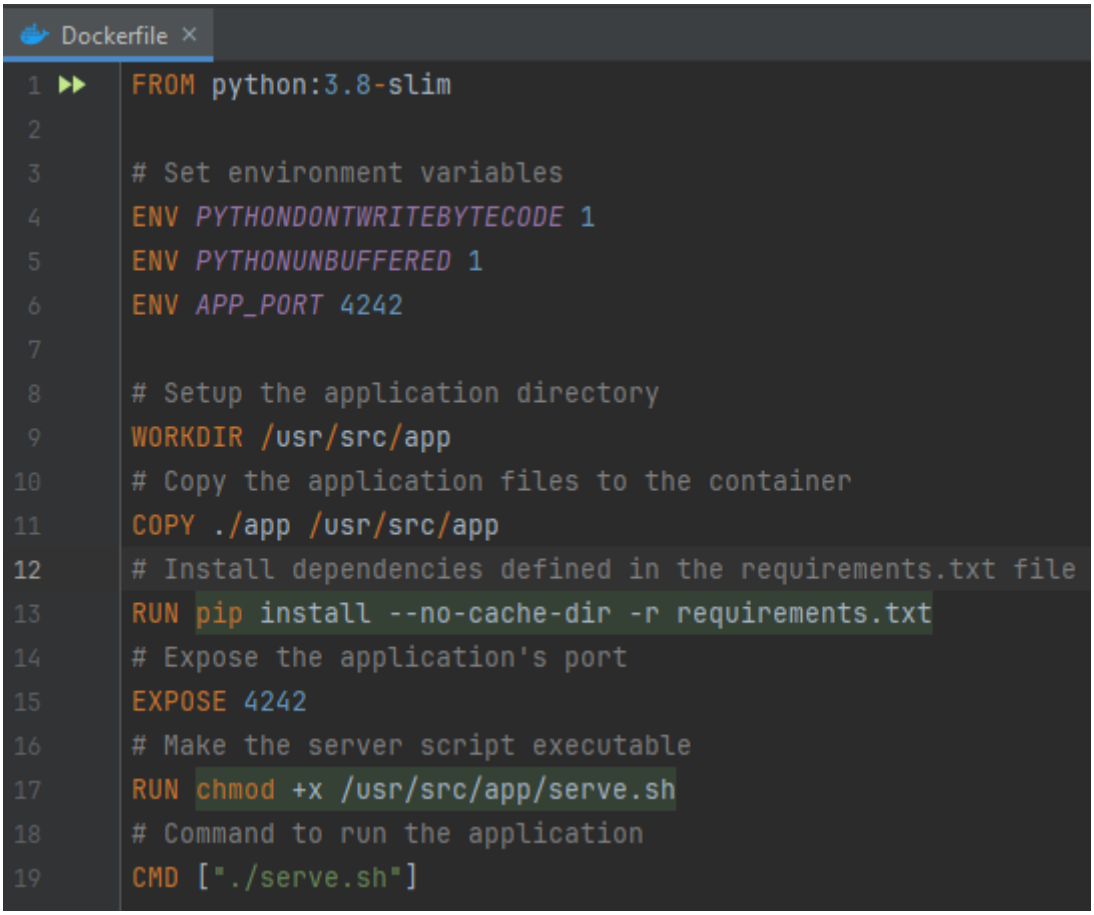


Figure 2.2.11 - Challenge Dockerfile

This Dockerfile creates a Docker container for a Python web application. It uses a Python 3.8 slim image, sets up environment variables for Python, copies the application files, installs dependencies, exposes port 4242, and configures serve.sh as the startup command to run the application.

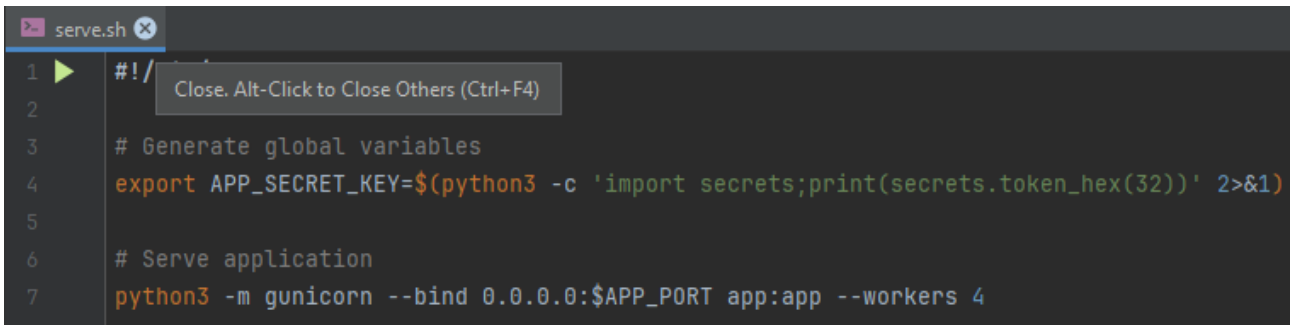


Figure 2.2.12 - serve.sh script for initializing the gunicorn service.

The serve.sh script is initializing the gunicorn service that deploys the flask application as depicted in Figure 2.2.12. Then we create the “app.py” which is the main backend file of the challenge.

```python
# Simulated data store for flags
flags = [
    {"id": 1, "name": "Azure Peace", "description": "A flag representing tranquility in the azure sky.", "in_stock": True},
    {"id": 2, "name": "Emerald Terrain", "description": "Symbolizes the lushness of vast green lands.", "in_stock": True},
    {"id": 3, "name": "Golden Triumph", "description": "Marks the victories achieved in golden times.", "in_stock": True},
    {"id": 4, "name": "Crimson Valor", "description": "A flag of bravery and strength in red.", "in_stock": True},
    {"id": 5, "name": "Cerulean Depths", "description": "Emblematic of the deep, mysterious oceans.", "in_stock": True},
    {"id": 6, "name": "Twilight Mystery", "description": "Captures the essence of the enigmatic twilight.", "in_stock": True},
    {"id": 7, "name": "Sunset Serenity", "description": "Represents the calmness of a beautiful sunset.", "in_stock": True},
    {"id": 8, "name": "Midnight Shadow", "description": "A dark flag, symbolizing the secrets of the night.", "in_stock": True},
    {"id": 9, "name": "Radiant Dawn", "description": "Signifies new beginnings and hope.", "in_stock": True},
    {"id": 10, "name": "Eternal Dusk", "description": "A reminder of the beauty in endings.", "in_stock": True},
    {"id": 100, "name": "Whispering Wind", "description": "Symbolizes the gentle, yet powerful force of nature.", "in_stock": False},
    {"id": 101, "name": "Mystic Flag", "description": load_flag(), "in_stock": False}
]
1 usage
@app.route('/')
def index():
    # Determine if hidden flags should be shown based on the 'show_hidden' cookie
    show_hidden = request.cookies.get('show_hidden', 'false').lower() == 'true'
    # Filter flags to be displayed based on their stock status and the 'show_hidden' cookie
    available_flags = [flag for flag in flags if flag['in_stock'] or show_hidden]

    # Render the index page with the available flags and set the 'show_hidden' cookie accordingly
    response = make_response(render_template( template_name_or_list: 'index.html', flags=available_flags))
    response.set_cookie( key: 'show_hidden', 'true' if show_hidden else 'false')
    return response

# Route to display detailed description of a specific flag based on its ID
1 usage
@app.route('/flag/<int:flag_id>')
def flag_description(flag_id):
    # Find the flag by ID or return None if not found
    flag = next((f for f in flags if f['id'] == flag_id), None)
    if not flag:
        return "Flag not found", 404

    # Check if the flag is in stock based on query parameter; if not, access is forbidden
    in_stock = request.args.get('in_stock', 'false').lower() == 'true'
    if in_stock:
        return render_template( template_name_or_list: 'flag_description.html', flag=flag)
    else:
        return "This flag is not in stock.", 403
```

Figure 2.2.13 - app.py / Main backend file.

As seen in Figure 2.2.13. this file defines a set of flags, including a special "Mystic Flag" whose description is loaded from a file. The flags visibility is based on the “in_stock” parameter status. Also, a client-controlled cookie (show_hidden) is set. The homepage (index.html) displays flags based on their stock status and the cookie's value. A detailed flag description page (flag_description.html) is accessible only if the flag is considered "in stock" or the URL parameter is manipulated.

The application allows exploration of Access Control and IDOR vulnerabilities through client-side manipulations.

3. Medium Challenges

This chapter explores the basic principles underlying the medium difficulty level challenges. It provides an in depth look at the necessary theoretical background, demonstrates the methodologies the participants need to follow to solve each challenge and showcases its creation process.

There are three medium challenges that demonstrate concepts such as the following:

- File uploads
- Path traversal
- Remote Command Execution (RCE)
- XML external entity injection (XXE)
- Server-Side Request Forgery (SSRF)

3.1 Digital Art Gallery

3.1.1 Challenge Overview

This “medium” level challenge presents a file upload scenario within a digital art gallery web application, designed to expose vulnerabilities in handling file uploads. Despite basic security measures are already implemented (file extension deny list), there are still some potential weaknesses that the participants can identify and exploit to gain unauthorized access and perform malicious actions.

3.1.2 Skills Acquired

This challenge will help the participants acquire several key skills.

- In-depth understanding of file upload mechanisms
- Critical Thinking and Problem-Solving
- Awareness of secure code practices (e.g. validating both type and content of uploaded files)
- Prevent / Mitigate file upload vulnerabilities
- Exploit file upload vulnerabilities

3.1.3 Background Theory

The background theory for this challenge encompasses a fundamental understanding of file upload mechanisms.

File upload vulnerabilities arise when a web server permits users to upload files to its filesystem without sufficiently validating attributes such as filename, type, contents, or size. These improper security configurations allow attackers to upload arbitrary and potentially dangerous files that can cause remote code execution or include server-side script files. [6]

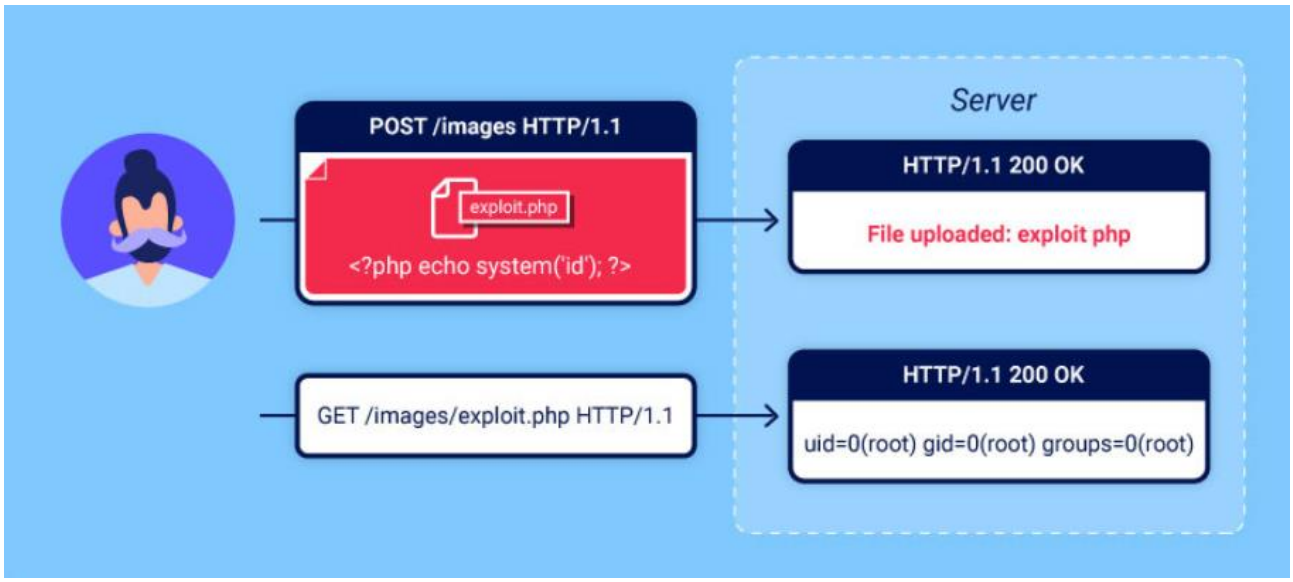


Figure 3.1.1 - Example of a file upload vulnerability.

Figure 3.1.1. illustrates an example of a file upload vulnerability. In this example, an attacker uploads a malicious PHP file that includes a command that is executed in the backend of the server and reveals the current user's identity information.

The impact of file upload vulnerabilities generally depends on two key factors.

- Which aspect of the file the website fails to validate properly, whether that be its size, type, contents, and so on.
- What restrictions are imposed on the file once it has been successfully uploaded.

Mitigating file upload vulnerabilities involves a multi-layered approach to ensure that uploaded files do not compromise the security of the web application or the server. Here are some good strategies to effectively reduce the risk associated with file uploads.

- Check the file extension against a whitelist of permitted extensions rather than a blacklist of prohibited ones.
- Make sure the filename doesn't contain any substrings that may be interpreted as a directory or a traversal sequence (../).
- Rename uploaded files to avoid collisions that may cause existing files to be overwritten.
- Do not upload files to the server's permanent filesystem until they have been fully validated.
- Limit file size and scan their content.
- Disable script execution in directories where files are uploaded through server configuration files.

3.1.4 Solution

First of all, we deploy the challenge from CTFLib.

Then we are redirected at http://localhost/index.php which is the home page of the challenge as seen in Figure 3.1.2. This is a “Digital Art Gallery” website.

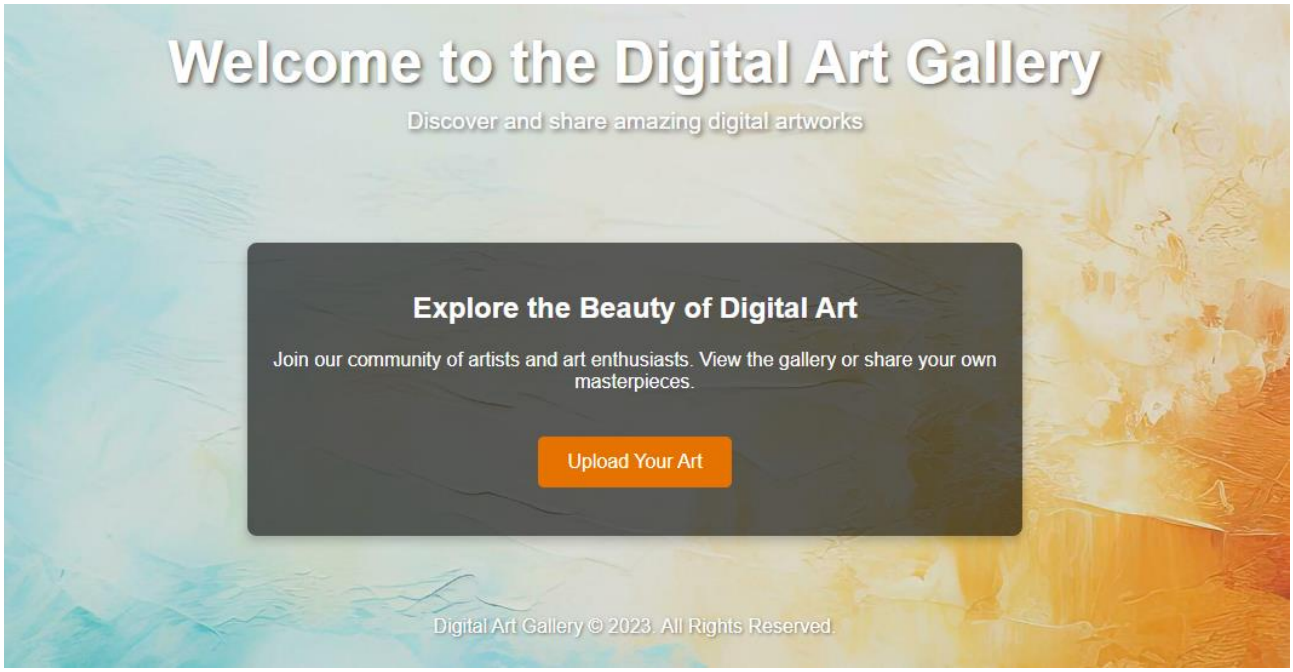


Figure 3.1.2 – Challenge’s Home Page / Digital Art Gallery.

When we click on the “Upload Your Art” button, we are transferred to http://localhost/upload_art.php as depicted in Figure 3.1.3. This page as indicated from the URL, appears to be a place we can upload our digital art pieces. Also, the URL unveils that this page is constructed using “PHP”.

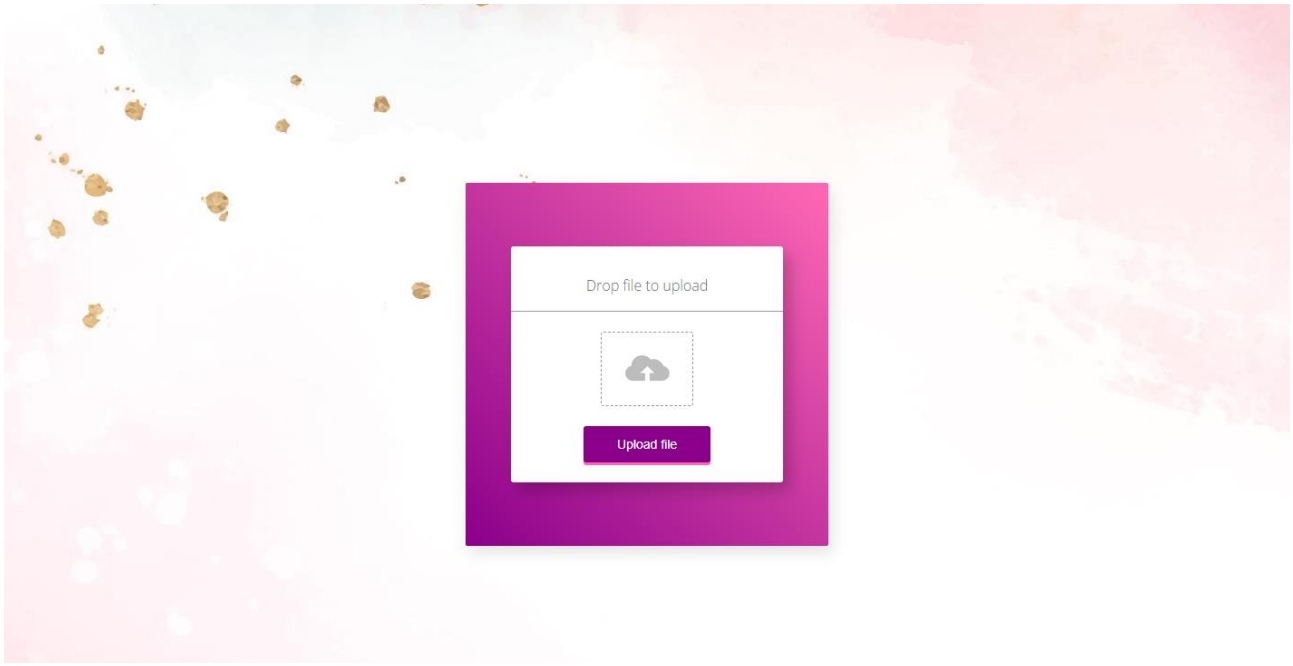


Figure 3.1.3 - Upload Art page.

Now let's try testing the upload art functionality of this page by uploading a simple image. After our image “obiwan.jpeg” is uploaded successfully as seen in Figure 3.1.4., we notice that we can access it at http://localhost/uploads/obiwan.jpeg.

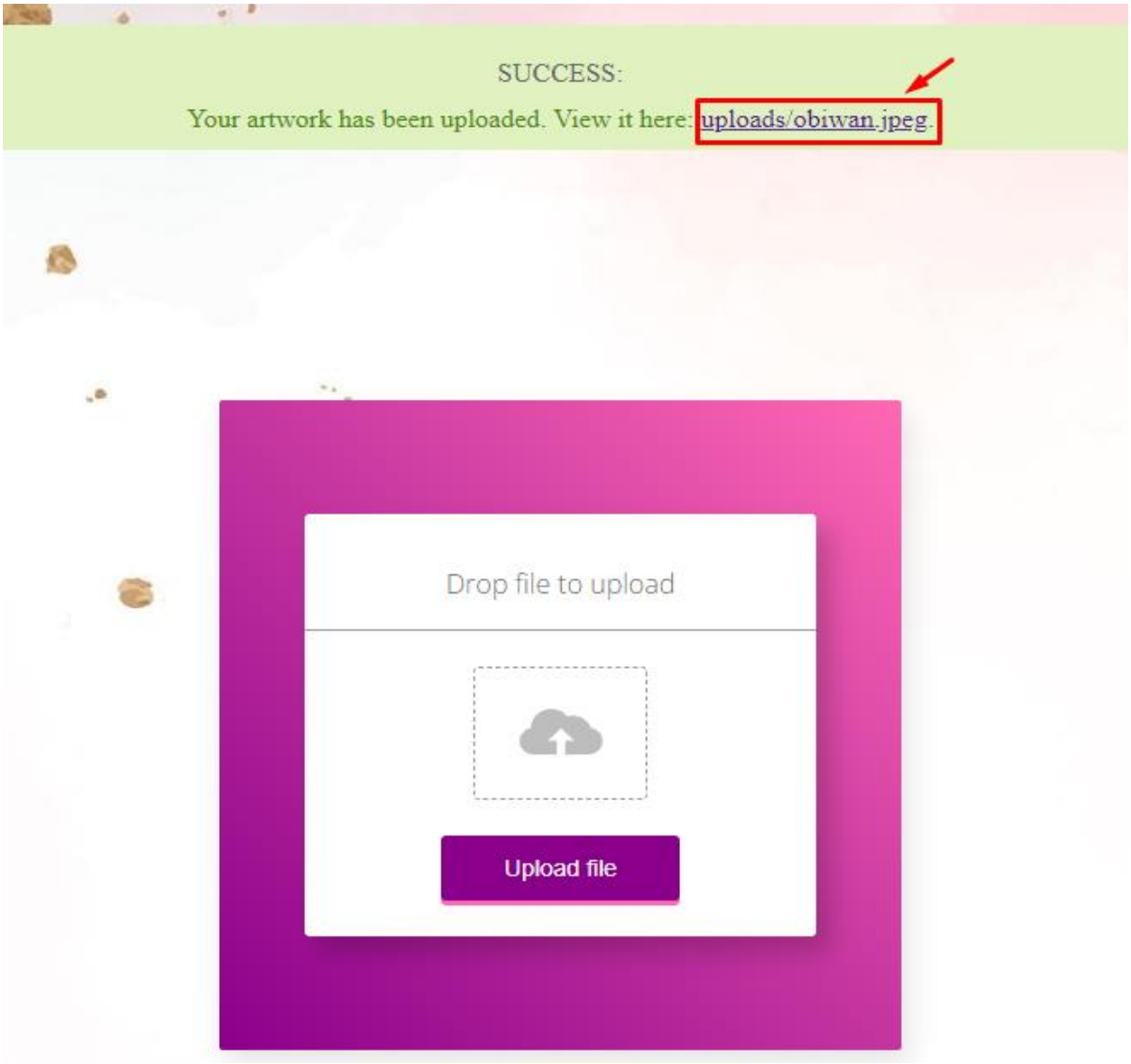


Figure 3.1.4 - Upload Success message.

However, if we try to access the http://localhost/uploads/ directory we get the following error as it's depicted in Figure 3.1.5.

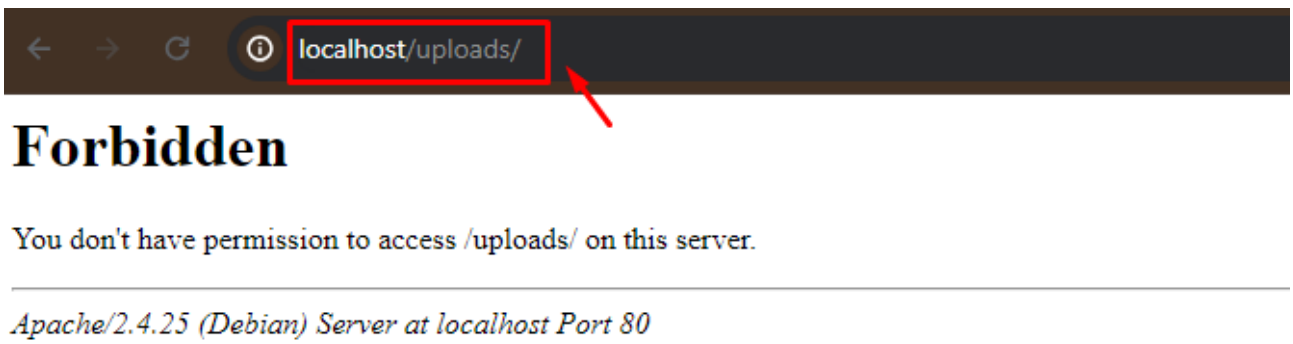


Figure 3.1.5 - Page Access error message.

Reading the description of the challenge one more time, I noticed that there is a note that indicates there are certain file extensions limited for the gallery's security. “Note: For the gallery's security, we've curated the canvas by limiting certain file extensions.”

After this, I tried to upload a PHP file including a simple “<?php phpinfo(); ?>” command as seen in Figure 3.1.6. This command is used to output a large amount of information about your PHP installation and can be used to identify installation and configuration problems.

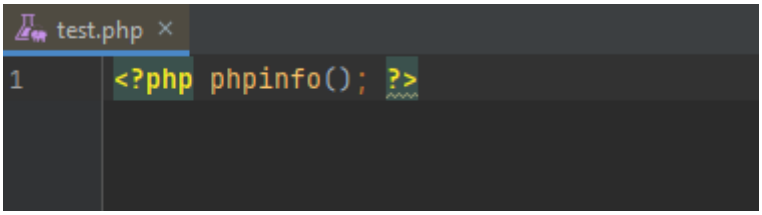


Figure 3.1.6 - Simple PHP script.

The reason I tried to upload such a simple PHP file is to test how the application responds to the different file extensions.

As expected, I got the error “This file type is not allowed” as depicted in Figure 3.1.7.

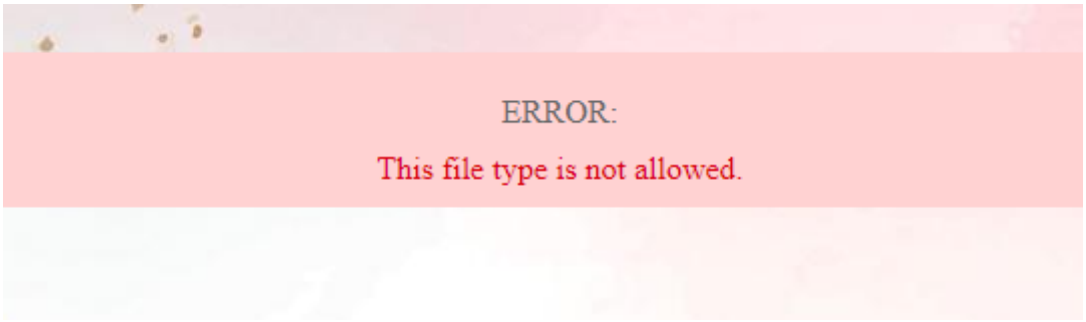


Figure 3.1.7 - File type is not allowed error message.

After trying many different PHP web shells I found online, I still couldn’t find a way to bypass this file extension limit.

However, while I was trying to figure out how this application works and checking the network traffic from the browser developer tools (by pressing the “F12” button on the browser), I distinguished a clue that could help me with my problem. The server used for this website was “Apache”.

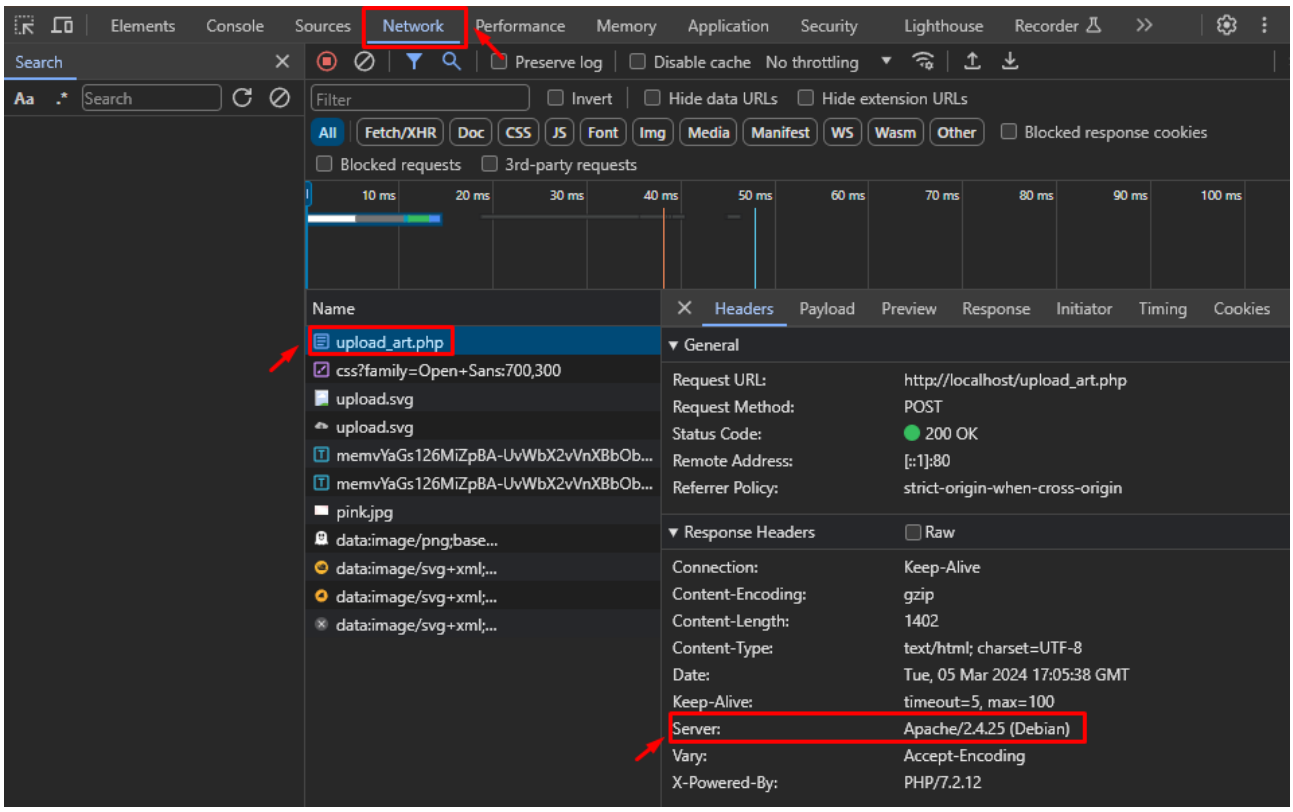
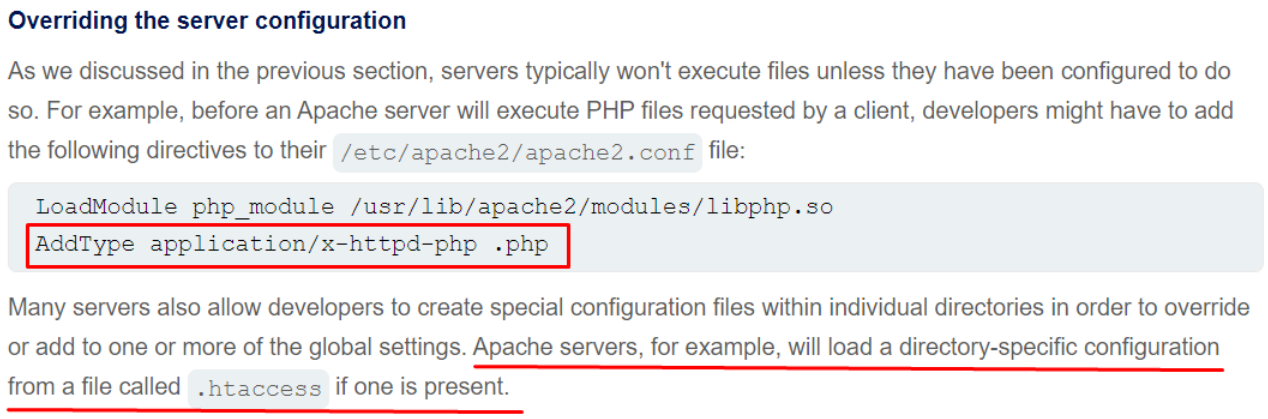


Figure 3.1.8 – Server used in browser developer tools.

Also, while I was searching about file upload vulnerabilities, I found this snippet of an article on "Portswigger" [6] that was indicating a possible trick to bypass blacklisted extensions via file upload when Apache server is used as seen in Figure 3.1.9.



Overriding the server configuration

As we discussed in the previous section, servers typically won't execute files unless they have been configured to do so. For example, before an Apache server will execute PHP files requested by a client, developers might have to add the following directives to their `/etc/apache2/apache2.conf` file:

```
LoadModule php_module /usr/lib/apache2/modules/libphp.so
AddType application/x-httpd-php .php
```

Many servers also allow developers to create special configuration files within individual directories in order to override or add to one or more of the global settings. Apache servers, for example, will load a directory-specific configuration from a file called `.htaccess` if one is present.

Figure 3.1.9 – Bypass blacklisted extension article on Portswigger.

This article is suggesting to upload an ".htaccess" file which is a configuration file used by Apache-based web servers. This file will contain the "AddType application/x-httpd-php .art" command that makes the application understand files with ".art" extension as files with ".php" extension and then upload a PHP web shell.

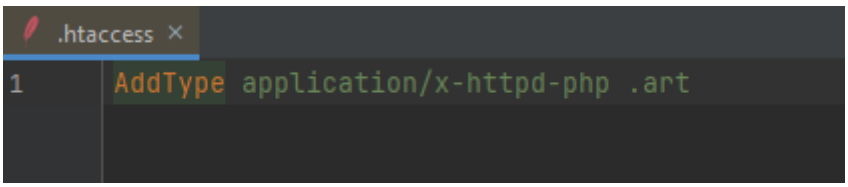


Figure 3 1 - Contents of .htaccess configuration file.

Since I uploaded this ".htaccess", the Apache configuration must have been updated.

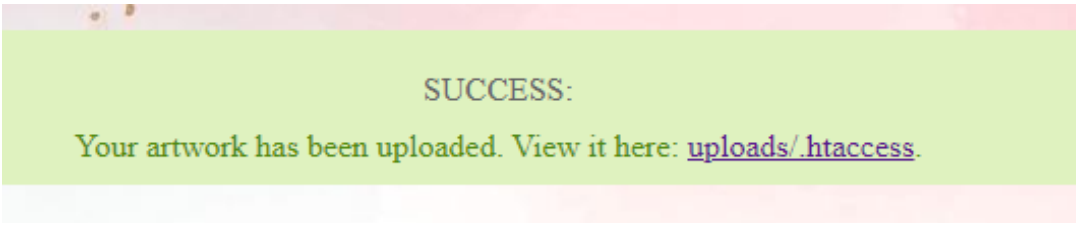


Figure 3.1.10 - .htaccess file successfully uploaded.

Right after that, I uploaded the new PHP web shell but this time as a "webshell.art" file.

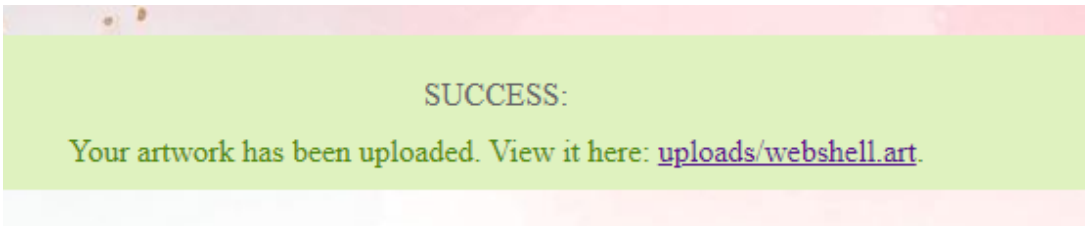


Figure 3.1.11 - webshell.art successfully uploaded.

Following some online research about which PHP web shell should I use in my situation, I found this online GitHub repository https://github.com/WhiteWinterWolf/wwwolf-php-webshell/blob/master/webshell.php that seems to have the best one. This web shell allows an attacker to run commands on a server with web hosting capabilities. It's designed to provide unauthorized access to the server's filesystem and execute commands remotely.

So, we head to the http://localhost/uploads/webshell.art page. There we select "/var/www/html/uploads" as the "CWD" which is the current working directory that we want to execute commands. For the "Cmd" which is the command we want to execute we will choose "ls" which lists the files of the chosen directory.

As depicted in Figure 3.1.12., there are listed all the files we uploaded in the gallery.

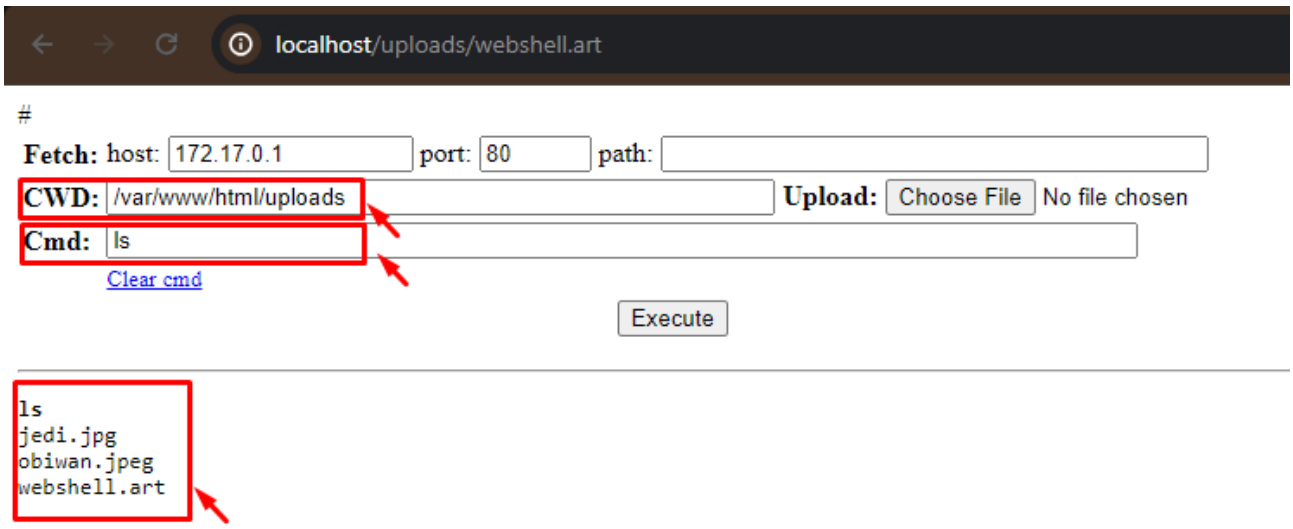


Figure 3.1.12 – Uploaded gallery files listed.

Now let's list all the files that exist in the root directory. As seen in Figure 3.1.13., there is a "flag.txt" which is most possibly to be the file that contains the hidden flag.

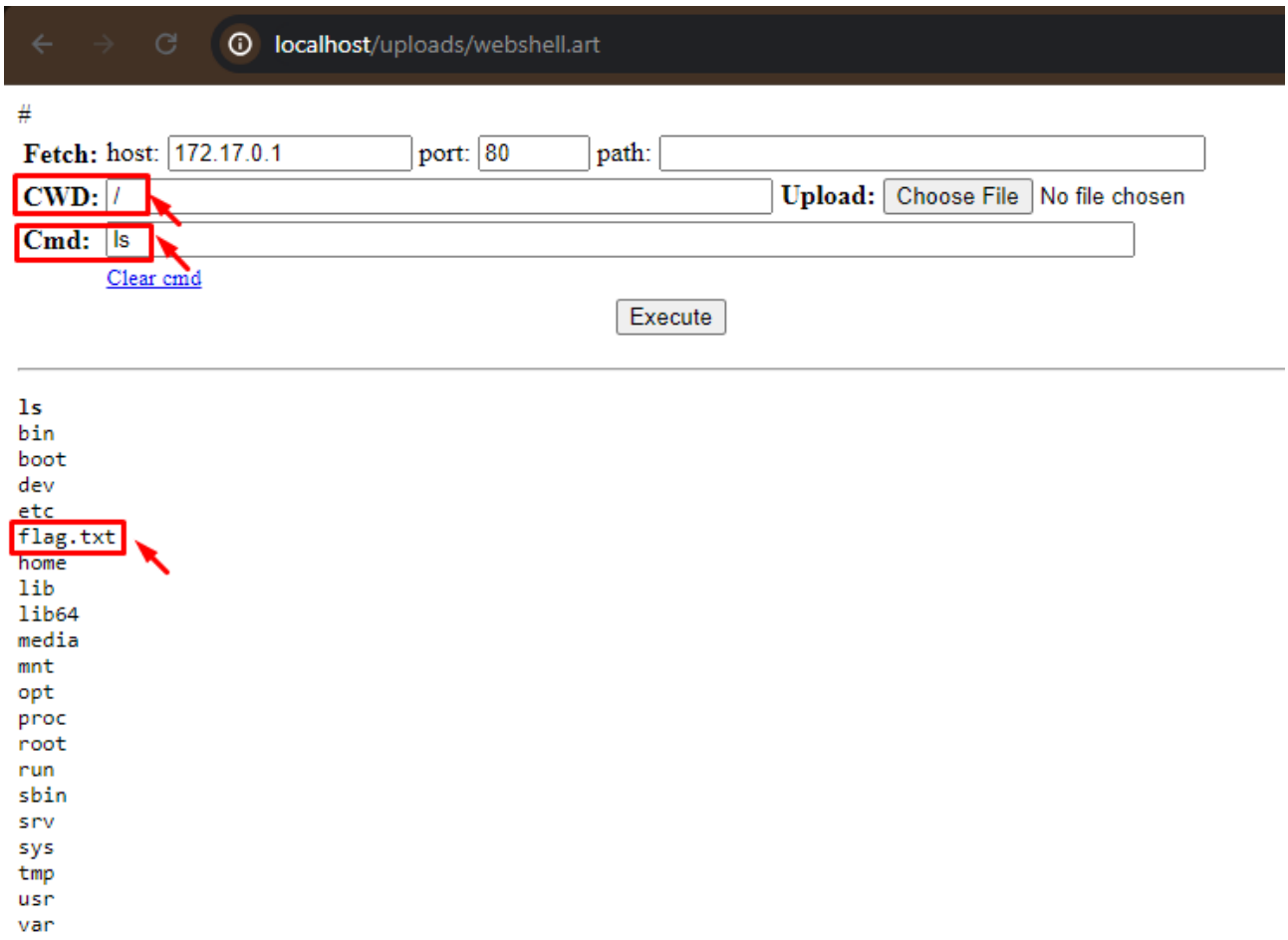


Figure 3.1.13 – Root files listed.

We execute the “cat /flag.txt” command to view the contents of the file “flag.txt” as depicted in Figure 3.1.14.

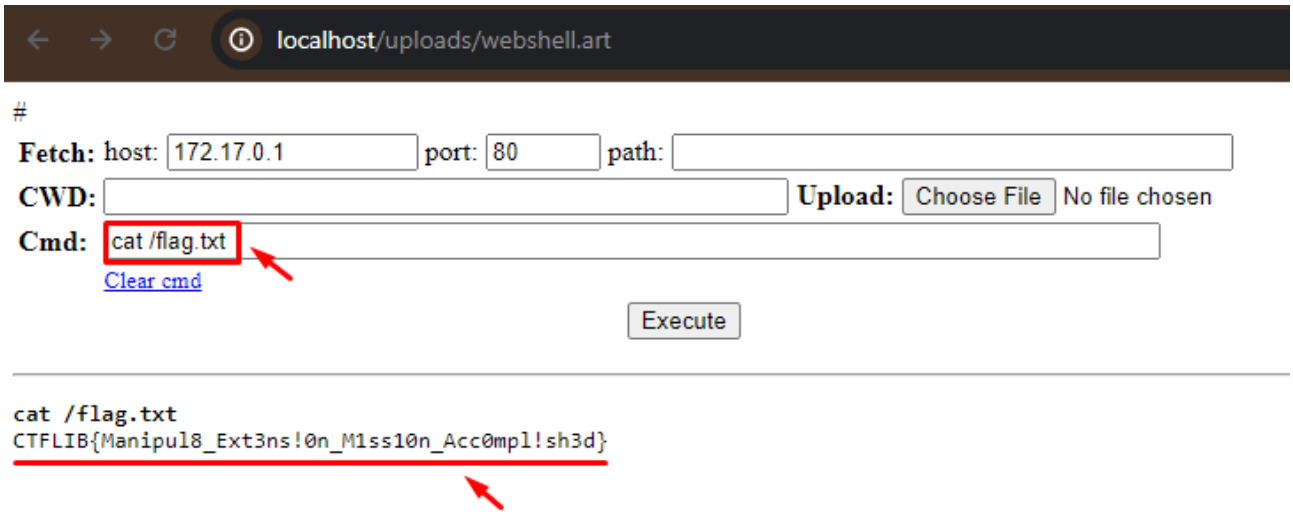


Figure 3.1.14 - Viewing flag.txt content.

ALRIGHT! We successfully retrieved the hidden FLAG.

CTFLIB{Manipul8_Ext3ns!0n_M1ss10n_Acc0mpl!sh3d}

3.1.5 Creation

This challenge as it is referred in chapter 3.1.1. Challenge Overview is a digital art gallery web application.

Here is the Dockerfile of the challenge as seen in Figure 3.1.15.

Dockerfile

```
FROM php:7.2.12-apache

# Copy application's source code to the image
COPY html/ /var/www/html/

# Create and set proper permissions for the uploads directory
RUN mkdir -p /var/www/html/uploads && chown -R www-data:www-data /var/www/html/uploads

# Copy the flag file into the root directory of the container
COPY flag.txt /
# Make the flag file readable to everyone.
RUN chmod a+r /flag.txt

# Expose port 80
EXPOSE 80
```

Figure 3.1.15 - Challenge Dockerfile.

This Dockerfile sets up a PHP and Apache web server environment using php:7.2.12-apache as the base image of the container. It creates a directory for saving the uploaded files and copies the application's source code and the hidden flag in the container. Then it makes the flag readable and configures the container to serve the web application on port 80.

index.php

```php
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="Explore and share digital art in our unique gallery.">
    <title>Digital Art Gallery</title>
    <style...>
</head>
<body>
<div class="header">
    <h1>Welcome to the Digital Art Gallery</h1>
    <p>Discover and share amazing digital artworks</p>
</div>

<div class="main-content">
    <h2>Explore the Beauty of Digital Art</h2>
    <p>Join our community of artists and art enthusiasts. View the gallery or share your own masterpieces.</p>
    <a href="upload_art.php" class="upload-btn">Upload Your Art</a>
</div>

<footer>
    <p>Digital Art Gallery © 2023. All Rights Reserved.</p>
</footer>
</body>
</html>
```

Figure 3.1.16 - Challenge home page / index.php file.

Figure 3.1.16. depicts the home page of the challenge (index.php) which is a simple front-end page that just redirects the participants to the file upload page.

This page is created in the upload_art.php file that also implements the file upload functionality.

upload_art.php

```php
<?php
// Define the extensions deny list
$ext_denylist = array(
    "php", "php2", "php3", "php4", "php5", "php6", "php7",
    "phps", "pht", "phtm", "phtml", "pgif", "shtml", "phar",
    "inc", "hphp", "ctp"
);

// Specify the maximum file size (in bytes)
$maxFileSize = 5000000; //5MB

// Initialize a variable to store messages
$message = "";
```

Figure 3.1.17 – First part of the PHP script / file extension denylist and file size limit setup. [7]

Figure 3.1.17. demonstrates the first part of the PHP script where a file extension denylist and a file size limit are set to prevent the upload of potentially executable or harmful files.

```php
// Check if the form was submitted using POST method
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    // Check if file was uploaded without errors
    if (isset($_FILES["fileToUpload"]) && $_FILES["fileToUpload"]["error"] == 0) {
        $filename = $_FILES["fileToUpload"]["name"];
        $filesize = $_FILES["fileToUpload"]["size"];

        // Check if the uploaded file exceeds the maximum size limit
        if ($filesize > $maxFileSize) {
            $message = "<div class='error-msg'>Error: File size is larger than the allowed limit.</div>";
        } else {
            // Extract file extension for further validation
            $ext = pathinfo($filename, flags: PATHINFO_EXTENSION);

            // Check if the file extension is in the deny list
            if (in_array($ext, $ext_denylist)) {
                $message = "<div class='error-msg'><h1>ERROR:</h1> This file type is not allowed.</div>";
            } else {
                // Construct the target filepath where the file will be moved
                $target_file = "/var/www/html/uploads/" . $filename;

                // Attempt to move the uploaded file to the target location
                $moved = move_uploaded_file($_FILES["fileToUpload"]["tmp_name"], $target_file);
                if ($moved) {
                    // Success: File is uploaded. Display success message with link to the file.
                    $message = "<div class='success-msg'><h1>SUCCESS:</h1> Your artwork has been uploaded. View it here: <a href='uploads/$filename'>uploads/$filename</a>.</div>";
                } else {
                    // Failure: File couldn't be moved. Show error message.
                    $message = "<div class='error-msg'><h1>ERROR:</h1> Sorry, there was an error uploading your file '$filename'.<br><br></div>";
                }
            }
        }
    } else {
        // No file selected: Inform the user to choose a file.
        $message = "<div class='error-msg'><h1>ERROR:</h1> No file was selected.</div>";
    }
}
?>
```

Figure 3.1.18 - Second part of the PHP script / Security checks for the uploaded files application.

As seen in Figure 3.1.18. this is the second part of the PHP script, where the appropriate security checks carried out for the uploaded file (file size, file extension). If the file passes these checks and considered safe, it is moved to a designated 'uploads' directory on the server and the appropriate success message is displayed, including a link to the uploaded file. Conversely, if the file does not pass these checks, the appropriate success fail is displayed.

3.2 V4ln Design

3.2.1 Challenge Overview

This “medium” level challenge presents Path Traversal and Remote command execution vulnerabilities within an Interior Design firm website, designed with a vulnerable Apache server configuration that allows attackers to navigate beyond intended boundaries or execute unauthorized commands on the server. The participants have to identify the vulnerabilities and craft malicious HTTP requests to exploit them.

3.2.2 Skills Acquired

This challenge will help the participants acquire several key skills.

- In-depth understanding of web server configurations
- Critical Thinking and Problem-Solving
- Awareness of secure code practices (e.g. input validation and sanitization, principle of least privilege, etc.)
- Prevent / Mitigate path traversal and remote command execution vulnerabilities
- Exploit path traversal and remote command execution vulnerabilities

3.2.3 Background Theory

The background theory for this challenge encompasses a fundamental understanding of web application security, especially focusing on Path Traversal and Remote Command Execution (RCE) vulnerabilities.

Path traversal, also known as directory traversal, involves exploiting a vulnerability that allows attackers to access files and directories that are stored outside the intended web server directory. By manipulating variables that reference files with dot-dot-slash (../) sequences and similar techniques, attackers can illegally navigate through the server's directory structure. This could potentially allow them to access sensitive files, such as configuration files, source code, or even execute server-side scripts that are not intended to be directly accessible. Figure 3.2.1. illustrates an attacker exploiting a path traversal vulnerability, reading the /etc/passwd file from the server’s root.

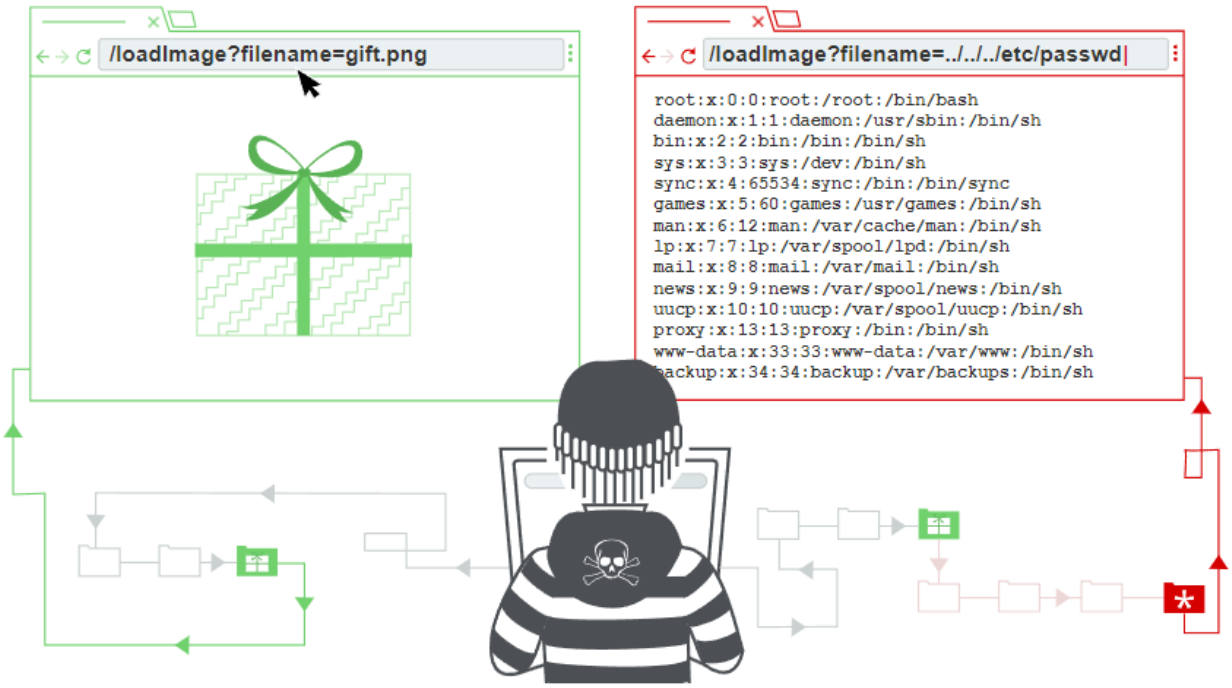


Figure 3.2.1 - Example of a Path Traversal vulnerability exploit. [8]

Remote Code Execution (RCE) is a critical vulnerability that allows an attacker to execute arbitrary code on an application's server. This type of exploit can lead to complete control over the affected system, allowing an attacker to manipulate server behavior, access confidential data, or disrupt service operations. RCE vulnerabilities often arise from improper input validation, allowing specially crafted inputs to manipulate the application or server into executing unintended commands. Figure 3.2.2. demonstrated an attacker exploiting a remote command execution vulnerability, executing remotely a malicious shell script.

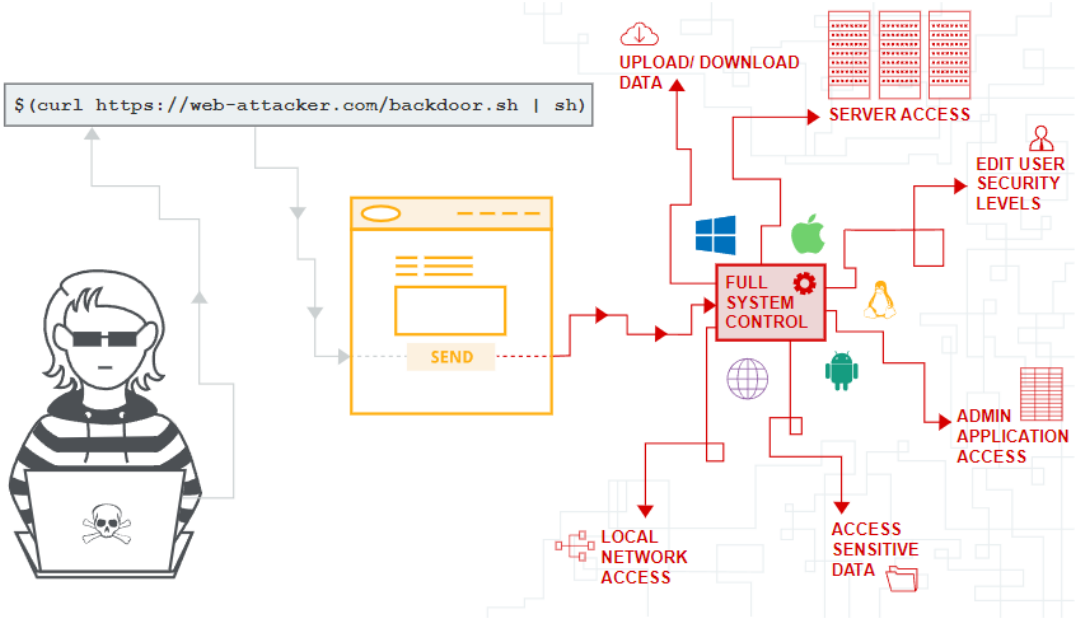


Figure 3.2.2 - Example of a Remote Command Execution vulnerability exploit. [9]

Here are some good strategies to reduce the risk and the potential impact of Path Traversal and Remote Command Execution (RCE) vulnerabilities within an application or a system.

- Validate user inputs before processing them, removing meta characters
- Use whitelists for user inputs
- Restrict file access operations to a secure, isolated section of the filesystem
- Update your web server and operating system to the latest versions available
- When making calls to the filesystem, never rely on user input for any part of the path
- Use security controls (firewalls, IDS, WAfs) for monitoring and detecting suspicious activities

3.2.4 Solution

First of all, we deploy the challenge from CTFLib.

Then we are redirected at http://localhost:8081/ which is the home page of the challenge as seen in Figure 4.1. This appears to be the webpage for the "Vuln" interior design firm.

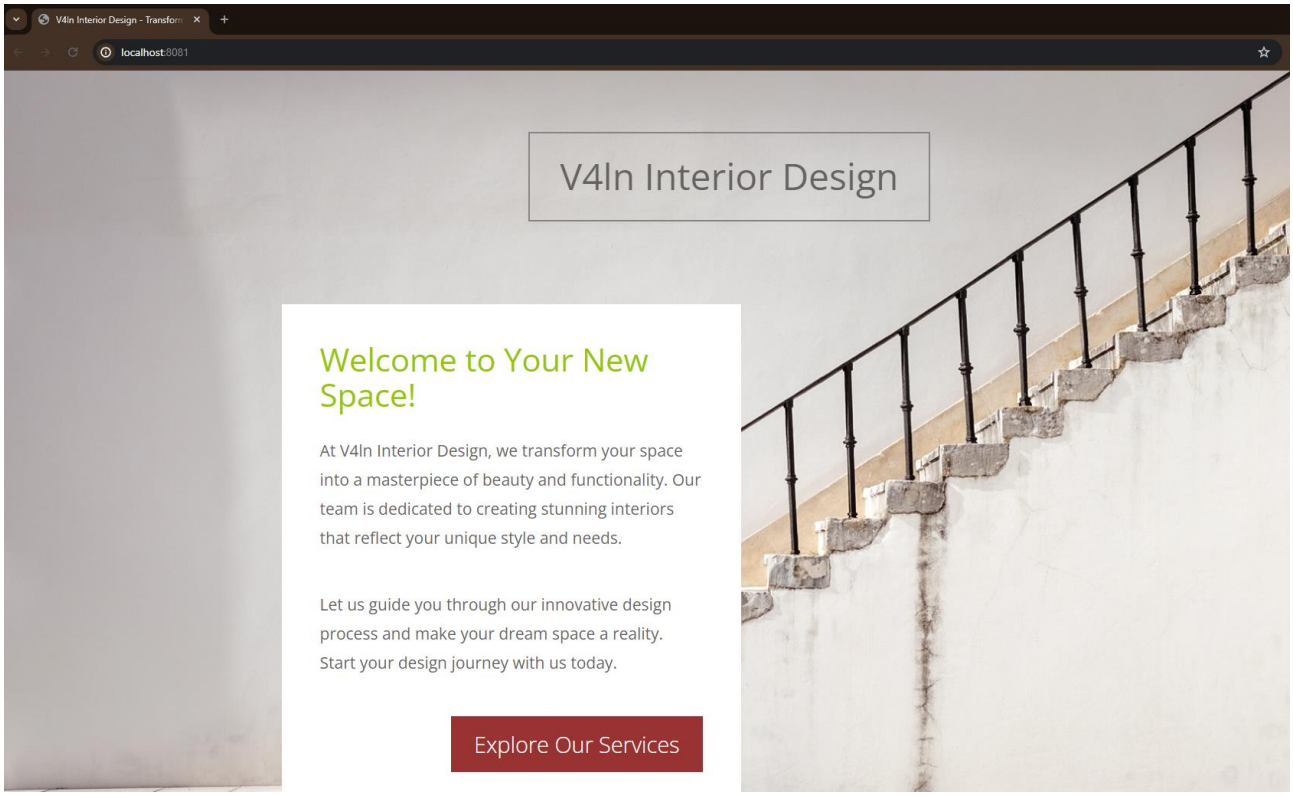


Figure 3.2.3 - Challenge's Home Page - Interior Design website.

At the first glance, the only thing that seems possible to apply some exploits on this website is the “Get in Touch” form at the bottom of the page.

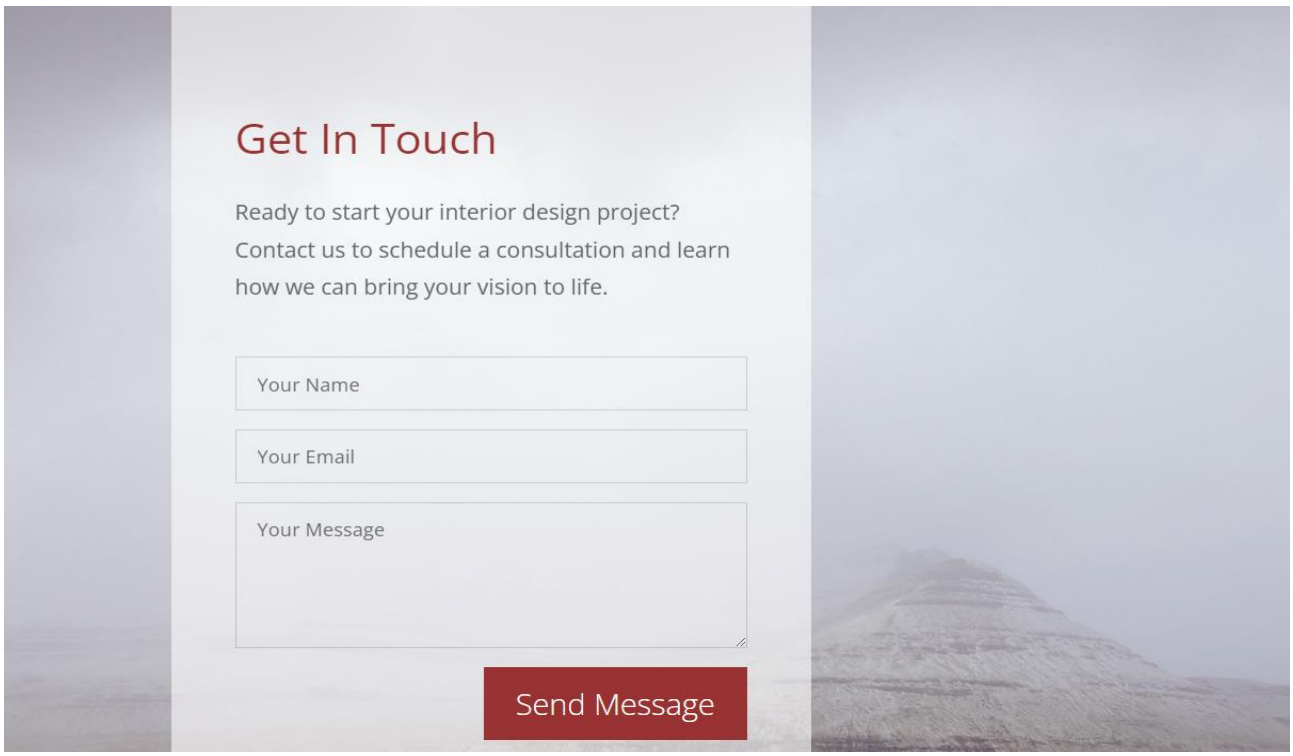


Figure 3.2.4 - "Get in Touch" form.

Challenge Description:

"Discover the elegance of V4ln Interior Design in this captivating challenge. Dive into the world of luxury and creativity, as you search for clues seamlessly woven into the fabric of this intricately designed website."

Since we can’t find any useful hints in the challenge description, we hop on inspecting the source code of the website (pressing “CTRL + U” on the page) as depicted in Figure 3.2.5.

There I realized that the contact form is not functional. We can cross validate it from two significant points. The form's action attribute points to "index.html", an HTML file, which cannot process or store form data in a database. HTML files are static and do not contain server-side processing capabilities.

```
<!-- contact form -->
<form action="index.html" method="post" class="tm-contact-form">
    <div class="form-group">
        <input type="text" id="contact_name" name="contact_name" class="form-control" placeholder="Your Name"  required/>
    </div>
    <div class="form-group">
        <input type="email" id="contact_email" name="contact_email" class="form-control" placeholder="Your Email"  required/>
    </div>
    <div class="form-group">
        <textarea id="contact_message" name="contact_message" class="form-control" rows="5" placeholder="Your Message" required></textarea>
    </div>
    <button type="submit" class="tm-btn">Send Message</button>
</form>
```

Figure 3.2.5 - Form's source code.

Likewise, there is no visible AJAX code provided that would handle the form submission asynchronously to send the data to a server or database, indicating the form's submission process does not interact with any backend system.

Every time I submitted the form, I was just redirected to the homepage without any message displayed on my screen. Now it makes sense why this was happening.

While checking the network traffic from the browser developer tools (by pressing the “F12” button on the browser), I distinguished a clue that could be proved crucial. The server used for this website was “Apache/2.4.49” as depicted in Figure 3.2.6.

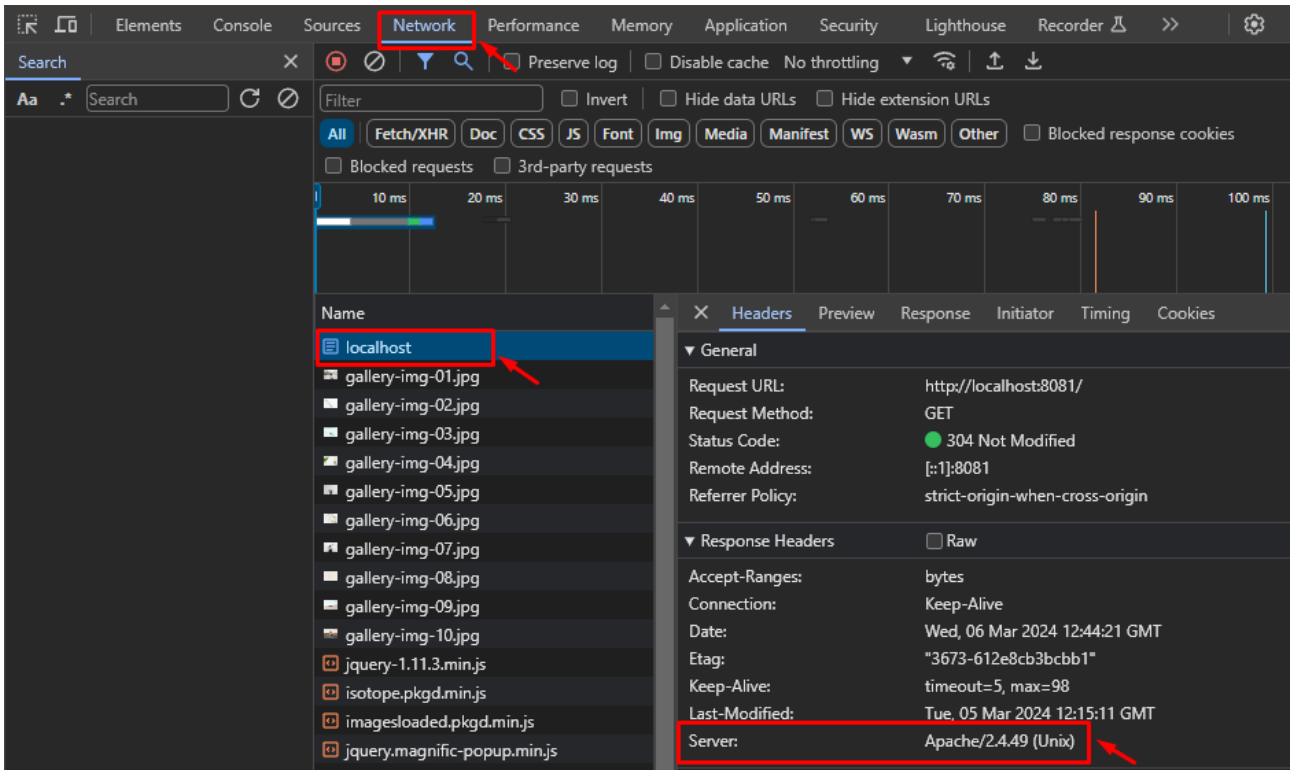


Figure 3.2.6 - Browser Developer tools / Server Used

So, let's do some googling to see if there are any possible vulnerabilities for this specific version of Apache.

Just typed “Apache/2.4.49” and as depicted in the figure, we understand that something goes wrong with this version.

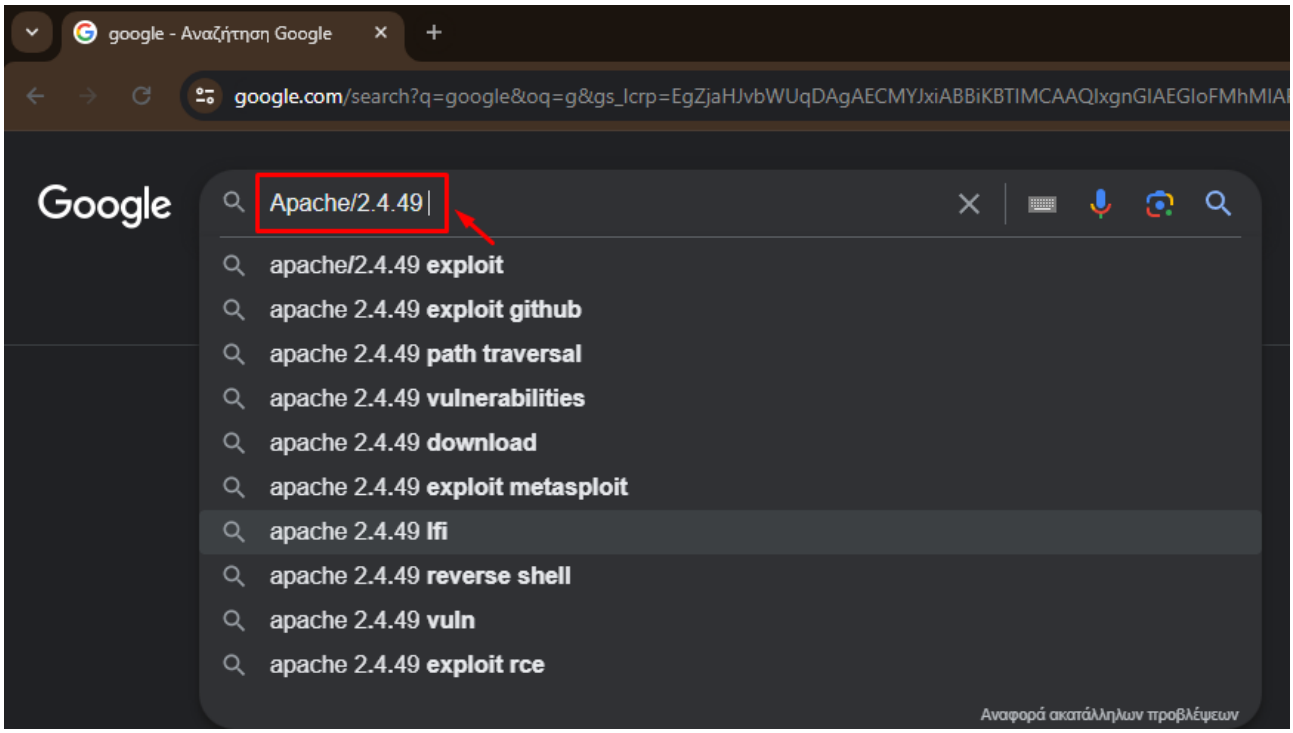


Figure 3.2.7 – Searching for Apache 2.4.49 vulnerabilities.

The first article I found from “ExlploitDB”, indicates that this version of Apache server is vulnerable to Path Traversal and Remote Code Execution (RCE) constituting “CVE-2021-41773”. (Exploit Database is a well-known website including Exploits, Shellcode, 0days etc).

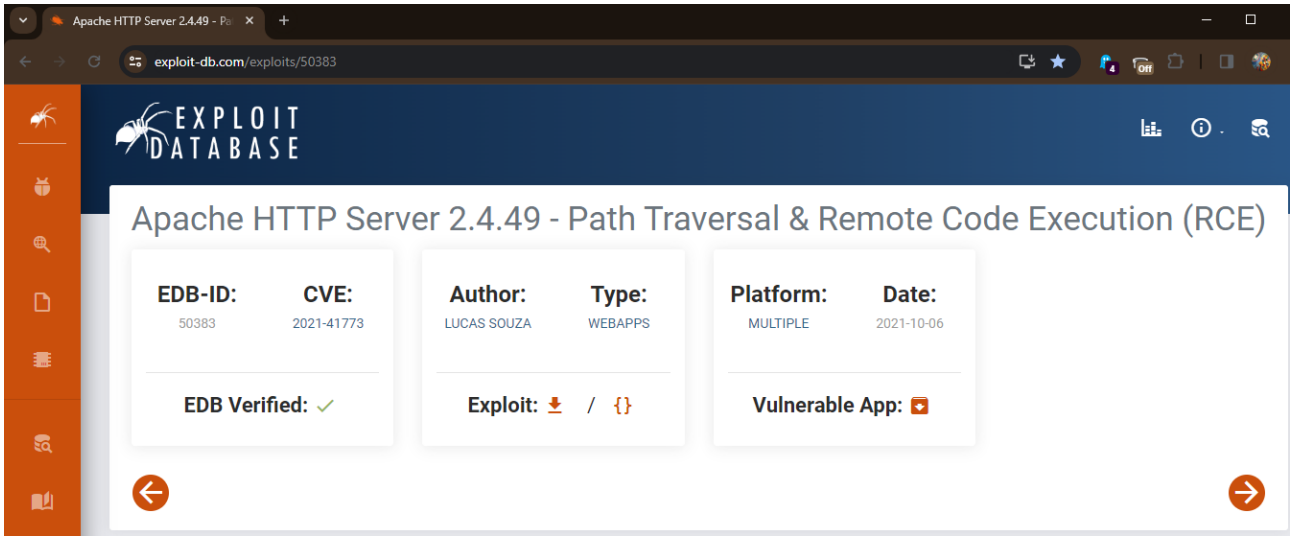


Figure 3.2.8 - ExploitDB Apache 2.4.49 vulnerabilities article. [10]

Upon further investigation, I found some very helpful articles and videos [11][12][13] that analyse extensively this CVE's vulnerability, explain everything that someone has to know about it and demonstrate how to exploit it.

In Figure 3.2.9. there is a snippet of the first article that explains briefly how this vulnerability arise.

Why is This Vulnerability Caused?

According to the Apache HTTP Server Project's advisory, CVE-2021-41773 vulnerability was discovered in a change to path normalization in Apache HTTP Server version 2.4.49. Because of this change, a path traversal attack could be used to map URLs to files not in the expected document root by sending specially crafted requests to the Apache webserver. These requests may succeed if files outside the document root are not protected by "require all denied."

Briefly, because of the change in path normalization mechanism of Apache HTTP Server 2.4.49, it does not properly neutralize sequences such as ".." that can resolve to a location outside of that directory.

Figure 3.2.9 - Snippet that explains how the vulnerability caused. [11]

Now let's apply this exploit in our situation. To do that I will use "Burp Suite Proxy" tool [17]. Utilizing this tool, I will capture the website's requests using "Burp's Browser".

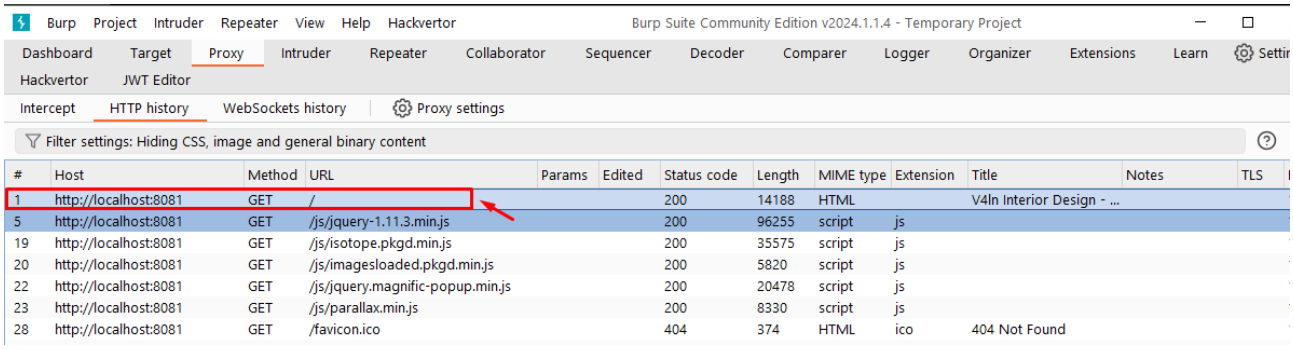


Figure 3.2.10 - Capturing website's traffic using BurpSuite Proxy.

Then I will modify one of them with “Burp’s Repeater” and resend the new malformed.

The payload I used on this request is the following and it consists of two different parts:

First part: “/cgi-bin/.%2e/.%2e/.%2e/.%2e/.%2e/bin/sh”

Second part: “echo;id”

So, lets break it down explaining what happens while executing this payload.

In the first part this URI (Uniform Resource Identifier) attempts to exploit the path traversal vulnerability.

/cgi-bin/: This is the directory on the server where CGI (Common Gateway Interface) scripts are typically stored and executed.

.%2e: This is URL encoding for . (dot), a directory level in Unix-like file systems. In this context, ”.%2e” represents a "dot" character, attempting to bypass security mechanisms by encoding.

The sequence /.%2e/.%2e/.%2e/.%2e/.%2e: Tries to traverse up five directory levels from the /cgi-bin/ directory. The goal is to reach the root directory of the filesystem.

/bin/sh: After reaching the filesystem's root, the request attempts to access the shell (sh) located typically in the /bin directory. This would theoretically allow the execution of shell commands.

In the second part, the payload contains the commands we want to execute on the server. The “echo;id” is a Unix command that prints the real and effective user and group IDs. Now that’s executed, it demonstrates that arbitrary commands can be run on the server, showing the user context under which the web server's process is running as seen in Figure 3.2.11.

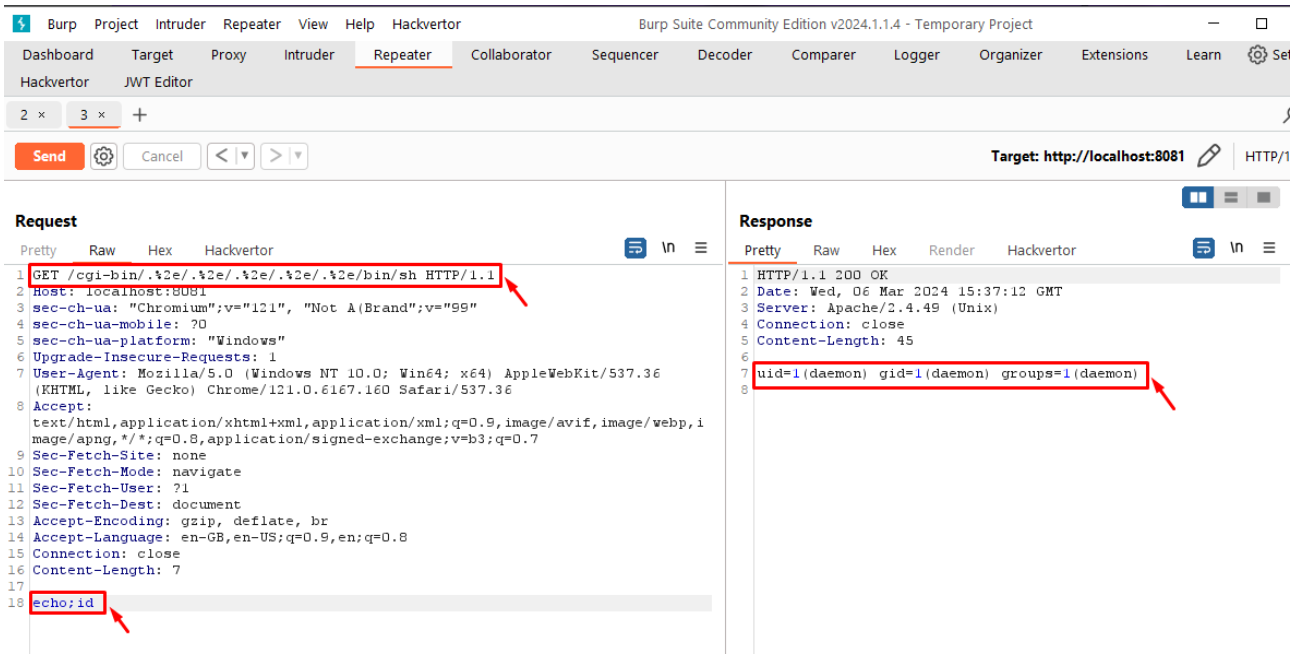


Figure 3.2.11 - Payload execution using Burp’s Repeater.

So, since we can execute arbitrary commands on the server, I did some testing trying to determine which commands could be proved useful.

As depicted in Figure 3.2.12. the command I executed is “echo;cd /var/www/html;ls”. This command lists the contents of the “/var/www/html” directory. Before ending up in this directory I was going into each directory and listing its contents separately so I see every possible file I could access.

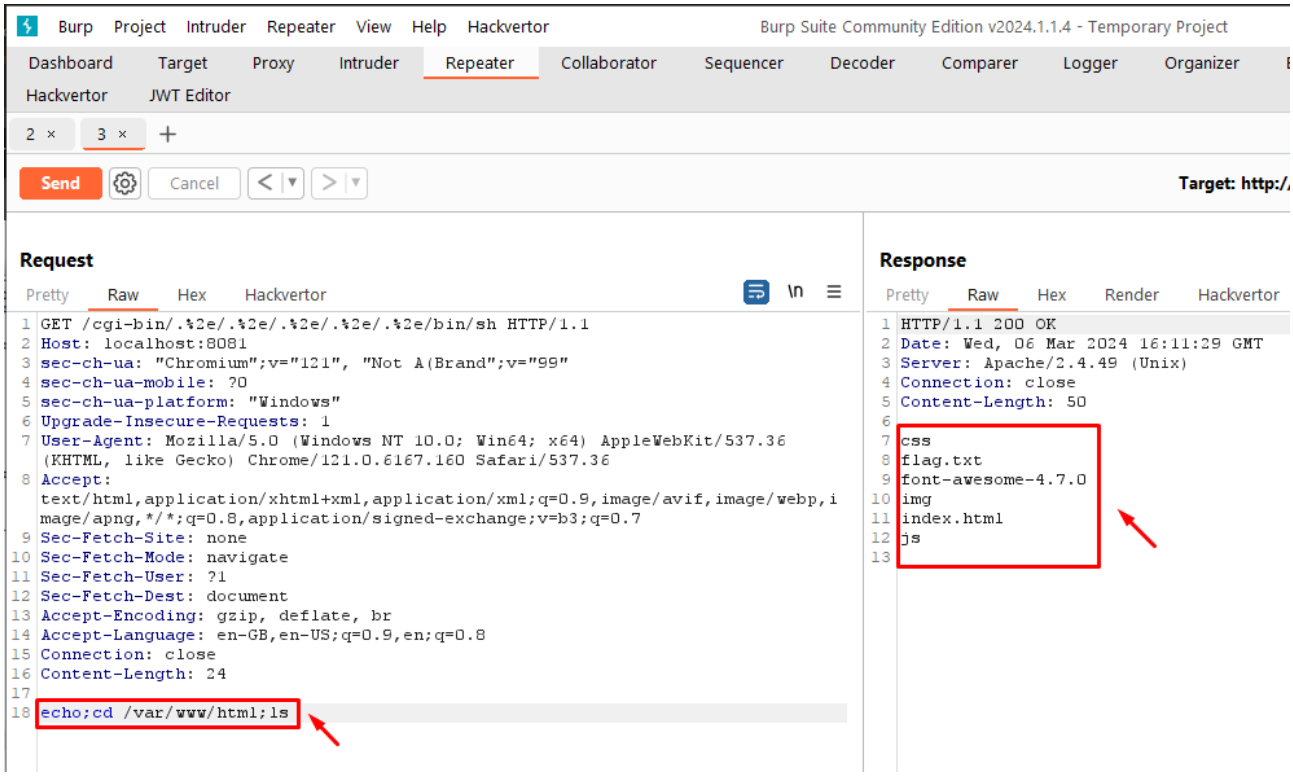


Figure 3.2.12 - Listing the contents of "/var/www/html" directory.

In the “/var/www/html” directory I noticed that there is a “flag.txt” file that’s most possible to include the hidden flag.

So let’s execute “echo;cd /var/www/html;cat flag.txt” command to view the contents of “flag.txt”.

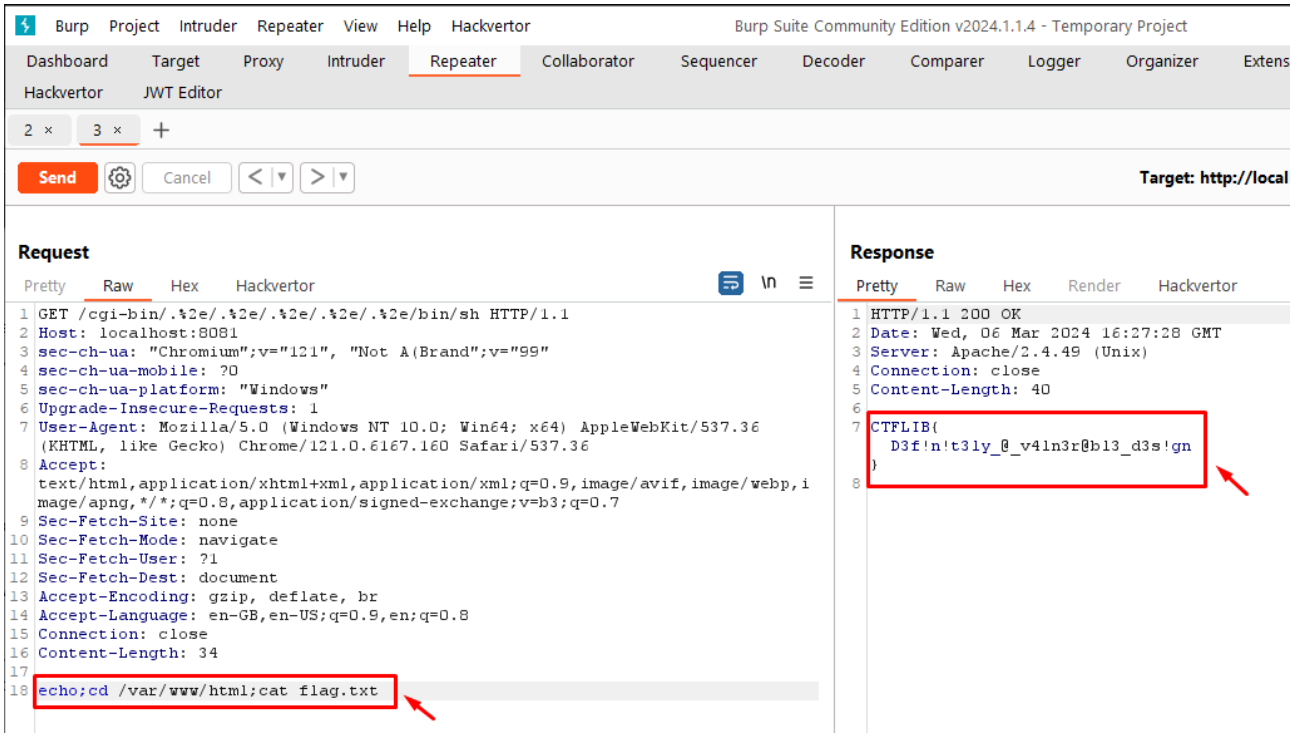


Figure 3.2.13 - Viewing the contents of flag.txt file / Hidden flag successfully retrieved.

NAILED IT, we found the hidden FLAG after successfully exploiting the path traversal and the remote command execution vulnerabilities.

CTFLIB{D3f!n!t3ly_@_v4ln3r@bl3_d3s!gn}

3.2.5 Creation

This challenge as it is referred in chapter 3.2.1. Challenge Overview is an interior design firm website.

Here is the Dockerfile of the challenge as seen in Figure 3.2.14.

Dockerfile

```
FROM httpd:2.4.49-buster

# Create /var/www/html and set permissions
RUN mkdir -p /var/www/html && chmod 755 /var/www/html

# Copy the application's source code into the container
COPY ./httpd.conf /usr/local/apache2/conf/httpd.conf
COPY ./html /var/www/html/
COPY ./flag.txt /var/www/html/flag.txt

# Copy the start script
COPY start-apache.sh /usr/local/bin/start-apache.sh

# Make sure the start script is executable
RUN chmod +x /usr/local/bin/start-apache.sh

# Use the custom script to start Apache
CMD ["/usr/local/bin/start-apache.sh"]
```

Figure 3.2.14 - Challenge Dockerfile.

This Dockerfile sets up an Apache 2.4.49 web server environment using httpd:2.4.49-buster as the base image of the container. It copies the application's source code and the hidden flag in the container. Then it configures a start script, makes it executable and utilizes it to start the application.

start-apache.sh

```
#!/bin/bash
# Ensure permissions are set correctly
chmod 755 /var/www/html

# Check the Apache configuration for syntax errors
httpd -t

# Start Apache in the foreground
httpd -DFOREGROUND
```

Figure 3.2.15 - Script for starting the Apache sever and run the application.

The front-end of the website is a free template that can be found online at https://templatemo.com/tm-504-page-one and it is adjusted properly to the challenges concept.

For the backend we create the httpd.conf file which is the Apache web server configuration file. The specific vulnerability that we want to implement particularly impacts environments where mod_cgi is enabled through the use of the ScriptAlias directive and the loading of the cgi_module. [14]

```
<IfModule !mpm_prefork_module>
    LoadModule cgid_module modules/mod_cgid.so
</IfModule>
<IfModule mpm_prefork_module>
    LoadModule cgi_module modules/mod_cgi.so
</IfModule>
```

Figure 3.2.16 - cgi-module loading.

```
# ScriptAlias: This controls which directories contain server scripts.
# ScriptAliases are essentially the same as Aliases, except that
# documents in the target directory are treated as applications and
# run by the server when requested rather than as documents sent to the
# client.  The same rules about trailing "/" apply to ScriptAlias
# directives as to Alias.
#
ScriptAlias /cgi-bin/ "/usr/local/apache2/cgi-bin/"
```

Figure 3.2.17 - ScriptAlias directive setup.

The Require all granted is also a key point in this configuration file for the demonstration of the vulnerability because it enables the access to the entire filesystem by default without any restrictions.

```
<Directory "/usr/local/apache2/cgi-bin">
    AllowOverride None
    Options None
    Require all granted
</Directory>
```

Figure 3.2.18 – Improperly setup restrictions with Require all granted.

3.3 Sci-Fi NameGen

3.3.1 Challenge Overview

This “medium” level challenge demonstrates an XML external entity injection (XXE) vulnerability within a web application that is designed to generate Sci-Fi character names. The participants must take advantage and manipulate the vulnerable input parameter that is presented on the URL of the website, so they perform an SSRF attack accessing the page from the web server.

3.3.2 Skills Acquired

This challenge will help the participants acquire several key skills.

- In-depth understanding of XML external entity injection (XXE) vulnerabilities
- Identify and manipulate encoded data
- Critical Thinking and Problem-Solving
- Awareness of secure code practices (e.g. implement proper access controls, secure XML data processing, etc.)
- Prevent / Mitigate XML external entity injection (XXE) vulnerabilities
- Exploit XML external entity injection (XXE) vulnerabilities

3.3.3 Background Theory

The background theory for this challenge encompasses a fundamental understanding of web application security, especially focusing on XML external entity injection (XXE) and Server-Side Request Forgery (SSRF) vulnerabilities.

XML external entity injection (XXE) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It frequently allows an attacker to communicate with any external or back-end systems that the application itself may access, as well as to see files on the server filesystem. In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by leveraging the XXE vulnerability to perform server-side request forgery (SSRF) attacks. Figure 3.3.1. illustrates an attacker exploiting an XXE injection vulnerability, reading the /etc/passwd file from the server's root.

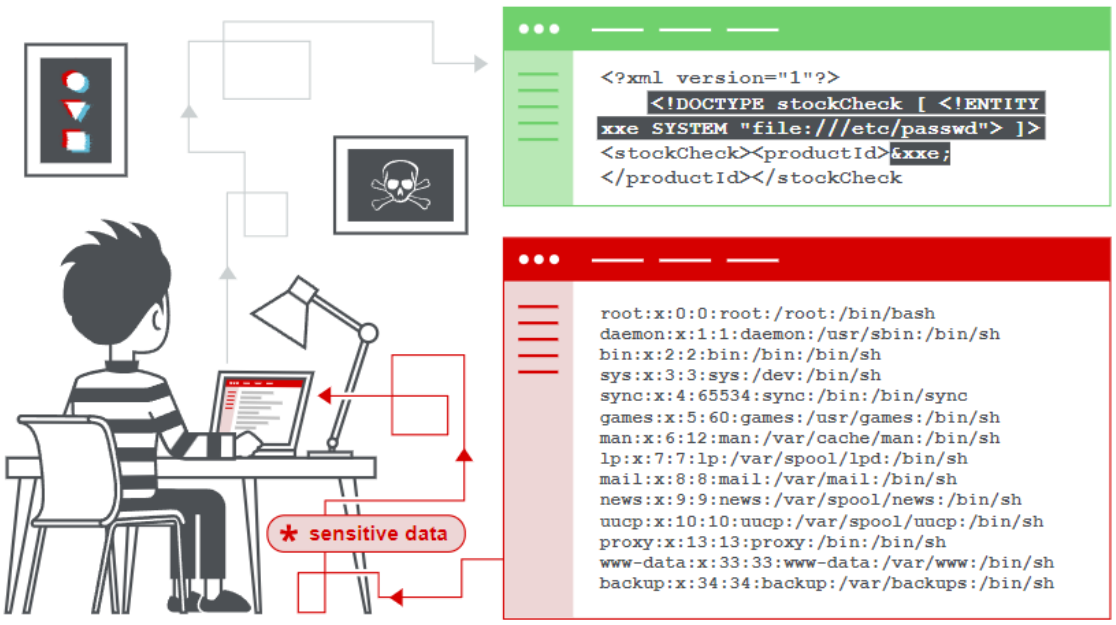


Figure 3.3.1 - Example of an XXE injection vulnerability exploit. [15]

Server-side request forgery (SSRF) is a web security vulnerability that allows an attacker to cause the server-side application to make requests to an unintended location that potentially leading to information disclosure, internal network enumeration, or external service interaction on behalf of the server. Figure 3.3.2. depicts an attacker performing an SSRF attack.

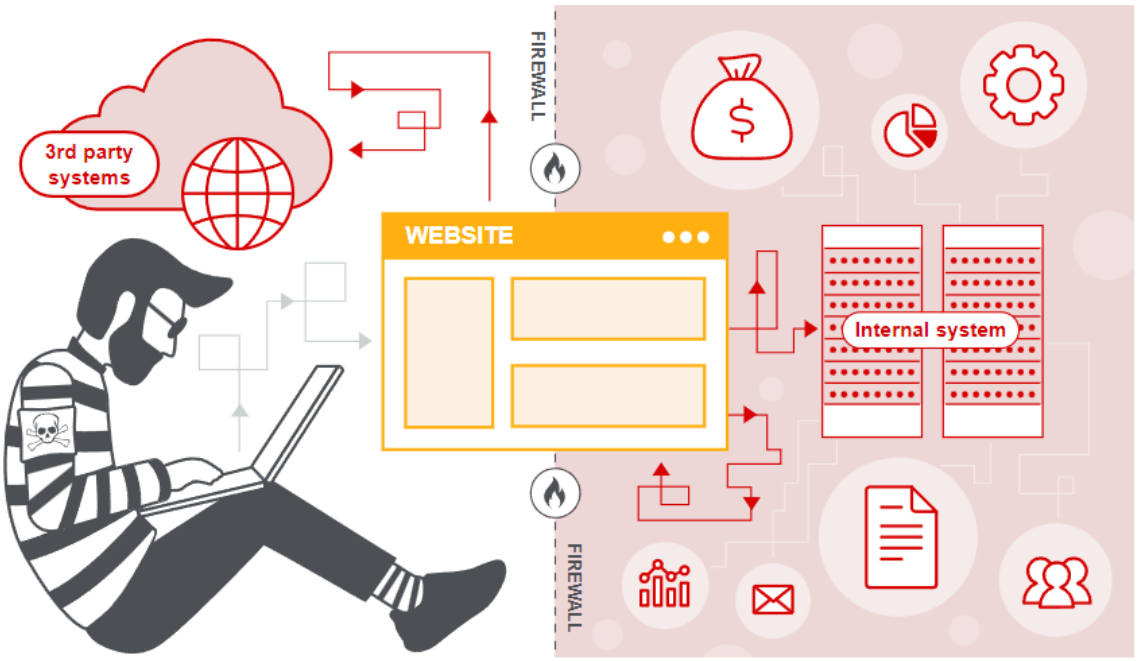


Figure 3.3.2 - Example of an SSRF attack. [16]

Here are some good principles to mitigate the XML external entity injection (XXE) vulnerabilities.

- Validate and sanitize user inputs before processing them
- Use whitelists for XML data, accepting only well-formed documents are processed
- Disable External Entity processing
- Use safe XML parsers
- Implement Proper Access Controls

3.3.4 Solution

First of all, we deploy the challenge from CTFLib.

Then we are redirected at http://localhost:8081/ which is the home page of the challenge as seen in Figure 3.3.3. To generate our character name, we need to fill up our first and last name and to choose one of the given character roles.

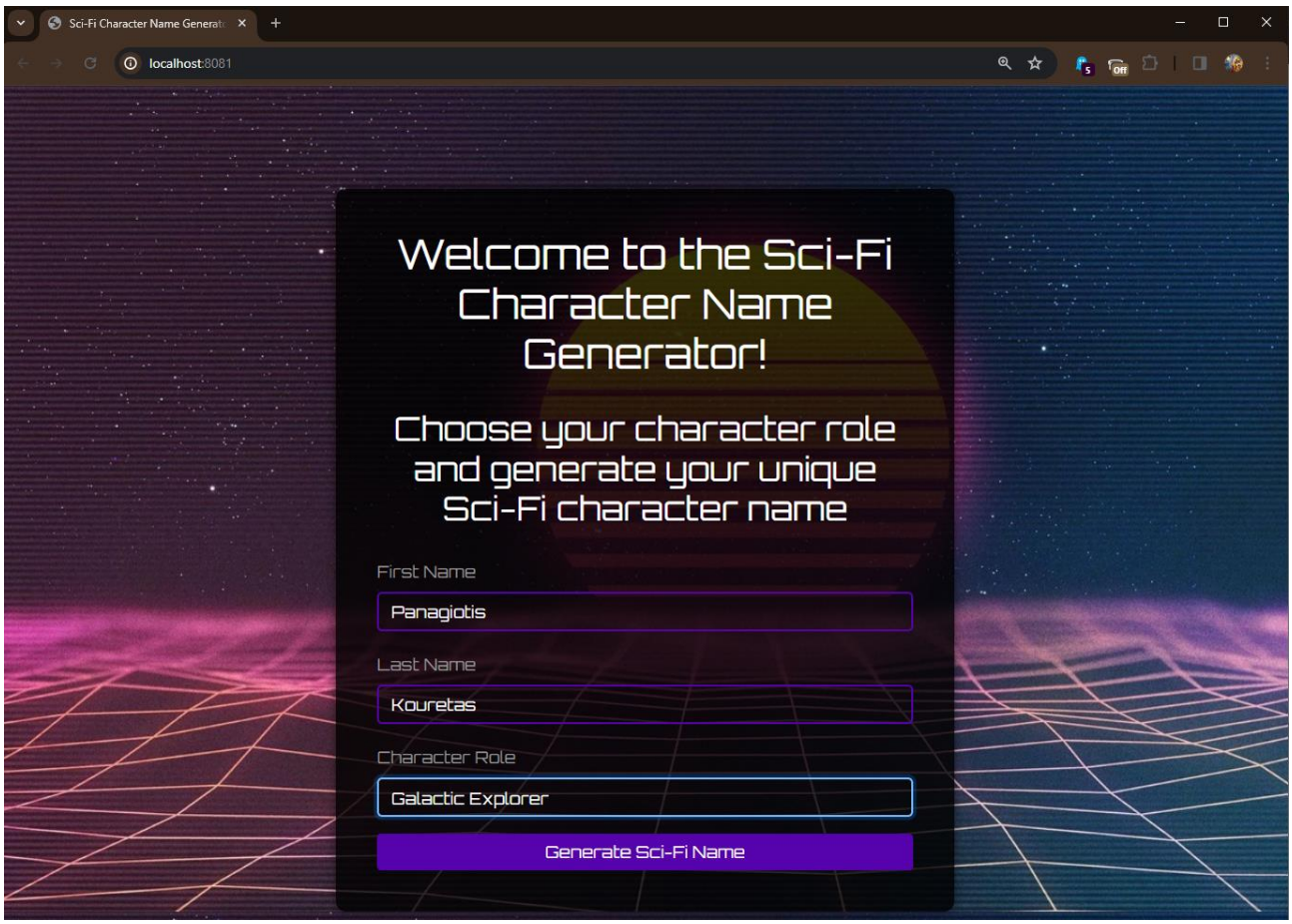


Figure 3.3.3 - Challenge's Home Page / Sci-Fi name generator page.

Then if we click the “Generate Sci-Fi Name” button, we are moved to another page where we can see our generated Sci-Fi character’s name at

http://localhost:8081/generate.php?input=PD94bWwgdmVyc2lvbj0nMS4wJyBlbmNvZGluZz0nVVRGLTgnPz48aW5wdXQ%2BPGZpcnN0TmFtZT5QYW5hZ2lvdGlzPC9maXJzdE5hbWU%2BPGxhc3ROYW1lPktvdXJldGFzPC9sYXN0TmFtZT48cm9sZT5HYWxhY3RpYyBFeHBsb3Jlcjwvcm9sZT48L2lucHV0Pg%3D%3D as demonstrated in Figure 3.3.4.

Figure 3.3.4 - Page for displaying the generated name.

The first thing we notice is that the URL of the challenge contains an input parameter. This parameter is an encoded value and most probably base64 encoded. Let's decode this value using Hackvertor which is one of the Burp Suite Proxy tool's [17] extensions. The decoded value is the following as depicted in the figure below:

<?xml version='1.0'
encoding='UTF8'?><input><firstName>Panagiotis</firstName><lastName>Kouretas</lastName><role>Galactic Explorer</role></input>

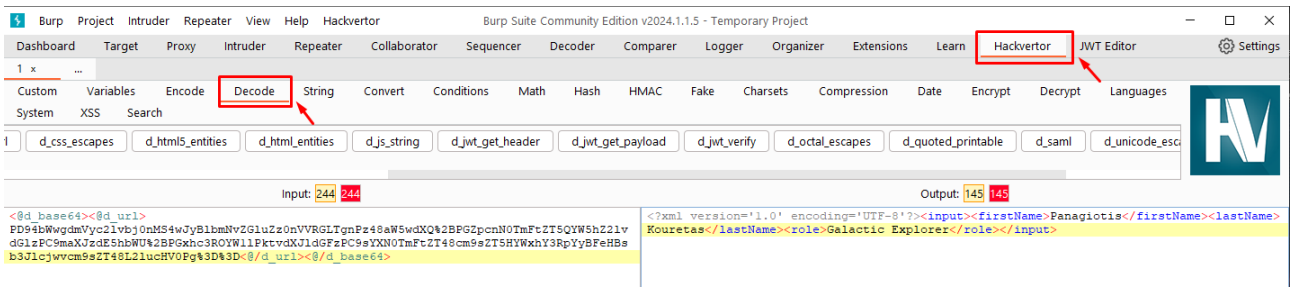


Figure 3.3.5 - Base64 + URL decoding value using Burp's Hackvertor.

If we try to modify, decode and send back the updated XML data in the input we can see that it's done successfully as depicted in Figure 3.3.6. So, it seems we have control over the XML data.

Figure 3.3.6 - Successfully modifying XML data in the input parameter.

Since we have XML data, let's try an XXE (XML external entity injection) attack. Upon further investigation, I found the following payload from "PayloadAllTheThings" [18] which is an online GitHub repository.

<?xml version='1.0' encoding='UTF-8'?><!DOCTYPE input [<!ENTITY xxe SYSTEM 'file:///etc/passwd'>]><input><firstName>&xxe;</firstName><lastName>Kouretas</lastName></input>

This payload attempts to read the contents of the "/etc/passwd" file on a Unix-like system, which is a common target for attackers as it contains a list of users on the system. Let's Base64+Url encode this and pass it in as our input in the URL.

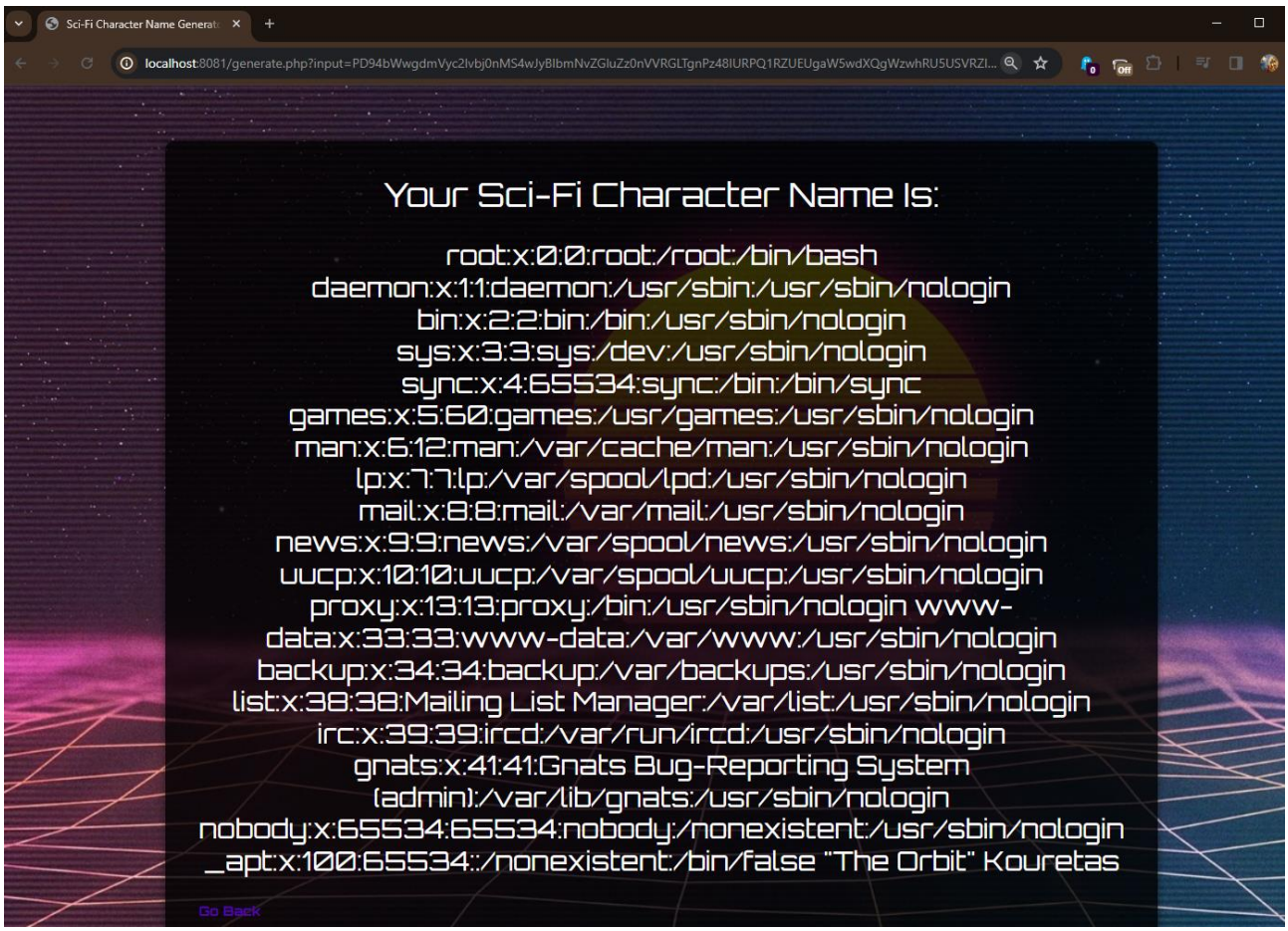


Figure 3.3.7 - XXE payload successfully returning "/etc/passwd" file.

It looks like we have a Local File Inclusion. Although there doesn't seem to be anything interesting in that file, so kept looking around the page searching for something that could help us take it one step further.

Inspecting the source code of the page, I found a comment that was indicating that there is a special functionality by accessing the website from a local address.

Comment: "<!-- Special sci-fi Character Name functionality: To test the special sci-fi title functionality, make sure you're accessing this page from the web server. -->".

Based on this let's send off the following updated payload that accesses the website from the localhost and see what will happen.

<?xml version='1.0' encoding='UTF-8'?><!DOCTYPE input [<!ENTITY xxe SYSTEM 'http://localhost/generate.php'>]><input><firstName>&xxe;</firstName><lastName>Kouretas</lastName></input>

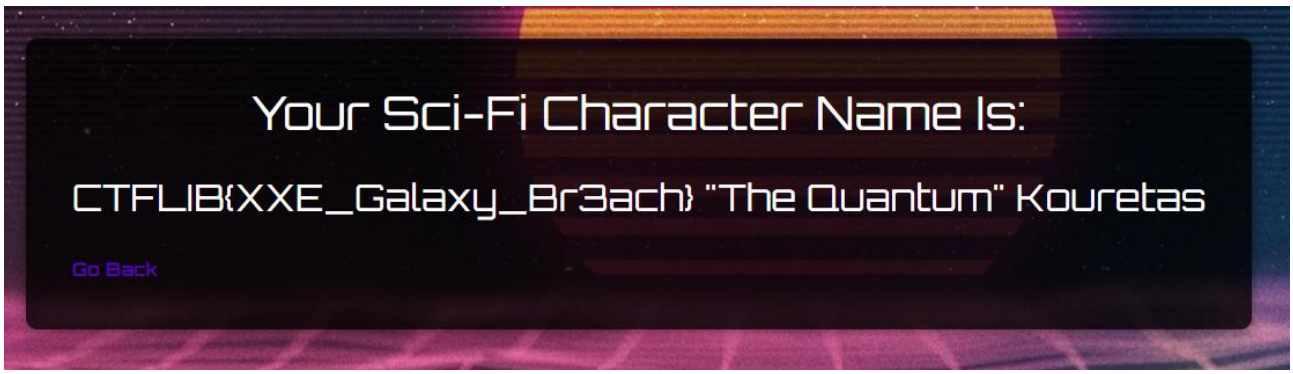


Figure 3.3.8 - Hidden flag successfully retrieved.

YEAH, it seems that the special sci-fi title functionality is returning the hidden FLAG.

CTFLIB{XXE_Galaxy_Br3ach}

3.3.5 Creation

Here is the Dockerfile of the application as seen in Figure 3.3.9.

Dockerfile

```
FROM php:7.0-apache

# Copy your application source code to the image
COPY html/ /var/www/html/

# Copy the flag into the container
COPY flag.txt /var/www/

# Modify permissions to make the flag file readable by the web server
RUN chmod a+r /var/www/flag.txt

# Expose port 80
EXPOSE 80
```

Figure 3.3.9 - Challenge Dockerfile.

This Dockerfile sets up php:7.0-apache as the base image of the container which includes Apache web server and PHP in our web application. It copies the application's source code and the hidden flag in the image. Finally, is exposes the port of the application.

```
<script>
    // Add an event listener to the button click
    document.getElementById("button").onclick = function(event) {
        // Prevent the default form submission behavior
        event.preventDefault();

        // Retrieve user inputs from the form
        var firstName = document.getElementById("firstName").value;
        var lastName = document.getElementById("lastName").value;
        var characterRole = document.getElementById("characterRole").value;

        // Validate inputs: ensure both first name and last name are provided
        if (!firstName || !lastName) {
            alert("Please fill in both your first name and last name.");
            return;
        }

        // Encode the input data into base64 format for URL passing
        var input = btoa("<?xml version='1.0' encoding='UTF-8'?><input><firstName>" + firstName + "</firstName><lastName>
        // Redirect to the generate.php page with the encoded input as a GET parameter
        window.location.href = "/generate.php?input=" + encodeURIComponent(input);
    };
</script>
```

Figure 3.3.10 - index.html page JS script.

Then we create the home page of our challenge (index.html), which is a simple HTML front-end page including a JS script. This script ensures that the inputs from the form are fulfilled and valid. If they are, an XML string that is storing the first name, last name, and character role gets base64 encoded. Finally, it redirects users to the page for generating the Sci-Fi character name, URL encodeds the data and passes them as a query parameter named input on the URL.

generate.php

```php
<?php
// Define an array of IP addresses considered safe (localhost)
$whitelist = array(
    '127.0.0.1',
    '::1'
);

// if this page is accessed from the web server, a special sci-fi title is returned
if(in_array($_SERVER['REMOTE_ADDR'], $whitelist)){
    // Output the contents of the flag.txt file
    echo file_get_contents( filename: '/var/www/flag.txt');
    return;
}
// Check if the 'input' GET parameter is provided with the request
if (empty($_GET["input"])) {
    echo "Please include the 'input' GET parameter with your request.";
    return;
}

// Decode the base64-encoded XML input from the GET parameter
$xmlData = base64_decode($_GET["input"]);
// Load the XML and handle errors
$xml = simplexml_load_string($xmlData, class_name: null, options: LIBXML_NOENT) or die("Error parsing XML: ".\n".$xmlData);
// Extract elements from the parsed XML
$firstName = $xml->firstName;
$lastName = $xml->lastName;
$role = $xml->role;

// Generate a random sci-fi character name from a predefined list
$titles = array("Starlord", "Galactic", "Space", "Quantum", "Nebula", "Vortex", "Astro", "Cosmic", "Orbit", "Stellar");
$title = $titles[array_rand($titles)];
// Concatenate the name and title to generate the full sci-fi name
$generatedName = $firstName.' "The '.$title.'" '.$lastName;
?>
```

Figure 3.3.11 - generate.php file for generating the Sci-Fi character name.

Figure 3.3.11. demonstrates the main backend file of the challenge. If a request comes from a whitelisted IP range (local server), then the hidden flag is returned. Otherwise, base64 encoded XML data are processed to generate a sci-fi character name. The generated name is based on provided first name, last name, and a randomly selected title from a predefined list. This file also including and the front-end part of the page with simple HTML.

4. Hard Challenges

This chapter explores the basic principles underlying the hard difficulty level challenges. It provides an in depth look at the necessary theoretical background, demonstrates the methodologies the participants need to follow to solve each challenge and showcases its creation process.

There are two hard challenges that demonstrate concepts such as the following:

- Json Web Token (JWTs) manipulation
- Server-Side Template Injection (SSTI)
- HTTP Request Smuggling
- Poor cryptographic algorithms implementation
- Unsecure proxy server configuration

4.1 Virtual Invitation Creator

4.1.1 Challenge Overview

This “hard” level challenge presents two different vulnerabilities within a web application that is designed to create virtual invitations. The first part of the challenge consists of manipulating an improperly implemented JSON Web Token (JWT) to gain unauthorized access to the create virtual invitations page, since this service is only for subscribed users and the subscribe button’s functionality is currently under maintenance. For the second part of the challenge the participants must exploit a Server-Side Template Injection (SSTI) vulnerability to execute arbitrary code in the backend server and retrieve sensitive information including the hidden flag.

4.1.2 Skills Acquired

This challenge will help the participants acquire several key skills.

- In-depth understanding of JSON web tokens mechanisms and Server-Side Template injection vulnerabilities
- Cryptographic security knowledge
- Critical Thinking and Problem-Solving
- Awareness of secure code practices (e.g. proper validation of signatures, sanitize user inputs , etc.)
- Prevent / Mitigate JWT and SSTI vulnerabilities
- Exploit JWT and SSTI vulnerabilities

4.1.3 Background Theory

The background theory for this challenge encompasses a fundamental understanding of web application security, especially focusing on JSON web tokens mechanisms and Server-Side Template injection vulnerabilities.

JSON Web Tokens (JWT) are essentially cryptographically signed JSON data that are sent between two systems, ensuring data integrity through digital signatures. They are frequently used for web authentication and authorization, encapsulating user details and permissions. A JWT consists of a header, a payload, and a signature part that are separated with dots. However, JWTs considered safe, improper implementations like poor signature verification or secret keys leakage can lead to critical security flaws. Figure 1.1 depicts an example of an attacker exploiting a vulnerable unsigned JWT, gaining unauthorized access.

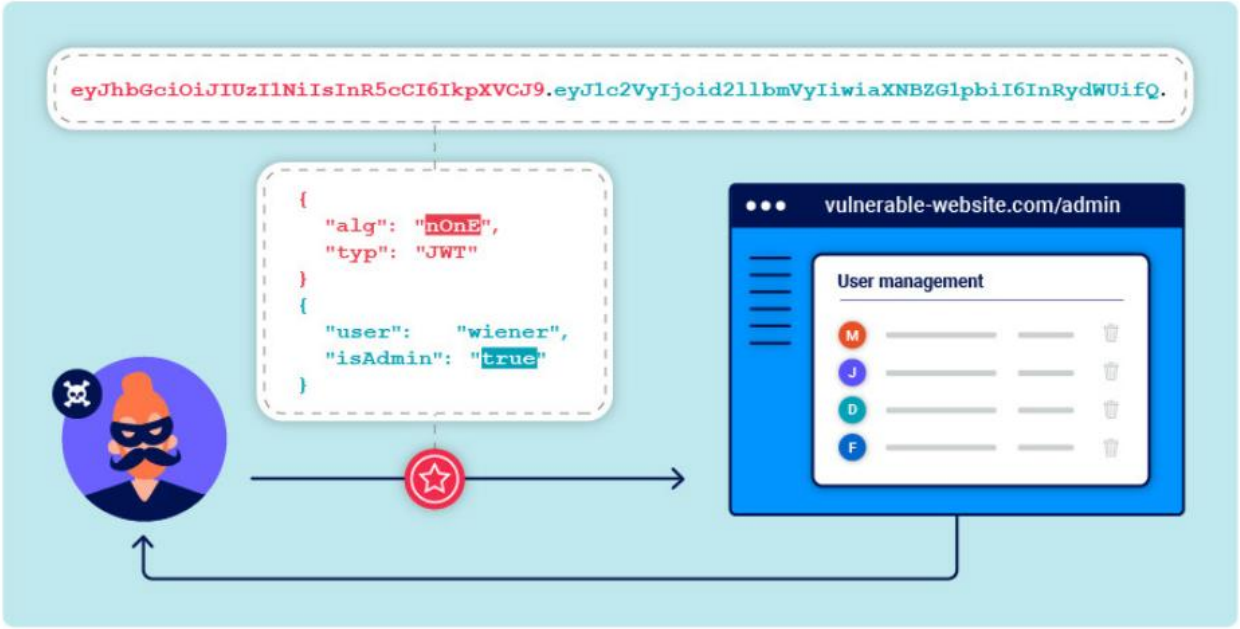


Figure 4.1.1 - Example of a JWT vulnerability exploit. [19]

Here are some good security practices to mitigate JWT vulnerabilities or reduce their potential impact.

- Use an up-to-date library for handling JWTs
- Perform robust signature verification
- Implement Proper Key Management
- Use Short Expiration Times
- Sanitize Input to Prevent Injection Attacks

Template engines are designed to generate web pages by using both static templates and dynamic data. Server-Side Template Injection (SSTI) is a vulnerability that allows an attacker to inject malicious code into a template that is executed in the backend server to gain unauthorized access or leak sensitive data. Figure 1.2. illustrates an example of some SSTI payloads that are used to identify which template engine is used. The payloads for exploiting SSTI vulnerabilities defer depending on which template engine is used.

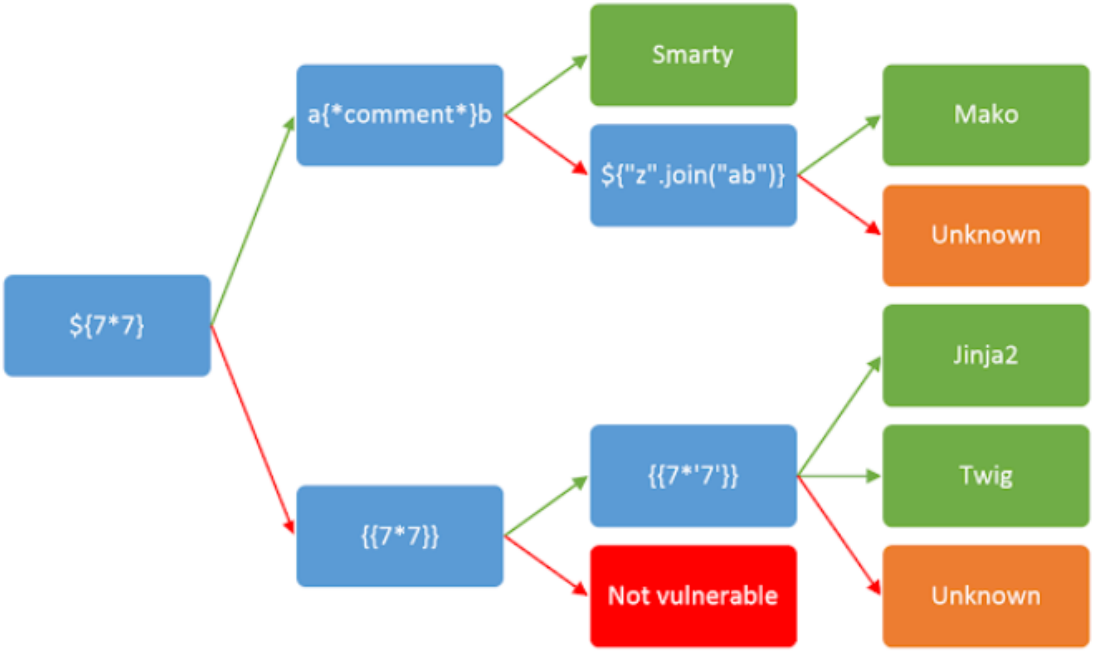


Figure 4.1.2 - Example of payloads to identify which template engine is used. [20]

SSTI vulnerabilities occur when template engines take untrusted user input and processing it. Here are some good security principles to prevent these vulnerabilities.

- Validate and Sanitize Input
- Use logic-less template engines
- Employ Sandboxing
- Do not allow users to submit or modify new templates if not necessary\
- Use context-aware escaping

4.1.4 Solution

First of all, we deploy the challenge from CTFLib.

Then we are redirected at http://localhost:4242/ which is the home page of the challenge as seen in Figure 4.1.3.

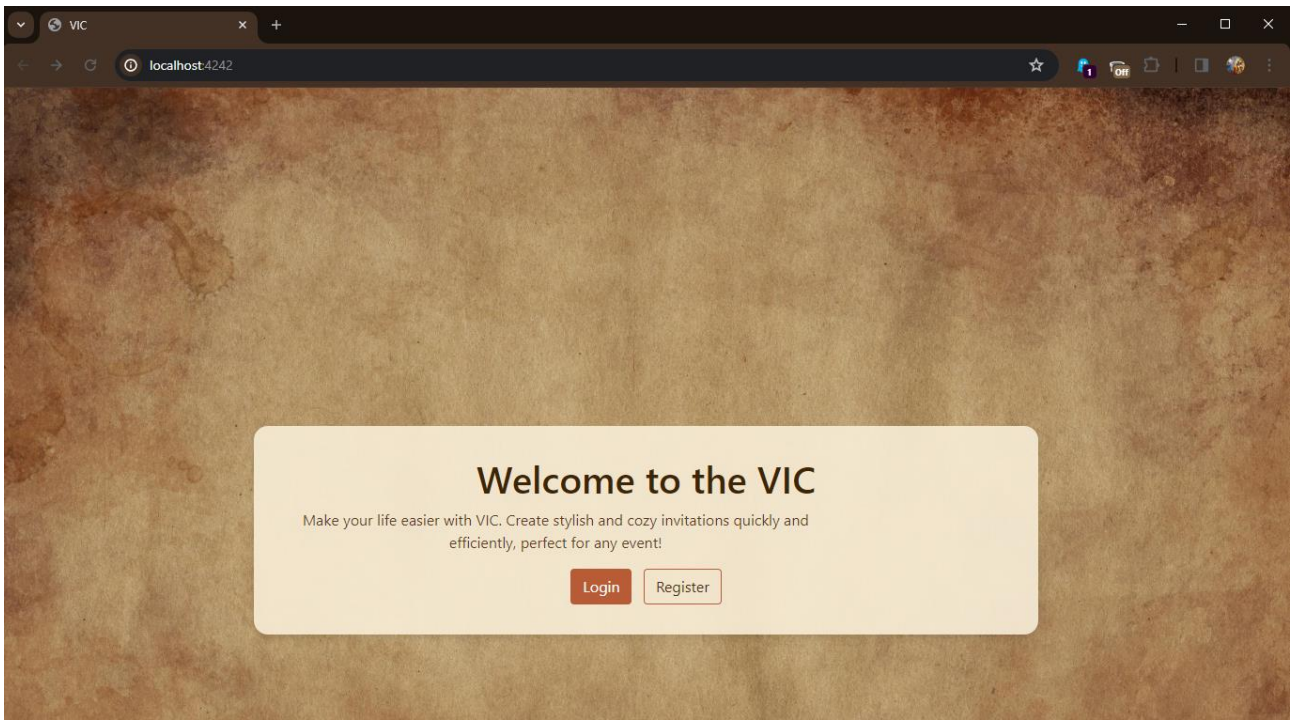


Figure 4.1.3 - Challenge's Home Page / Welcome page.

In this page we have two choices, to login or to register if we already haven't an account. Even though I'm not signed up and I don't have an account yet, I will first try using some common credentials (usename:admin, password:admin etc.) to bypass the login page found at http://localhost:4242/login.

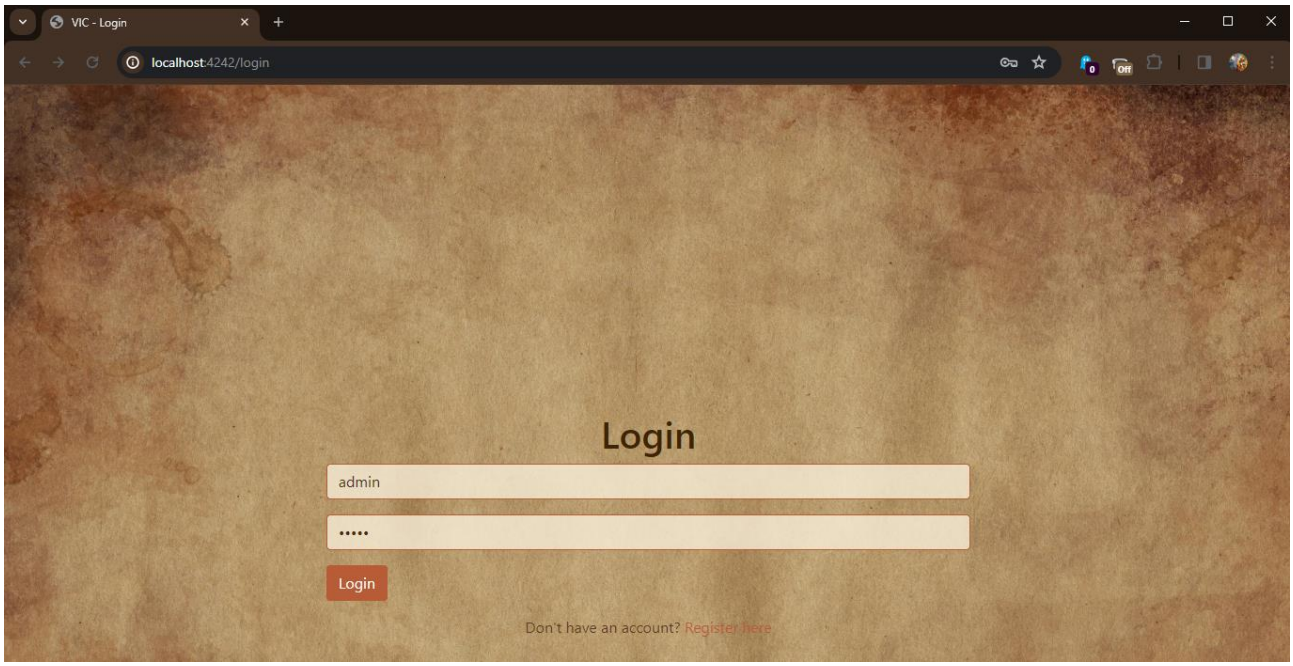


Figure 4.1.4 - Trying to login without an account.

Figure 4.1.5. displays the error I was getting every time I was trying to login without an account.

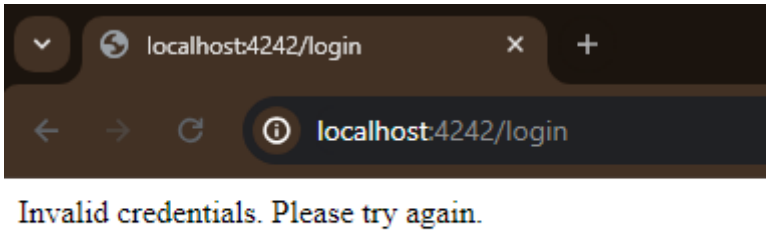


Figure 4.1.5 - Invalid credentials error message.

After a few failed tries, I clicked on the button “Register here” under the login form and then I was redirected at http://localhost:4242/register where I successfully created my own account.

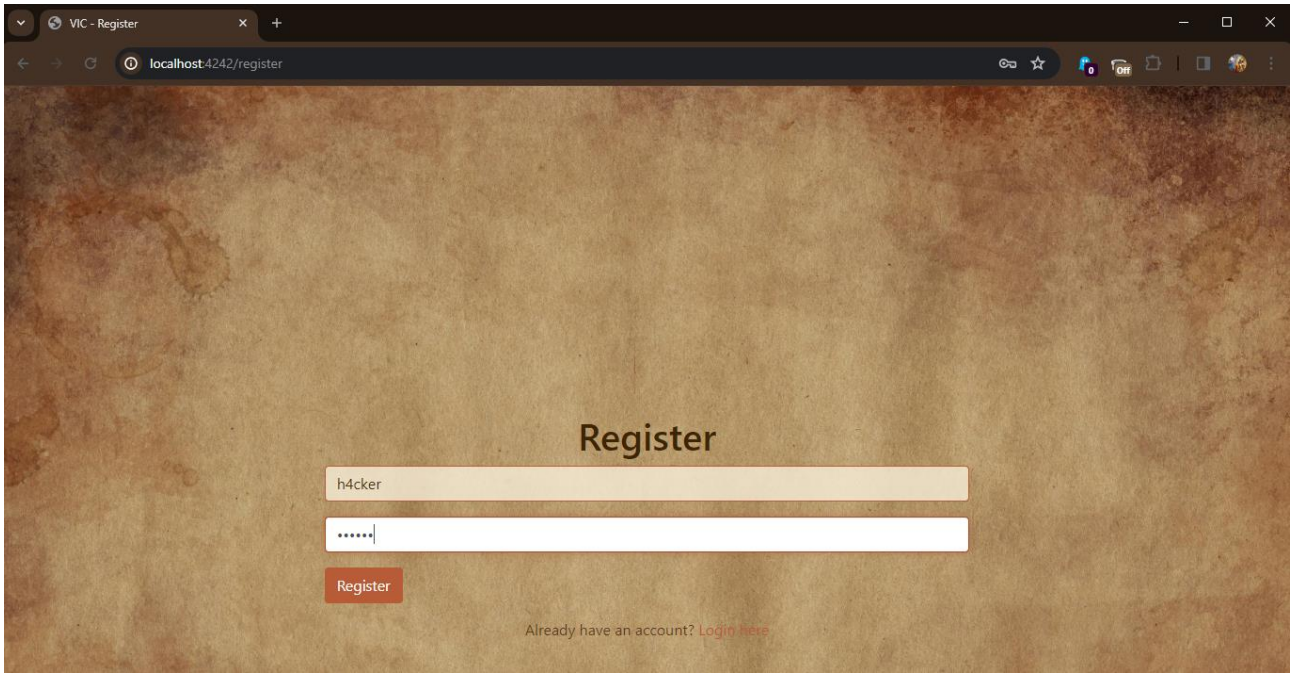


Figure 4.1.6 - Creating account at register page.

Following that, I leaped back to the login page, signed into my new account and got redirected to my profile page at http://localhost:4242/profile as depicted in Figure 4.1.7.

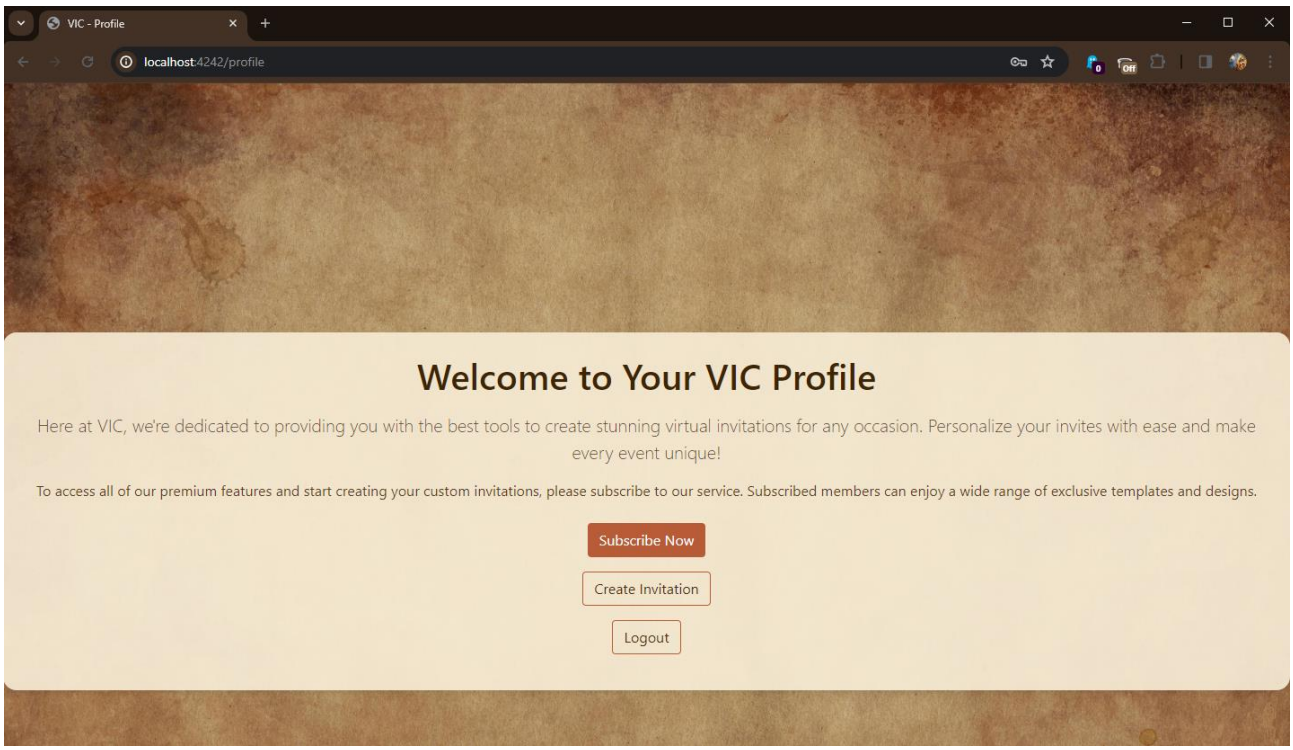


Figure 4.1.7 - My Profile page.

In this page I had three buttons I could press “Subscribe Now”, “Create Invitation” and “Logout”. When I pressed the “Create Invitation” button, I got this error message indicating that I need to be a subscribed member to create my personalized virtual invitations.

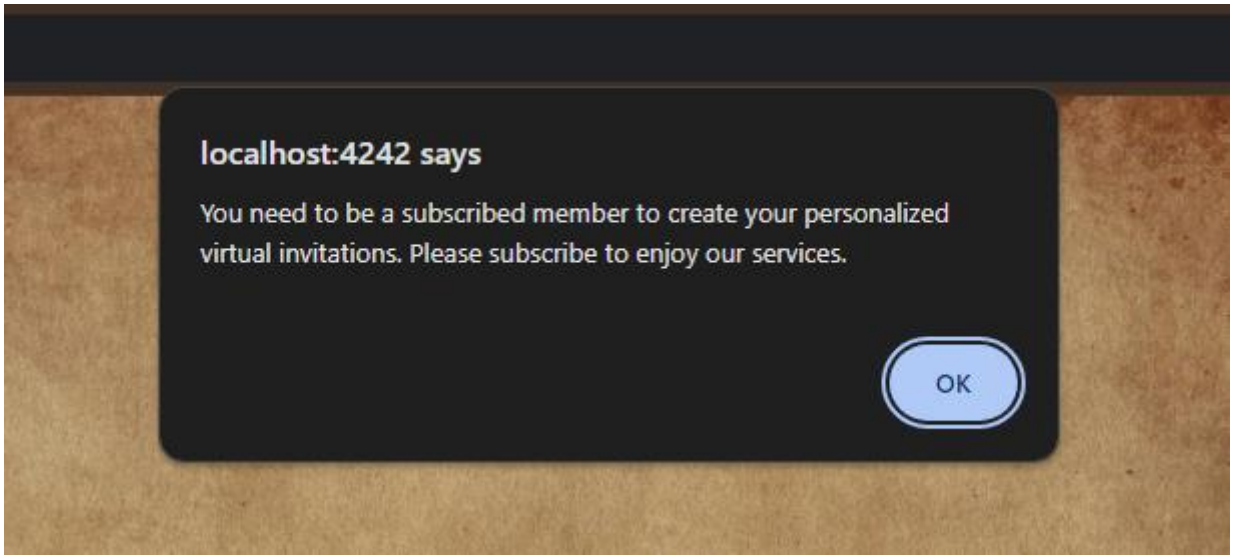


Figure 4.1.8 - Error message indicating you need to be a subscribed user to create virtual invitations.

Then, I pressed the “Subscribe Now” button to subscribe to the service but once again I met up with a new error message. This time the error message was indicating that the subscribe

functionality is under maintenance. So, I had to find out another way to make my profile appeared as a subscribed user.

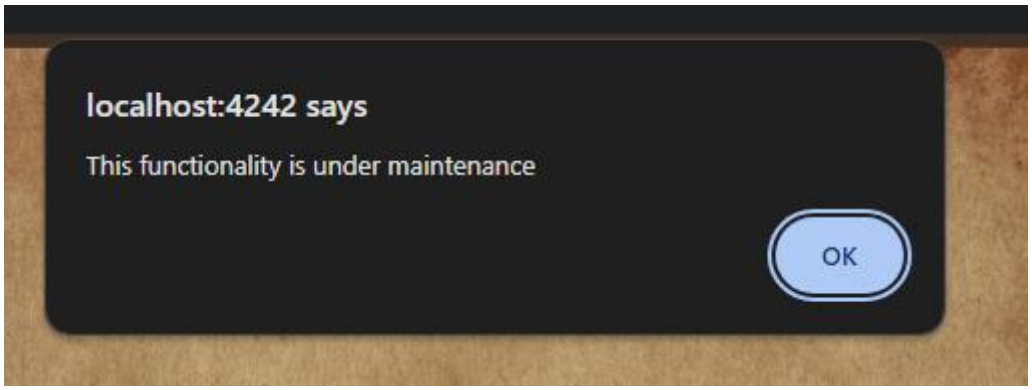


Figure 4.1.9 - Error message indicating that the "Subscribe Now" functionality is under maintenance.

Inspecting the source code of the page, I noticed that there is a script that looks interesting. This script contains two functions, "checkSubscription" and "subscribe".

The first function initializes a "subscribed" variable with the value "no" and then checks the value of this variable. If this value is changed to "yes" in some way, it lets us access the "/create-invitation" page otherwise an error message is displayed.

The second function just sets the previous error message we met for the subscribed functionality that its currently under maintenance.

```
<script>
    function checkSubscription() {
        var subscribed = "no";
        if (subscribed == 'yes') {
            location.href = '/create-invitation';
        } else {
            alert('You need to be a subscribed member to create your personalized virtual invitations. Please subscribe to enjoy our services.');
        }
    }
    function subscribe() {
        alert('This functionality is under maintenance');
    }
</script>
```

Figure 4.1.10 - Viewing script in the source code of the page.

I considered where this "subscribed" variable could come from. These values are typically obtained from either a server-side source (e.g., through an API call that checks the user's subscription status) or from client-side storage mechanisms (like cookies, localStorage, or sessionStorage) that store the user's subscription status.

Indeed, with a quick look at the browser developer tools (by pressing the "F12" button on the browser) under the "Application" tab, we can see that there is a cookie value available.

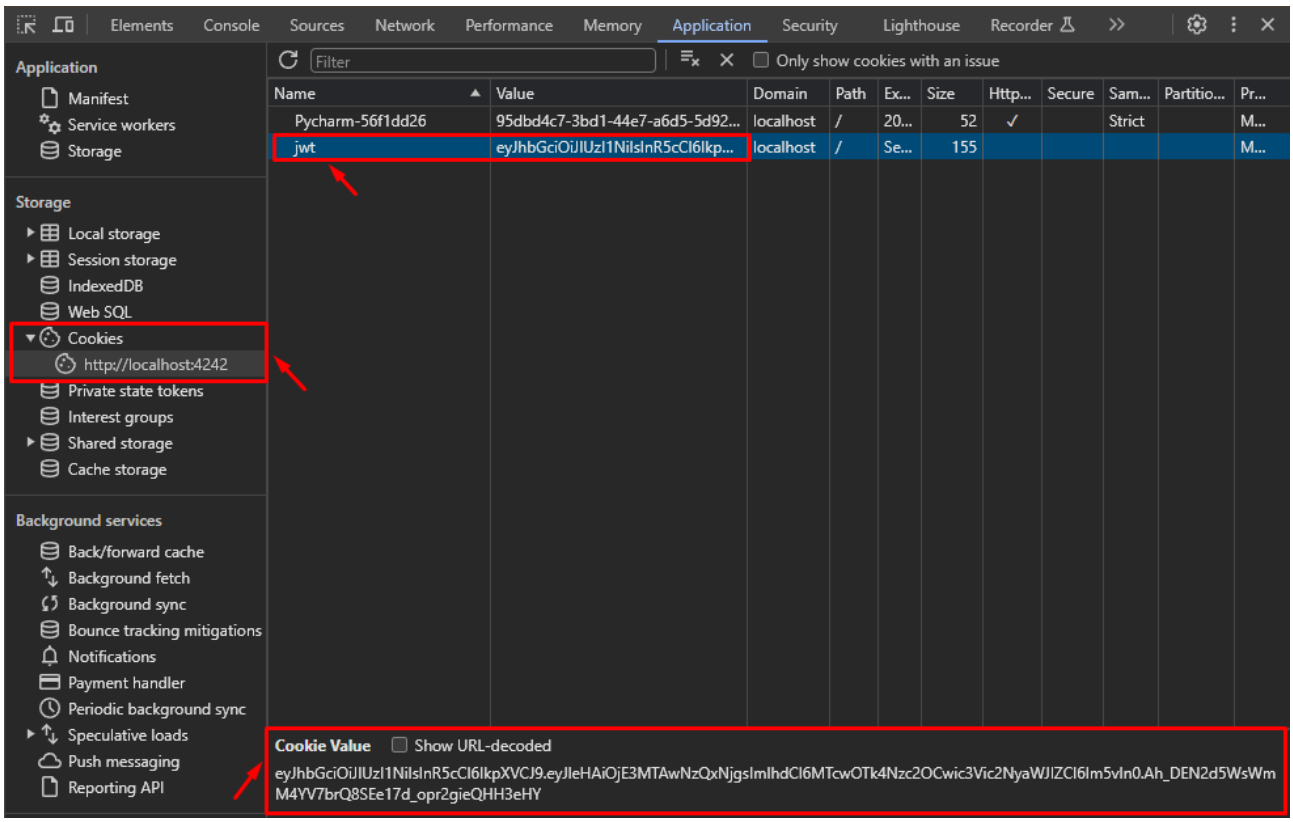


Figure 4.1.11 - Viewing cookies in Browser Developer tools.

The cookie as its name indicates looks like a "JWT" (Jason Web Token) and its value is "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE3MTAwNzQxNjgsImlhdCI6MTcwOTk4Nzc2OCwic3Vic2NyaWJlZCI6Im5vIn0.Ah_DEN2d5WsWmM4YV7brQ8SEe17d_opr2gieQHH3eHY".

This value is most likely to be base64 encoded since JWTs are just base64url-encoded JSON objects. So, I used https://jwt.io/ which is a free online website that allows users to decode, verify and generate JWTs. Figure 4.1.12. depicts the decoded JWT.

Figure 4.1.12 - Decoding JWT value on jwt.io website.

JWTs are digitally signed using a secret key or a public/private key combination, which ensures the integrity and validity of the data they contain. Their value cannot change without invalidating the signature. Altering any part of the token would require re-signing with the secret or private key, which only the issuer possesses.

So, when I modify the “subscribed” value to “yes”, update the cookie and refresh the page, due to the new invalidated signature I’m getting signed out of my account and redirected at the login page.

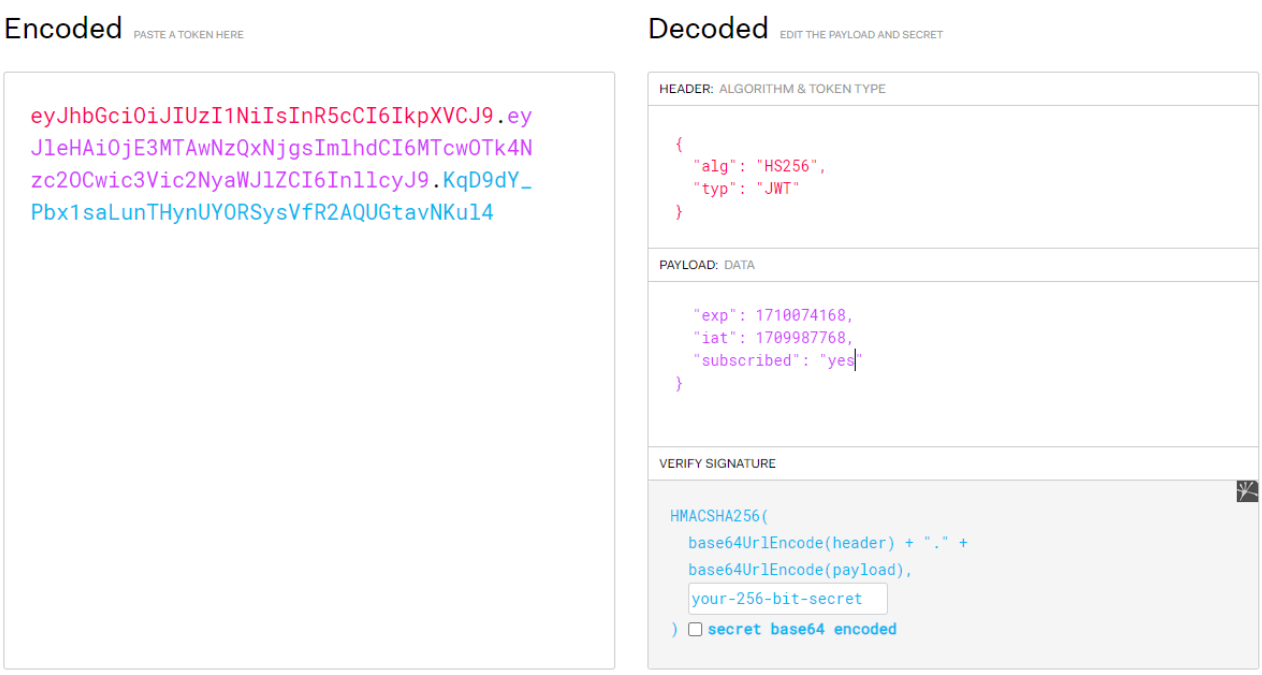


Figure 4.1.13 - Modifying our token's "subscribed" value on jwt.io website.

Knowing that, it looks like we can't make any modifications to our JWT. However, reading the description of the challenge one more time, I noticed that one of the reasons some functionalities are under maintenance are some misconfigurations in the backend and more specifically improper header validation.

Description: "Make your life easier with VIC. Create stylish and cozy invitations quickly and efficiently, perfect for any event! ATTENTION: Currently some functionalities are under maintenance due to critical misconfigurations in the backend of the website (improper header validation)"

Upon further research about JWT vulnerabilities, I found this useful snippet that is possible to have an application in my case as seen in Figure 4.1.14.

JWTs can be signed using a range of different algorithms, but can also be left unsigned. In this case, the `alg` parameter is set to `none`, which indicates a so-called "unsecured JWT". Due to the obvious dangers of this, servers usually reject tokens with no signature. However, as this kind of filtering relies on string parsing, you can sometimes bypass these filters using classic obfuscation techniques, such as mixed capitalization and unexpected encodings.

Figure 4.1.14 - Snippet about JWT header's vulnerability. [19]

Since the description hints that the application validates the header improperly, the first thing I tried based on this article I found is to change both “alg” and “subscribed” values to “none” and “yes” respectively as demonstrated in Figure 4.1.15.

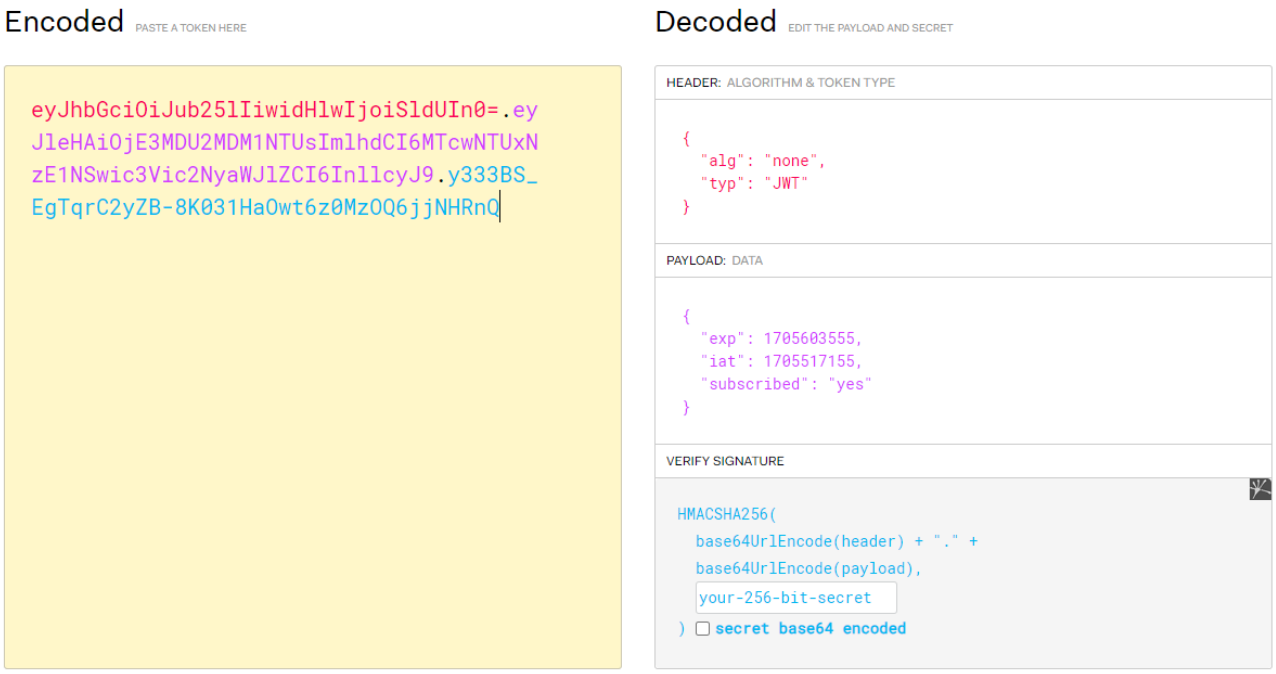


Figure 4.1.15 - Modifying "alg" and "subscribed" values on jwt.io website.

And it worked, using the new modified token, I got successfully redirected to the create-invitation page at http://localhost:4242/create-invitation tricking the application, to consider me as a subscribed user as seen in Figure 4.1.16.

So, I tried to figure out how this page was working. After filling up the input fields and pressing the “Create Invitation” button, my virtual invitation is generated and displayed at the bottom of the page as depicted in the following figure.

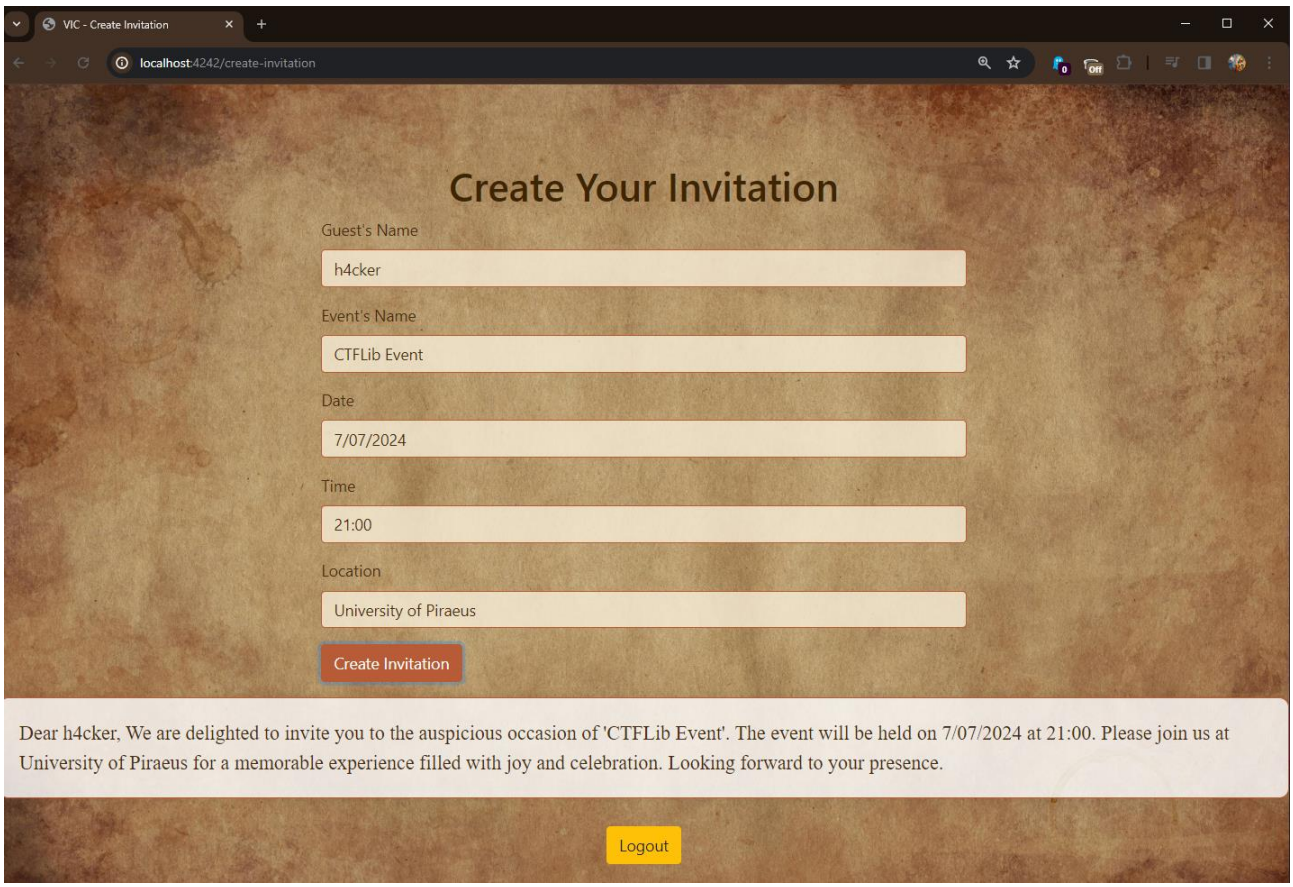


Figure 4.1.16 - Create Invitation page.

The key point here that can lead us to find a possible vulnerability is that each of the inputs we provided to the form, passed directly into the template of the invitation we created. After relevant research, I concluded that it's most probably this vulnerability to be Server-Side Template Injection (SSTI).

In Server-Side Template Injection (SSTI) an attacker exploits web application templates by injecting malicious code into them, allowing for remote code execution or data leakage. This vulnerability arises when user input is improperly sanitized before being included in a template, leading to the execution of unintended server-side commands or code.

Before trying SSTI payloads we need to ensure that this is the vulnerability lies here. As its depicted in Figure 4.1.17. I used the {{7*'7'}} command which indicates that SSTI indeed exists and more especially the template engine that's used is Python's Jinja.

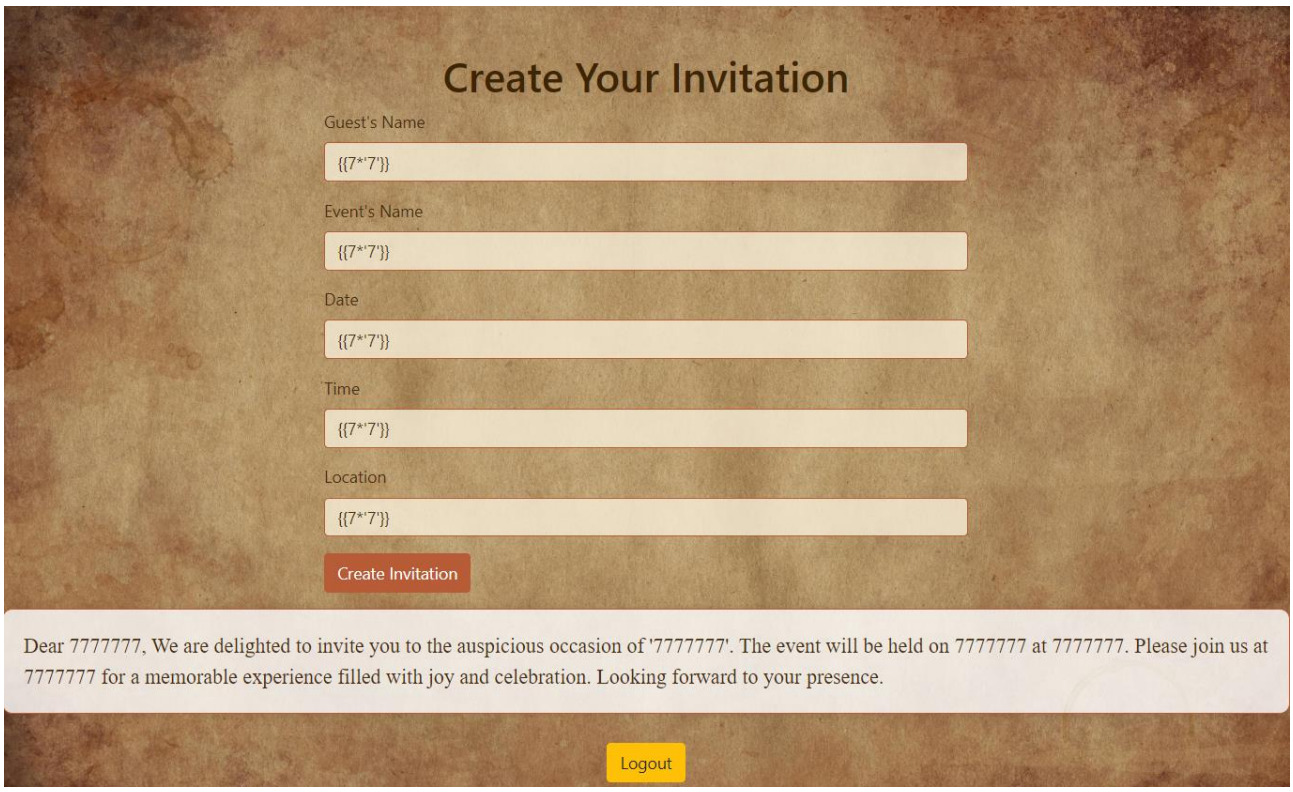


Figure 4.1.17 - Testing if SSTI vulnerability exists.

So, I found some SSTI payloads for Python's Jinja template engine at the infamous GitHub repository "PayloadAllTheThing". [21]

The payload I will use for testing is the following:
{{ cycler.__init__.__globals__.os.popen('id').read() }}

Let's break it down a little bit explaining how it works.

- cycler: A reference to an object with cyclic capability for iterating over a sequence of values that is available in the template environment.
- __init__: Describes the cycler object's constructor method.
- __globals__: This __init__ method attribute gives you access to the module's global variables, which are where the cycler class was defined. It's a method for accessing the global scope in Python through any object or function.
- os: This part of the payload is trying to access the os module, a standard Python module that offers an interface to the operating system, within the __globals__ dictionary.
- popen('id'): Calls the popen method from the os module with the command id, which is a Unix/Linux command that prints the real and effective user and group IDs. popen executes the specified system command.
- read(): Reads the output of the command executed by popen, effectively capturing the result of the id command.

Putting all these together, the constructed payload executes the id system command on the server and attempts to return its output. This could reveal sensitive information about the server's execution environment, such as user privileges, and is indicative of a critical security vulnerability allowing arbitrary command execution.

For the testing of the payload, I will continue using the “Burp Suite Proxy” tool. Utilizing this tool, I will capture the website’s requests using “Burp’s Browser”. Then I will modify one of them with “Burp’s Repeater” and send the new malformed.

Even though I chose the shortest payload I could find online, I got the following error as seen in Figure 6.16. This error indicates that I can’t use more than 25 characters in each one of the fields.

Figure 4.1.18 - Testing payload on BurpSuite Proxy.

Also, I tried to add one more field which was including my payload and I got another error message this time specifying that exactly 5 fields need to be provided as depicted in Figure 4.1.19.

Figure 4.1.19 - Testing payload on BurpSuite Proxy.

I had to bypass the character limit in some way, so I tried to send my payload but this time inside a list. By including my payload in a list, it is considered that it has “length = 1” because its only one item in the list.

As Figure 4.1.20. demonstrates, the payload was executed successfully and we got back the user and group IDs.

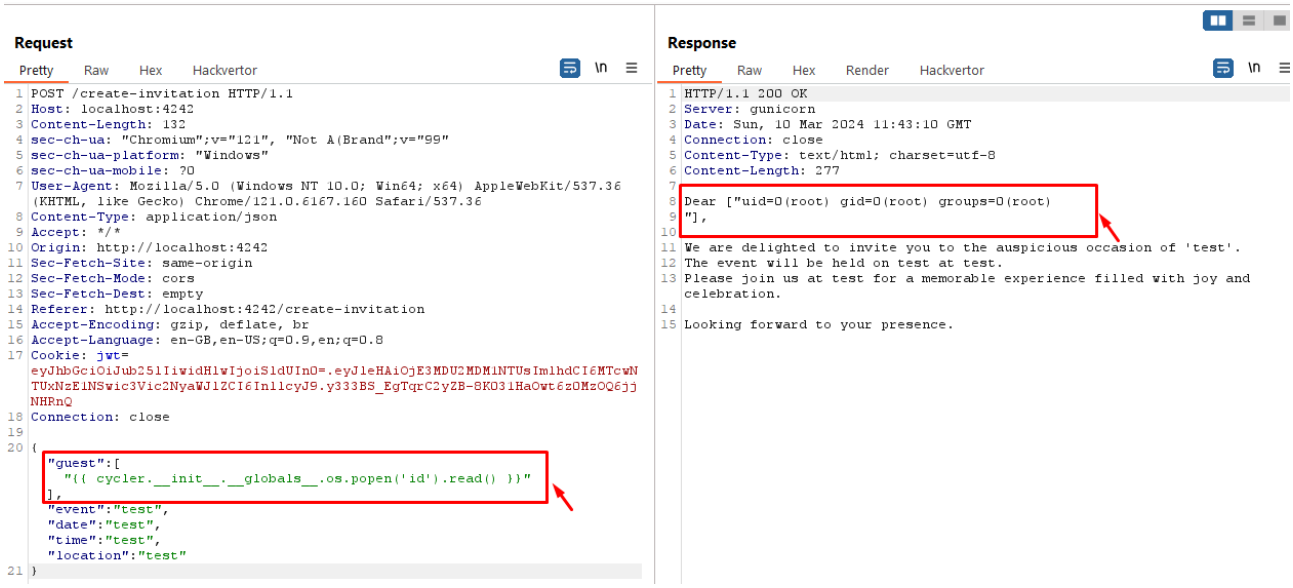


Figure 4.1.20 – Executing payload to retrieve user and group IDs.

Then we resend the payload but instead of asking about the ID’s, we use the “ls” command that lists the content of a directory. As we can see in the response, there is a “flag.txt” that could most possibly contain the hidden flag.

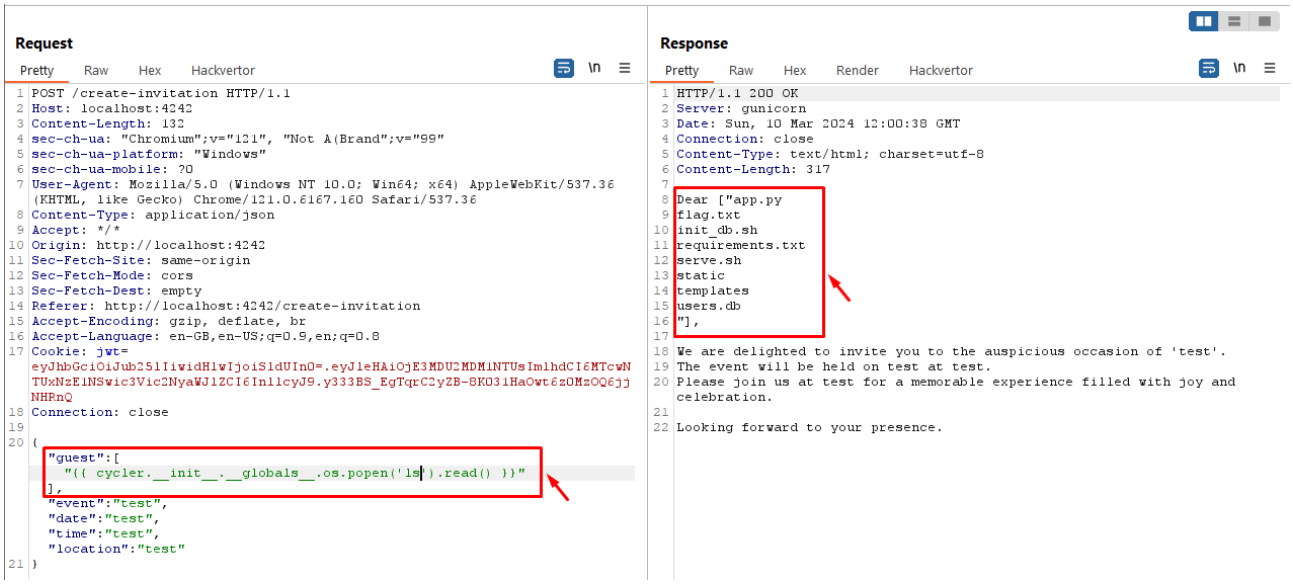


Figure 4.1.21 - Executing payload to list directory's contents.

We replace “ls” with “cat flag.txt” command to view the contents of “flag.txt”.

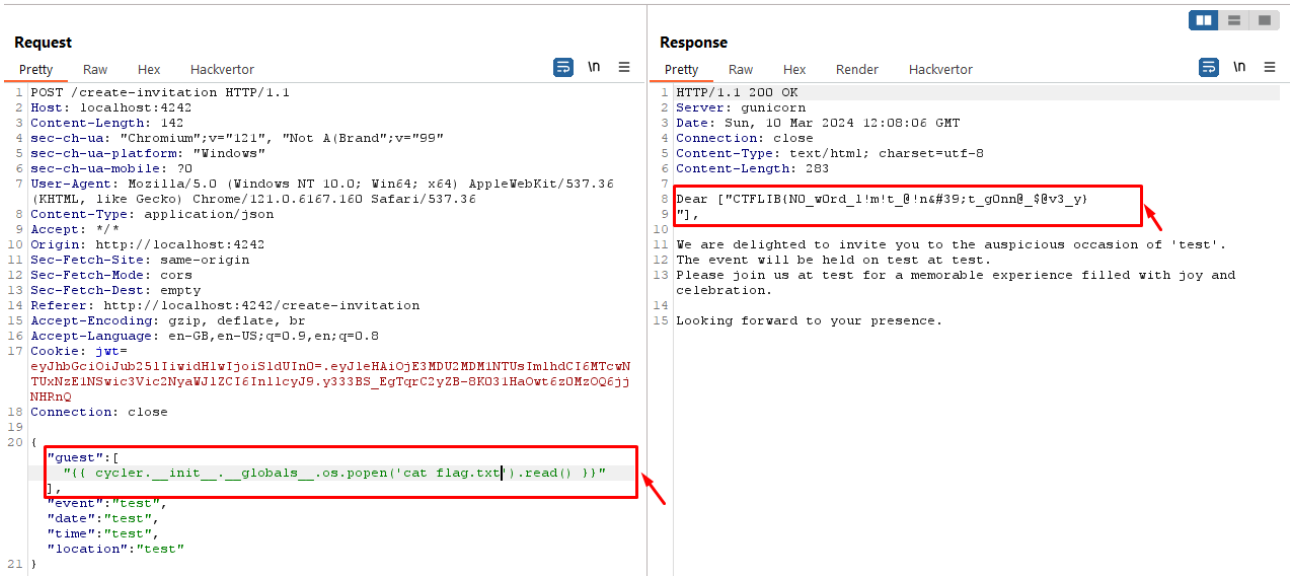


Figure 4.1.22 - Executing payload to view the contents of flag.txt file / Successfully retrieved the hidden flag.

NAILED IT, we successfully retrieved the hidden FLAG!

CTFLIB{N0_w0rd_l!m!t_@!nt_g0nn@_$@v3_y}

4.1.5 Creation

Here is the Dockerfile of the challenge as seen in Figure 4.1.23.

Dockerfile

```
FROM python:3.8-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
ENV APP_PORT 4242

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy the application directory contents into the container at /usr/src/app
COPY ./app /usr/src/app

# Install the dependencies defined in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Expose the application's port
EXPOSE 4242

# Copy the serve.sh script into the container
COPY ./app/serve.sh /usr/src/app

# Give execution rights on scripts
RUN chmod +x /usr/src/app/serve.sh

# Initialize the database and then start the Flask app
CMD python3 -c 'from app import create_users_table; create_users_table()' && ./serve.sh
```

Figure 4.1.23 - Challenge Dockerfile.

This Dockerfile defines python:3.8-slim as the container's base image which is a Debian Buster based image with Python 3.8. installed. Then the Python environment is configured, and the necessary dependencies are also installed. Application's source code is copied in the image and the port of the application is exposed. Execution rights are granted to serve.sh and init_db.sh scripts, which is responsible for starting the server. Finally, the database is initialized by creating the tables before launching the Flask application through the serve.sh script.

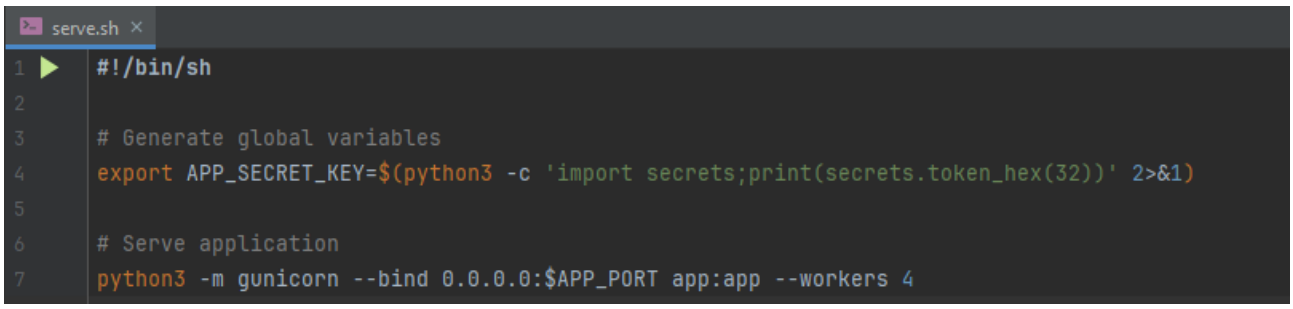


Figure 4.1.24 - serve.sh script for initializing the gunicorn service.

The serve.sh script is initializing the gunicorn service that deploys the flask application as depicted in Figure 4.1.24.

Then we create the app.py which is the main backend file for the application. This file is organized into three primary components: user management, JWT handling, and invitation creation.

```python
# Establish a connection to the SQLite database.
4 usages
def get_db_connection():
    conn = sqlite3.connect('users.db')
    conn.row_factory = sqlite3.Row
    return conn

# Create a users table if it does not already exist in the database.
1 usage
def create_users_table():
    conn = get_db_connection()
    conn.execute('CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, username TEXT, password TEXT)')
    conn.commit()
    conn.close()

# Check if a user exists in the database by username.
1 usage
def user_exists(username):
    conn = get_db_connection()
    user = conn.execute( sql: 'SELECT * FROM users WHERE username = ?', parameters: (username,)).fetchone()
    conn.close()
    return user is not None

# Create a new user with a hashed password.
1 usage
def create_user(username, password):
    hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
    conn = get_db_connection()
    conn.execute( sql: 'INSERT INTO users (username, password) VALUES (?, ?)', parameters: (username, hashed_password))
    conn.commit()
    conn.close()

# Verify a user's credentials.
1 usage
def verify_user(username, password):
    conn = get_db_connection()
    user = conn.execute( sql: 'SELECT * FROM users WHERE username = ?', parameters: (username,)).fetchone()
    conn.close()
    if user and bcrypt.checkpw(password.encode('utf-8'), user['password']):
        return True
    return False
```

Figure 4.1.25 - Part of app.py file for the user management.

```python
@app.route( rule: '/register', methods=['GET', 'POST'])
def register():
    try:
        if request.method == 'POST':
            username = request.form['username']
            password = request.form['password']

            if user_exists(username):
                return 'User already exists. Please choose a different username.', 400

            create_user(username, password)
            return redirect(url_for('login'))
    except Exception as e:
        app.logger.error(f"Error during registration: {e}")
        return 'An error occurred during registration.', 500

    return render_template('register.html')

5 usages
@app.route( rule: '/login', methods=['GET', 'POST'])
def login():
    try:
        if request.method == 'POST':
            username = request.form['username']
            password = request.form['password']

            if verify_user(username, password):
                token = generate_jwt()
                resp = make_response(redirect(url_for('profile')))
                resp.set_cookie( key: 'jwt', token)
                return resp

            return 'Invalid credentials. Please try again.', 401
    except Exception as e:
        app.logger.error(f"Error during login: {e}")
        return 'An error occurred during login.', 500

    return render_template('login.html')
```

Figure 4.1.26 - Part of app.py file for the user management.

For the user management we create the appropriate functions and routes for registering new users, logging in existing users, and logging out. The application interacts with a SQLite database to store and verify user credentials.

```python
# Generate a JWT token. The payload includes a subscribed attribute which could be manipulated.
1 usage
def generate_jwt(subscribed='no'):
    payload = {
        'exp': datetime.datetime.utcnow() + datetime.timedelta(days=1),
        'iat': datetime.datetime.utcnow(),
        'subscribed': subscribed
    }
    return jwt.encode(payload, app.config['SECRET_KEY'], algorithm='HS256')

# Decode a JWT token. This function introduces a JWT vulnerability by allowing the use of 'none' as an algorithm.
2 usages
def decode_jwt(token):
    try:
        # Split the JWT token into Header, Payload, and Signature parts
        header_b64, payload_b64, _ = token.split('.')

        # Decode the Header and Payload from Base64 to JSON
        header = json.loads(base64.urlsafe_b64decode(header_b64 + '==').decode('utf-8'))
        payload = json.loads(base64.urlsafe_b64decode(payload_b64 + '==').decode('utf-8'))

        # Check the algorithm specified in the JWT header
        if header.get('alg') == 'none':
            # If the algorithm is 'none', accept the token without verifying its signature.
            # This simulates a vulnerability where the server trusts the 'alg' header
            # from the client without proper verification.
            return payload
        else:
            # For all other cases, perform normal JWT signature verification.
            # This uses the application's secret key and ensures that the token
            # was not tampered with. It supports only the algorithms specified
            # (in this case, 'HS256').
            return jwt.decode(token, app.config['SECRET_KEY'], algorithms=['HS256'])
    except Exception as e:
        # Log any errors encountered during the JWT decoding process
        app.logger.error(f"JWT Error: {e}")
        return None
```

Figure 4.1.27 - Part of app.py file for the JWT handling.

JWT handling involves generating and decoding JWT tokens, introducing a vulnerability by accepting tokens with "alg":"none".

```python
@app.route( rule: '/create-invitation', methods=['GET', 'POST'])
def create_invitation():
    token = request.cookies.get('jwt')
    if not token:
        return redirect(url_for('login'))

    payload = decode_jwt(token)
    if not payload or payload.get('subscribed') != 'yes':
        return redirect(url_for( endpoint: 'profile', error='unsubscribed'))

    if request.method == 'POST':
        # Extract information from the JSON payload of the request.
        guest = request.json.get('guest', '')
        event = request.json.get('event', '')
        date = request.json.get('date', '')
        time = request.json.get('time', '')
        location = request.json.get('location', '')

        # Ensure exactly five fields are provided.
        if len(request.json) != 5:
            return jsonify({'error': 'Please provide exactly 5 fields.'}), 400

        # Ensure no field exceeds 25 characters in length.
        params = [guest, event, date, time, location]
        # Improper character length check without converting value to a string before checking
        if any(len(i) > 25 for i in params):
            return jsonify({'error': 'Each field must not exceed 25 characters.'}), 400

        # Construct the invitation string using the input from the user.
        # SSTI VULNERABILITY: Using render_template_string with user-controlled input without sanitization or escaping
        # allows an attacker to inject malicious template syntax, leading to arbitrary code execution
        invitation_string = (
            f"Dear {guest},\n\n"
            f"We are delighted to invite you to the auspicious occasion of '{event}'.\n"
            f"The event will be held on {date} at {time}.\n"
            f"Please join us at {location} for a memorable experience filled with joy and celebration.\n\n"
            f"Looking forward to your presence."
        )
        # Render the invitation string directly with the user inputs, leading to an SSTI vulnerability
        return render_template_string(invitation_string)

    return render_template('invitation.html')
```

Figure 4.1.28 - Part of app.py file for creating invitations.

Then we create the invitation creation page that access is allowed only to subscribed users, where the SSTI vulnerability lies. This vulnerability arises due to the use of render_template_string in combination with poor input validation and sanitization.

The appropriate front-end pages are constructed. These pages are simple HTML pages including some JS scripts.

- index.html (home page)
- login.html
- register.html
- profile.html
- invnitation.html

```
<script>
    function checkSubscription() {
        var subscribed = "{{ subscribed }}";
        if (subscribed == 'yes') {
            location.href = '/create-invitation';
        } else {
            alert('You need to be a subscribed member to create your personalized virtual invitations. Please subscribe
        }
    }
    function subscribe() {
        alert('This functionality is under maintenance');
    }
</script>
```

Figure 4.1.29 – JS script that checks if a user is subscribed to the service.

Figure 4.1.29. depicts the checkSubscription function that is implemented in the profile page. This script checks if the current user is subscribed at the service and redirects him at the create invitation page. Otherwise, the user gets the appropriate informative message.

```
<script>
document.getElementById('invitationForm').onsubmit = function(event) {
    event.preventDefault();

    const data = {
        guest: document.getElementById('guest').value,
        event: document.getElementById('event').value,
        date: document.getElementById('date').value,
        time: document.getElementById('time').value,
        location: document.getElementById('location').value
    };

    fetch('/create-invitation', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(data)
    }).then(response => response.text())
      .then(data => {
          const resultElement = document.getElementById('result');
          resultElement.textContent = data;
          resultElement.classList.add('invitation-text'); // Apply the new style class
      })
      .catch(error => console.error('Error:', error));
};
```

Figure 4.1.30 – JS script for asynchronous form submission.

Figure 4.1.30. demonstrated a script that enables asynchronous form submission, by sending the form data straight into the created invitations and avoiding page reloads.

4.2 Jedi Archives

4.2.1 Challenge Overview

This “hard” level challenge consists of two different vulnerabilities within a web application that is related with the Star Wars universe. For the first part of the challenge the participants must exploit an HTTP request smuggling vulnerability to gain unauthorized access to a secret page where they will be able to send messages to the Jedi council and then view their messages. This vulnerability arises due to the use of an outdated vulnerable configuration of a proxy server. For the second part of the challenge, the configuration of the messaging system is also poorly implemented. The participants must take advantage and exploit the vulnerability that lies in the poor implementation of a cryptographic algorithm to regenerate a specific message that encapsulates the hidden flag.

4.2.2 Skills Acquired

This challenge will help the participants acquire several key skills.

- In-depth understanding of HTTP requests and HTTP headers
- Cryptographic algorithms knowledge
- Critical Thinking and Problem-Solving
- Awareness of secure code practices (e.g. never use outdated versions of any service, use randomly generated keys, etc.)
- Prevent / Mitigate HTTP request smuggling vulnerabilities and poor cryptographic implementations
- Exploit HTTP request smuggling vulnerabilities and poor cryptographic implementations

4.2.3 Background Theory

The background theory for this challenge encompasses a fundamental understanding of web application security, especially focusing on HTTP requests, HTTP headers and on robust implementation of cryptographic algorithms.

HTTP (Hypertext Transfer Protocol) is the main protocol for web communication, which enables request-response exchanges between clients (such as browsers) and servers. On the one hand, the client sends an HTTP request to access a web resource, along with the proper action (e.g. GET, POST, etc.) and the resource's URL. On the other hand, the server replies with the content that was requested and a status code (e.g. 200 for success). HTTP request headers are part of both requests and responses for sending additional information such as content type, authentication, and caching policies etc. Figure 4.2.1. illustrates an example of an attacker exploiting an HTTP request smuggling vulnerability, gaining unauthorized access to an admin page.

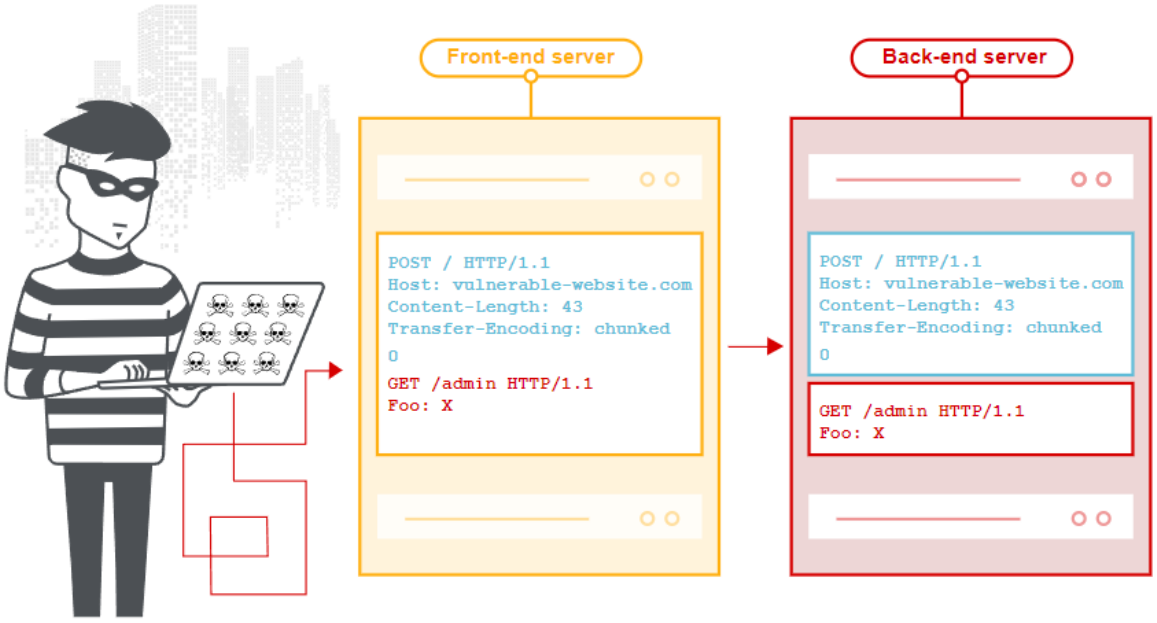


Figure 4.2.1 - Example of an HTTP request smuggling vulnerability exploit. [22]

HTTP request smuggling is a vulnerability that allows attackers to interfere the processing of the HTTP requests from the server by injecting a maliciously crafted request that can subvert the application logic. Here are some good security techniques to mitigate this type of vulnerabilities.

- Always use an up-to-date services like web servers or proxy servers
- Use robust server or proxy configurations
- Use HTTP/2 end to end and disable HTTP downgrading if possible
- Never assume that requests won't have a body
- Deploy WAFs (Web Application Firewalls)

HMAC is a cryptographic algorithm that combines a message with a secret key and applies a hash function (e.g. SHA-256). Then a message authentication code that can be verified by the recipient possessing the same secret key is produced. This is the main reason why the secret key is so significant, and its secrecy is crucial. If the secret key is easily guessable or known, an attacker can compromise the security of the HMAC, by forging messages or bypassing security controls. Figure 4.2.2. depicts a diagram of how HMAC works.

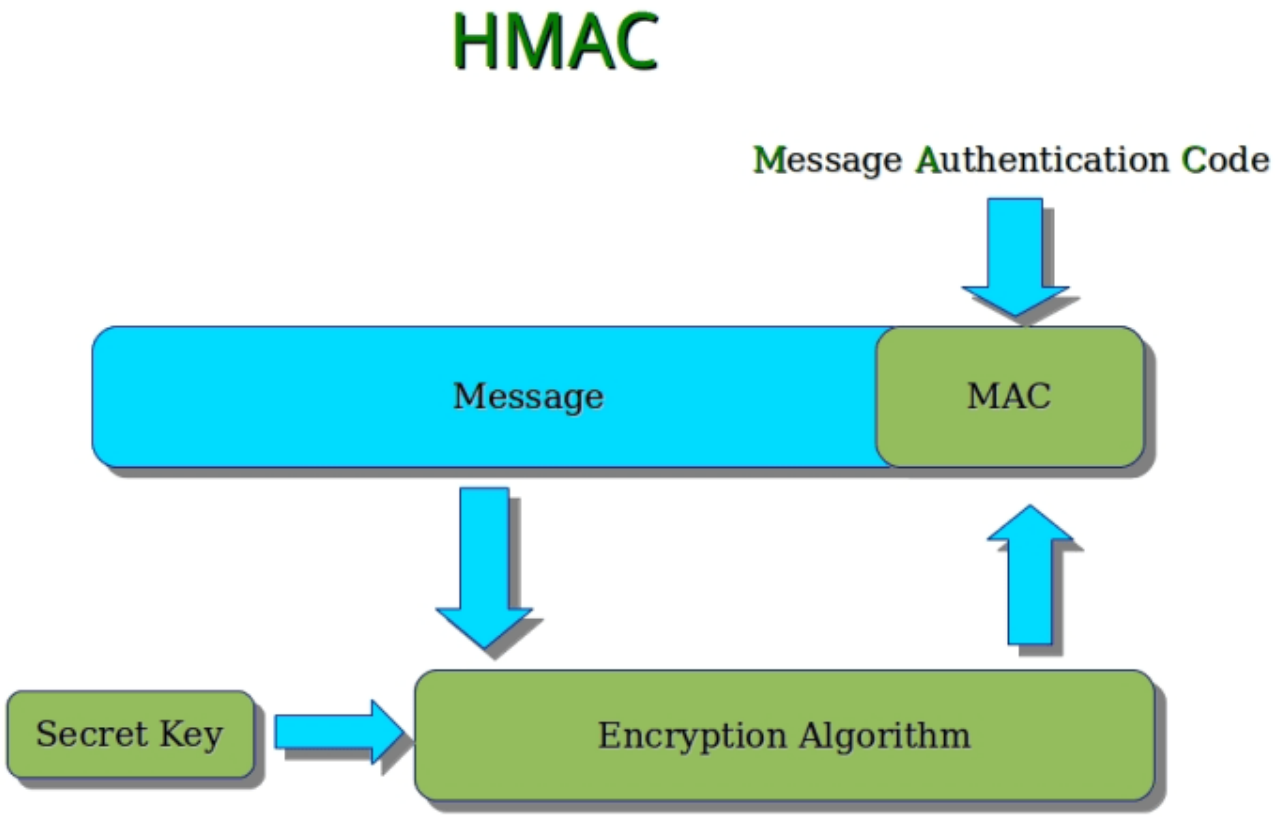


Figure 4.2.2 - HMAC cryptographic algorithm diagram. [23]

Here are some good practices to avoid these vulnerabilities when using cryptographic algorithms in your application.

- Generate secure strong and unpredictable keys
- Apply secure key management
- Make sure your keys are saved in a secured environment
- Perform proper audits and penetration testing

4.2.4 Solution

First of all, we deploy the challenge from CTFLib.

Then we are redirected at http://localhost:4242/index which is the home page of the challenge as seen in Figure 4.2.3.

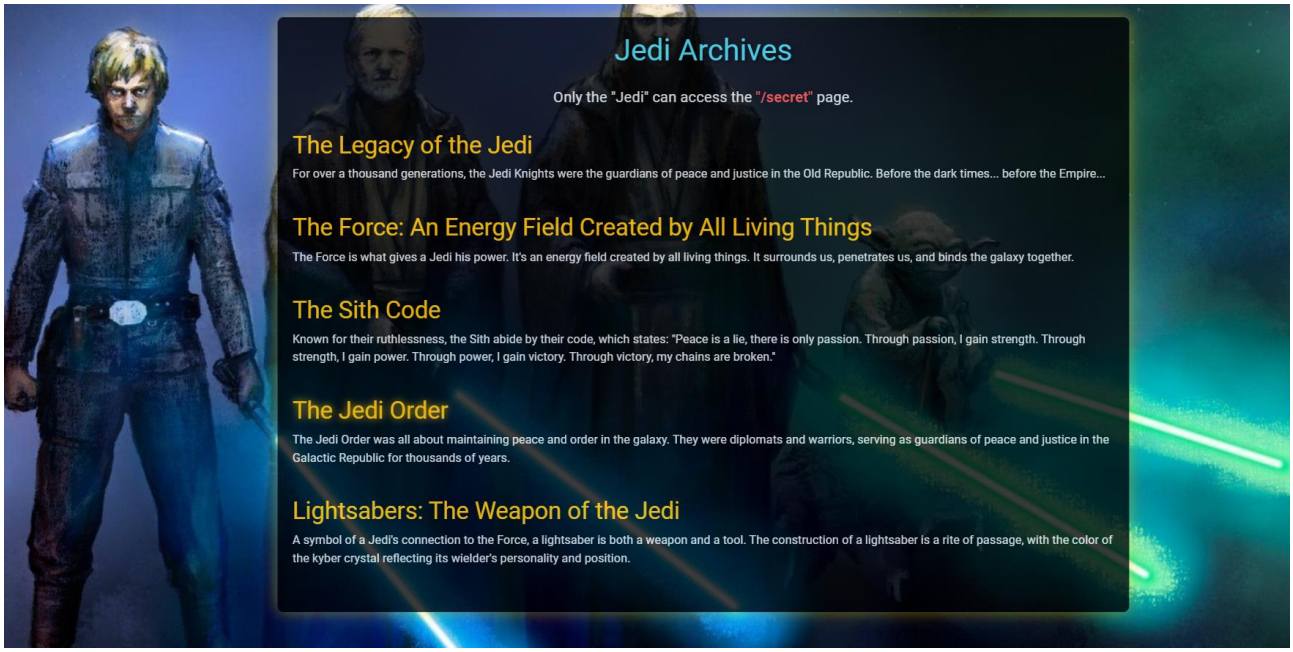


Figure 4.2.3 - Challenge's Home Page / Jedi Archives website.

With a quick look around at the page we can see that there is a “/secret” page available.

After we make sure that there is nothing helpful in the source code of the “/index” page, we head over to http://localhost:4242/secret.

As depicted in Figure 4.2.4. we receive a "403 Forbidden" error while attempting to access the "/secret" page. This implies that access to some challenge pages may be restricted.

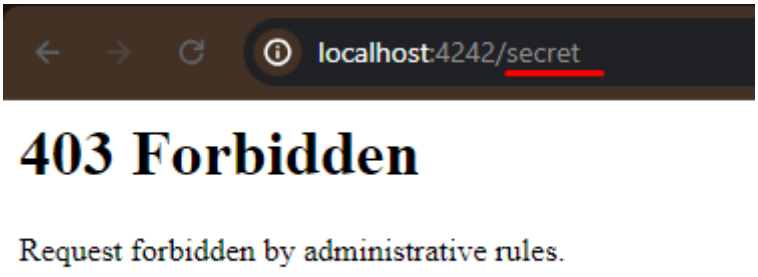


Figure 4.2.4 – Error message while trying to access “/secret” page.

We can see that the challenge's concept is based on "Star Wars" from both the description and the main website. A closer examination of the challenge description reveals that the droids are in charge of creating the page, as seen by the statement, "and the droids have been busy constructing the digital walls."

Essentially, “Droids” are robots. Therefore, it is plausible this web application to contain a "/robots.txt" page. The next figure shows the contents of the "/robots.txt" page that appears when we navigate to http://localhost:4242/robots.txt.

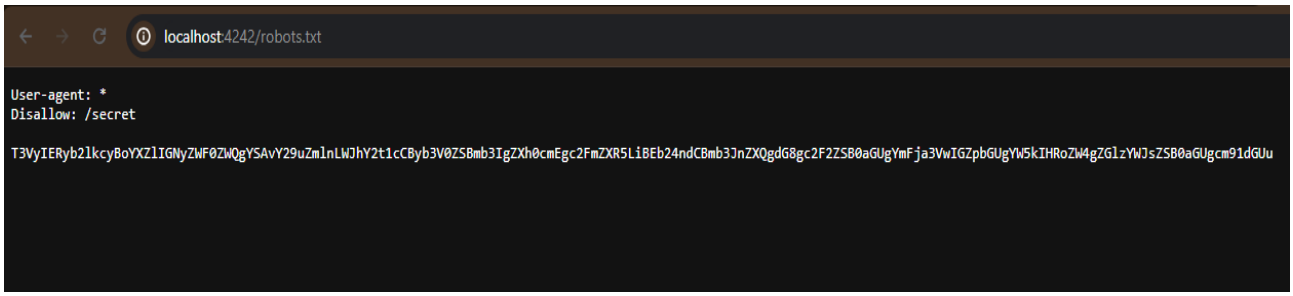


Figure 4.2.5 - Contents of "/robots.txt" page.

A robots.txt file tells search engine crawlers which URLs they can access on your site. This is used mainly to avoid overloading your site with requests; it is not a mechanism for keeping a web page out of Google.

Figure 7.3 illustrates that there is something unusual about this "/robots.txt" page. It appears to be a string that has been base64 encoded or something similar.

This string is indeed base64 encoded, as we can see by using a base64 decoder. Personally, for the decoding of the string I used “hackvertor” which is an encoding/decoding extension on Burp Suite Proxy.

Once the string is decoded, we obtain the following message: "Our Droids have created a /config-backup route for extra safety." Remember to disable the route after saving the backup file.

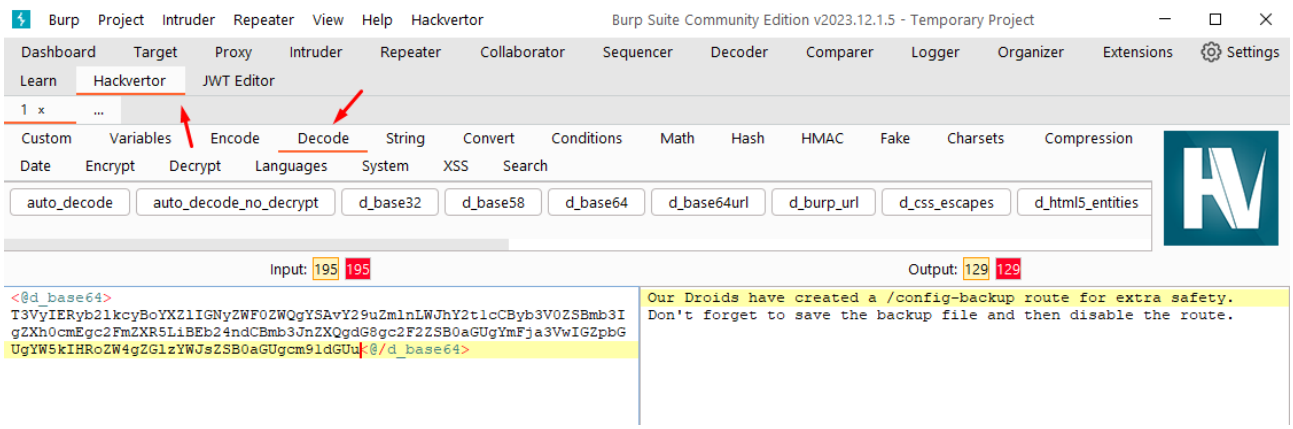


Figure 4.2.6 - Decoding base64 string using Burp's Hackvertor.

This indicates that a "/config-backup" file might be accessible. This page is visible when we go to http://localhost:4242/config-backup and the content of it is shown in the Figure 4.2.7.

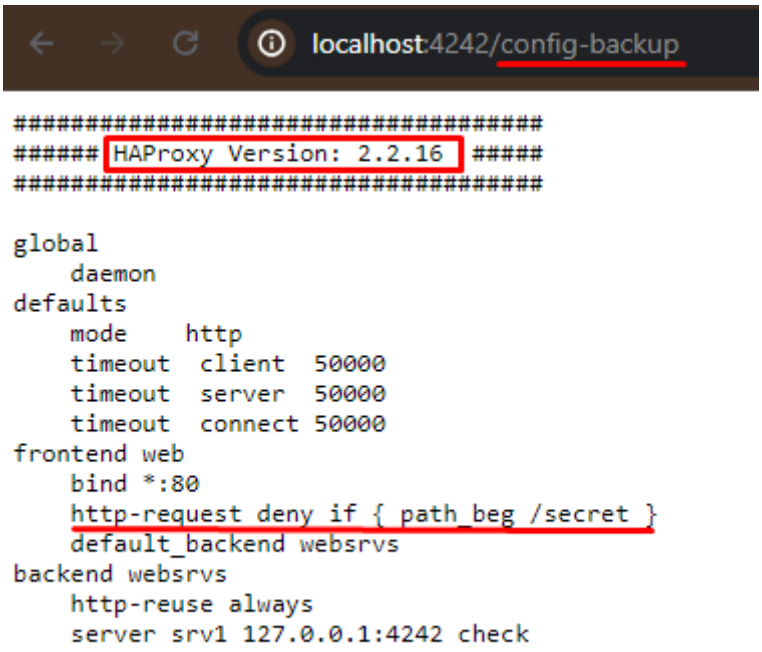


Figure 4.2.7 - Contents of "config-backup" file.

Upon initial observation, it appears to be a proxy server configuration that blocks access to the "/secret" page by establishing an access control list. Additionally, the top of the backup file has a reference indicating the precise proxy server version being utilized, which is "HAProxy Version: 2.2.16."

So, let's hop on some research about "HAProxy Version: 2.2.16." and its possible vulnerabilities.

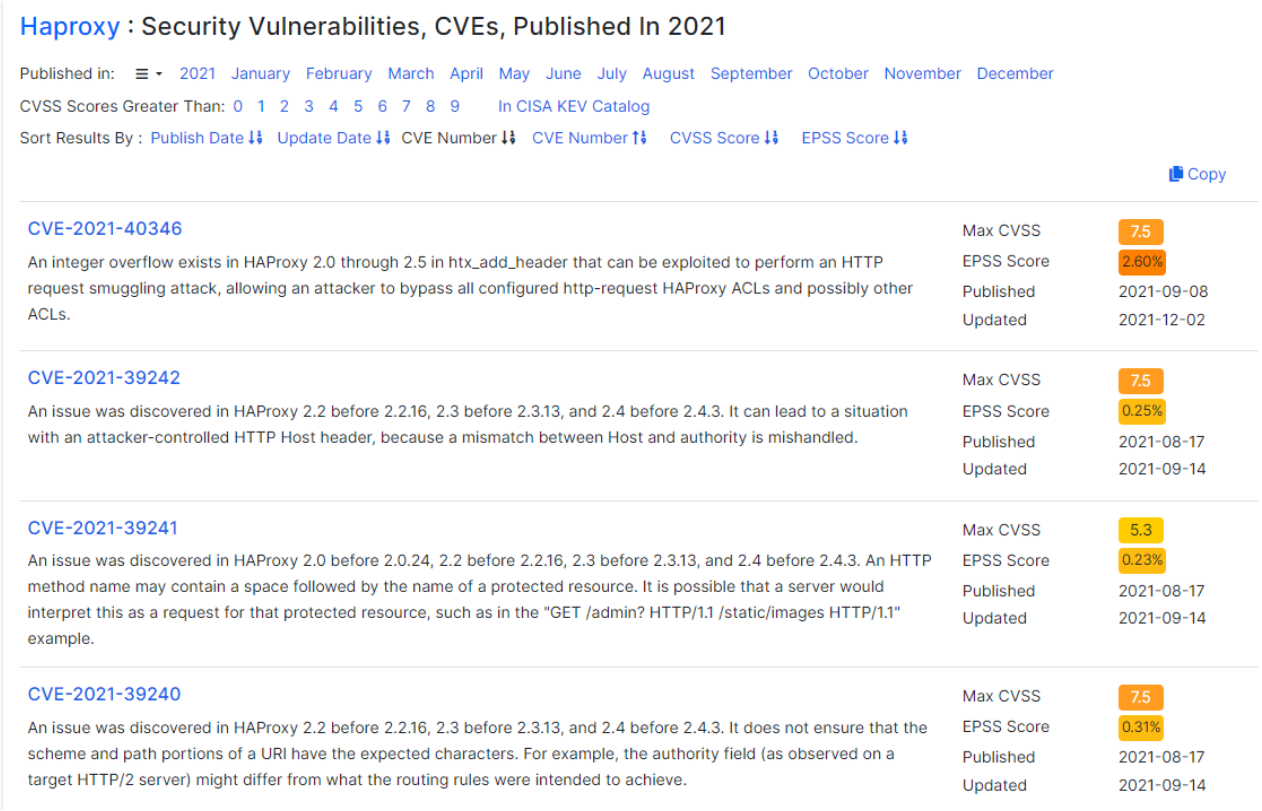


Figure 4.2.8 - Searching about HAProxy CVEs. [24]

After further investigation, we have determined that there are a few CVEs related to HAProxy. Based on the information provided, I have concluded that “CVE-2021-40346” is the most likely to be applicable in our situation. This is because it describes how to conduct an "HTTP REQUEST SMUGGLING” attack to bypass HAProxy ACLs and possibly other ACLs. Also, our HAProxy version (2.2.16) is between 2.0 and 2.5.

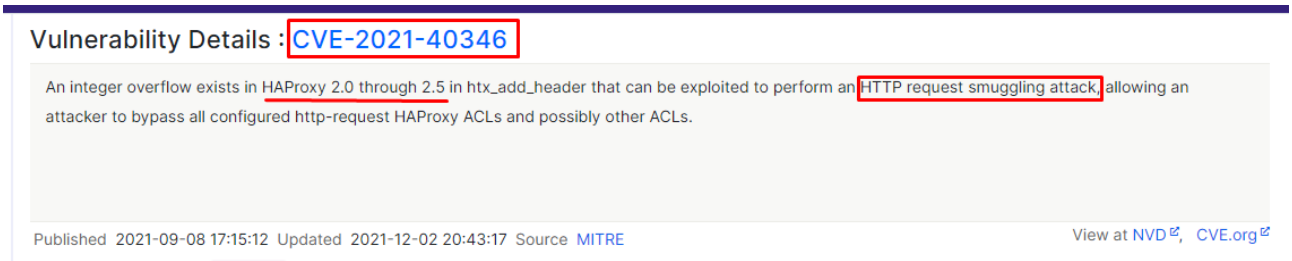


Figure 4.2.9 - CVE-2021-40346 vulnerability details. [25]

So, let’s do some more extensive googling while searching for the specific code of the “CVE-2021-40346” as seen in Figure 4.2.10.

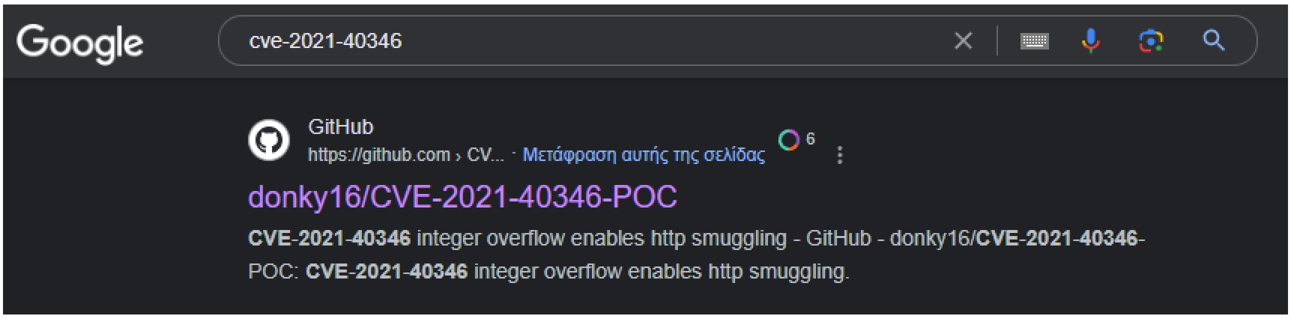


Figure 4.2.10 - Searching about CVE-2021-40346.

Perfect, I found this GitHub repository [26] that looks like it is demonstrating how to bypass a "HAProxy" ACL by performing an HTTP REQUEST SMUGGLING that is caused by an integer overflow. [27]

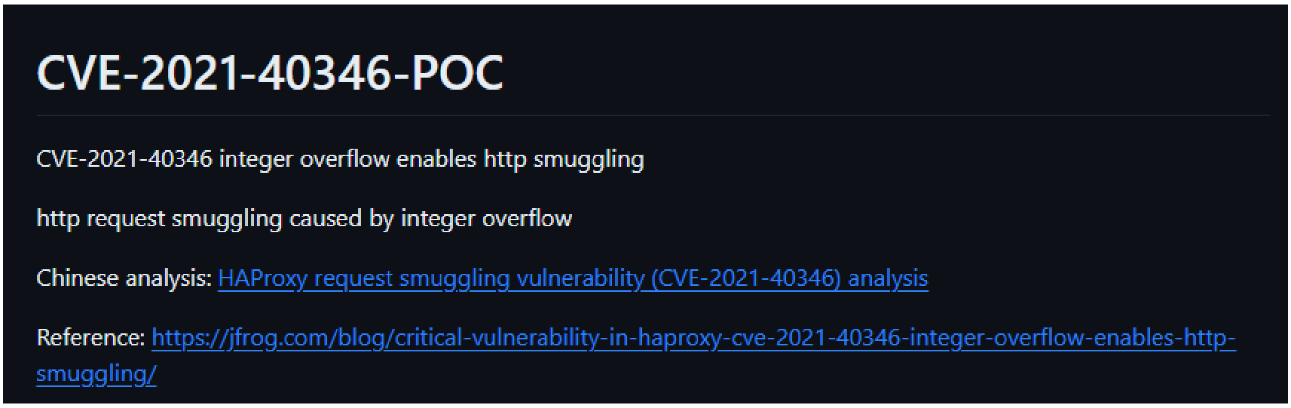


Figure 4.2.11 - CVE-2021-40346 Proof of Concept.

The payload's execution is depicted in Figure 4.2.12.

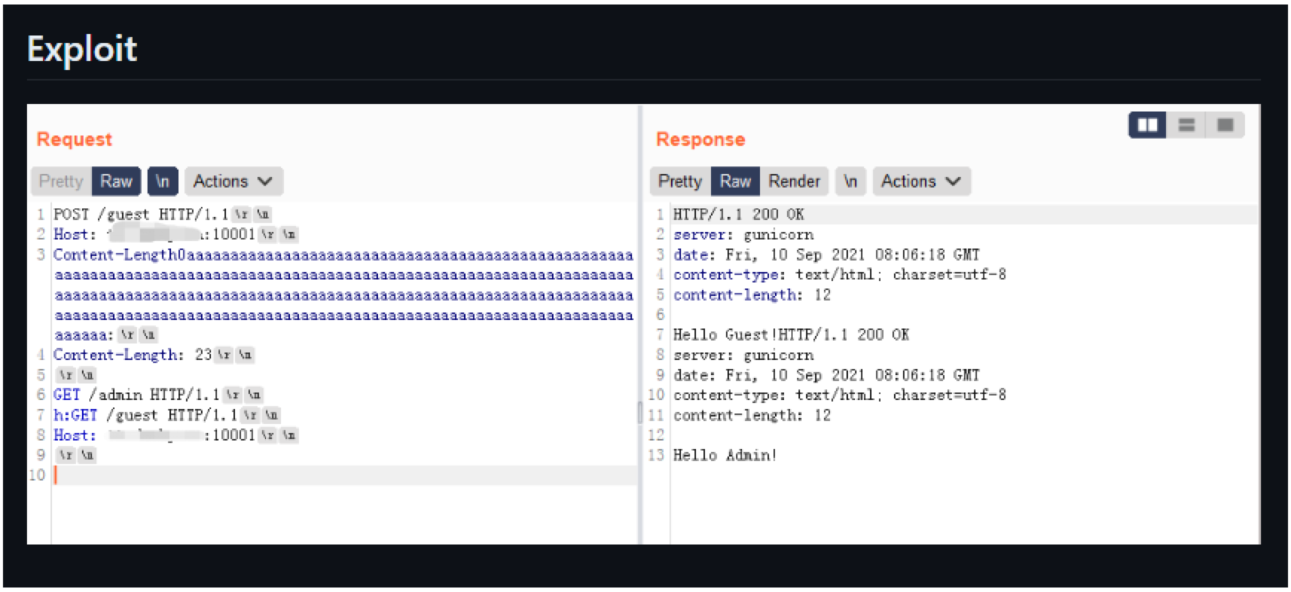


Figure 4.1.12 - Executing the payload.

Now let's try to perform the HTTP REQUEST SMUGGLING in our case so we bypass the ACL that is blocking access to the “/secret” page.

To intercept the request, I used Burp Suite Proxy as demonstrated in Figure 4.2.13.

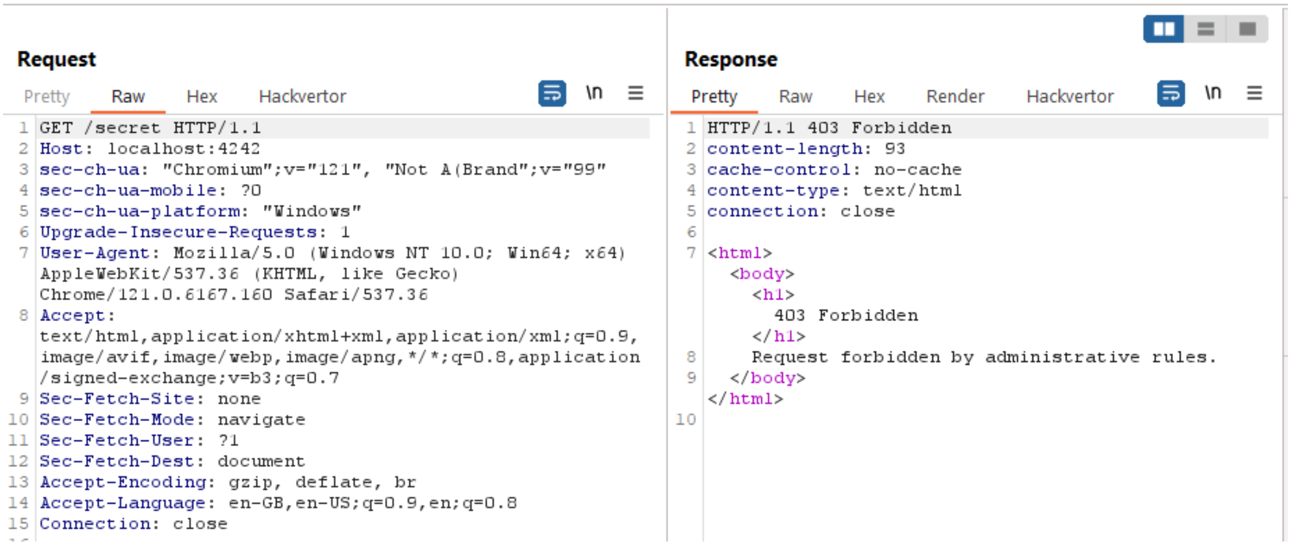


Figure 4.2.13 - Intercepting websites traffic using Burp Suite Proxy.

Afterwards with Burp’s Repeater, I modified the captured request and sent the new malformed.

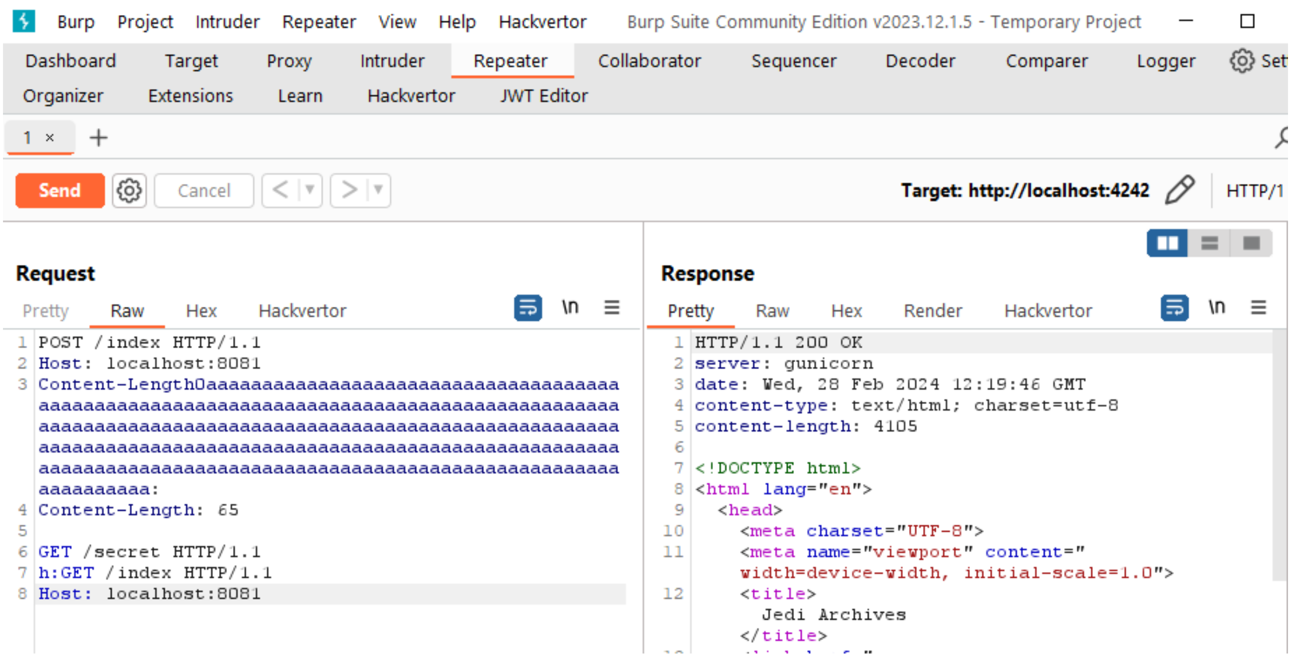


Figure 4.2.14 - Modifying the captured request and sending the malformed that includes the payload using Burp’s Repeater.

The "/index" page appears in the response after sending the updated request for the first time, as seen in the figure.

However, when we send the modified request for the second time, the integer overflow occurs successfully, and we get the “/secret” page in the response as seen in Figure 4.1.15

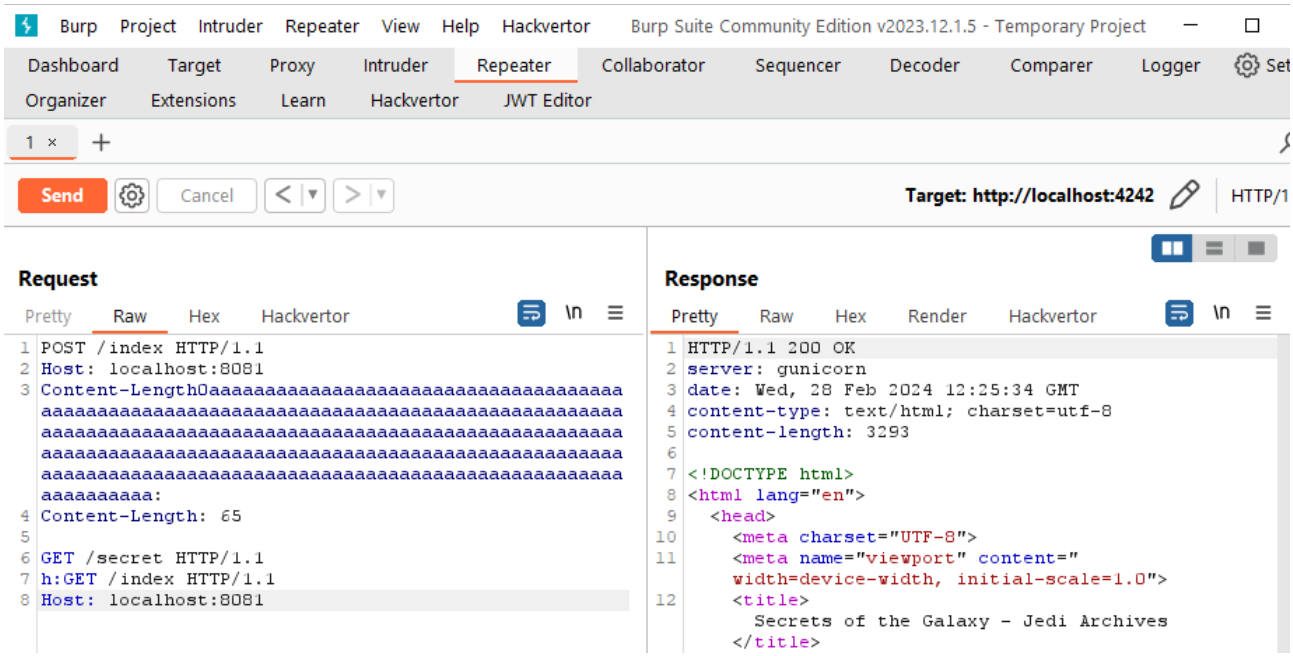


Figure 4.2.15 - Executing the payload successfully.

Next, we right-click on the response, and we choose “Show response in browser” to access the secret page through Burp’s browser.

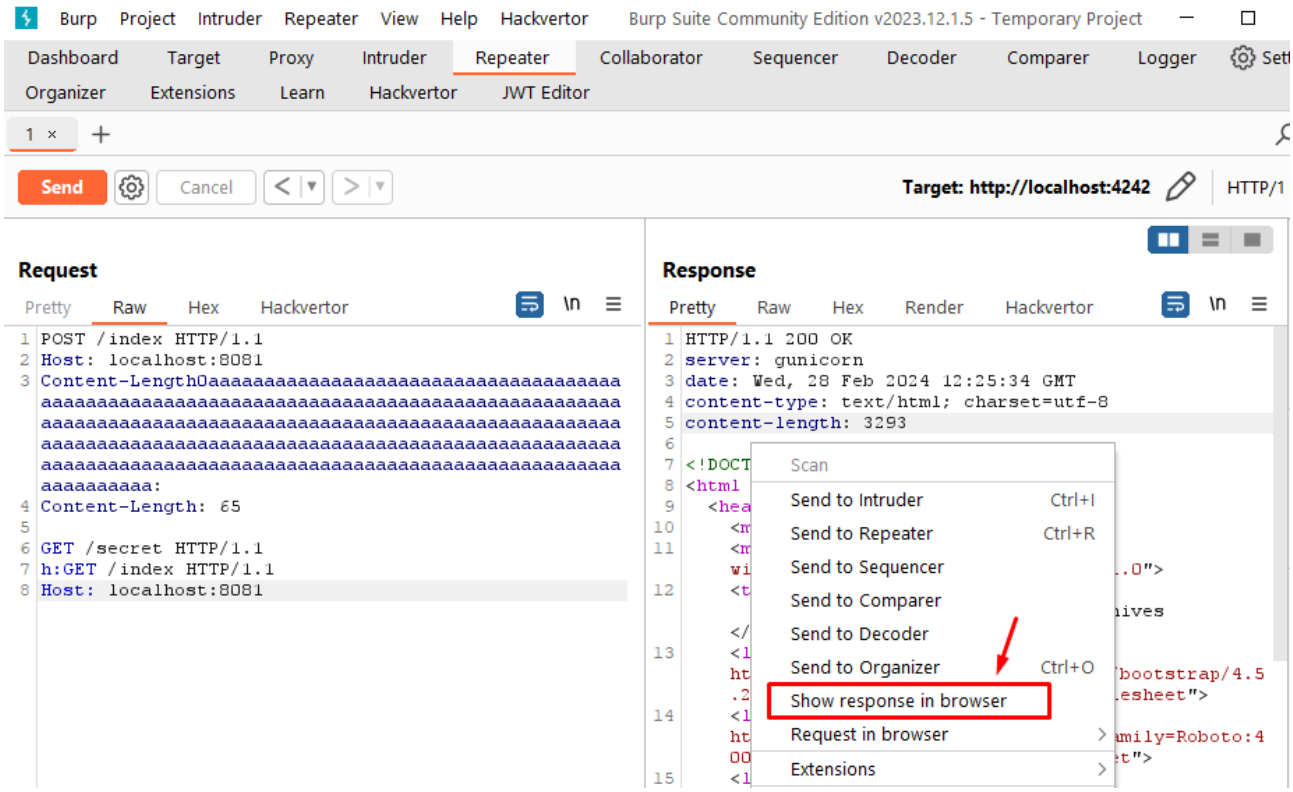


Figure 4.2.16 - Choosing the "Show response in browser" from the response.

In doing so, we are able to access the "/secret" page and view its contents in Burp's browser as depicted in Figure 4.2.17.

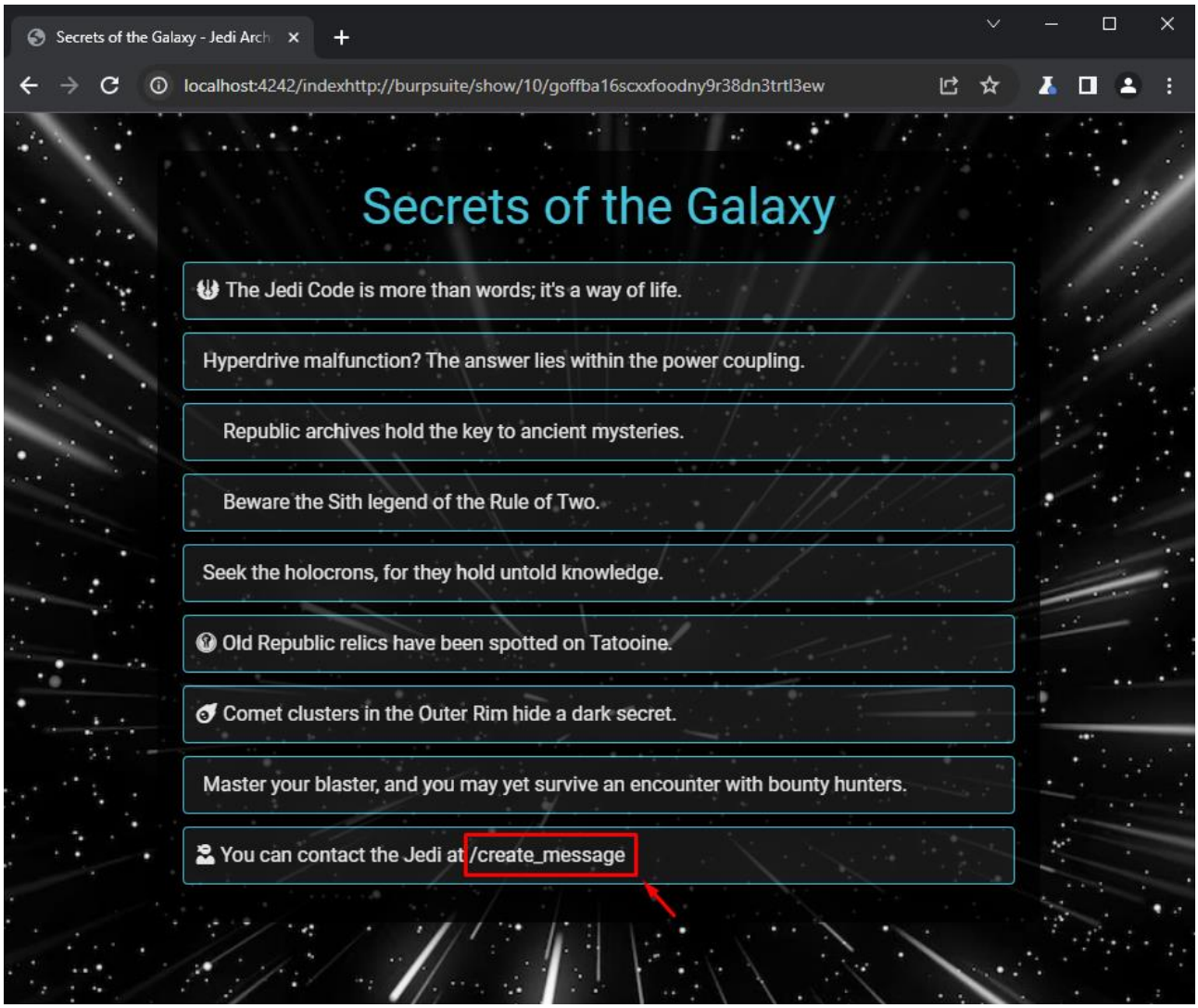


Figure 4.2.17 – Displaying "/secret" page contents on Burp's browser.

Upon a brief glance at the contents of the page, we notice that we can contact the Jedi at the "/create_message" page, therefore there may be another page to investigate.

Indeed, there is a new page at http://localhost:4242/create_message where we can send messages to the Jedi as seen in Figure 4.2.18.

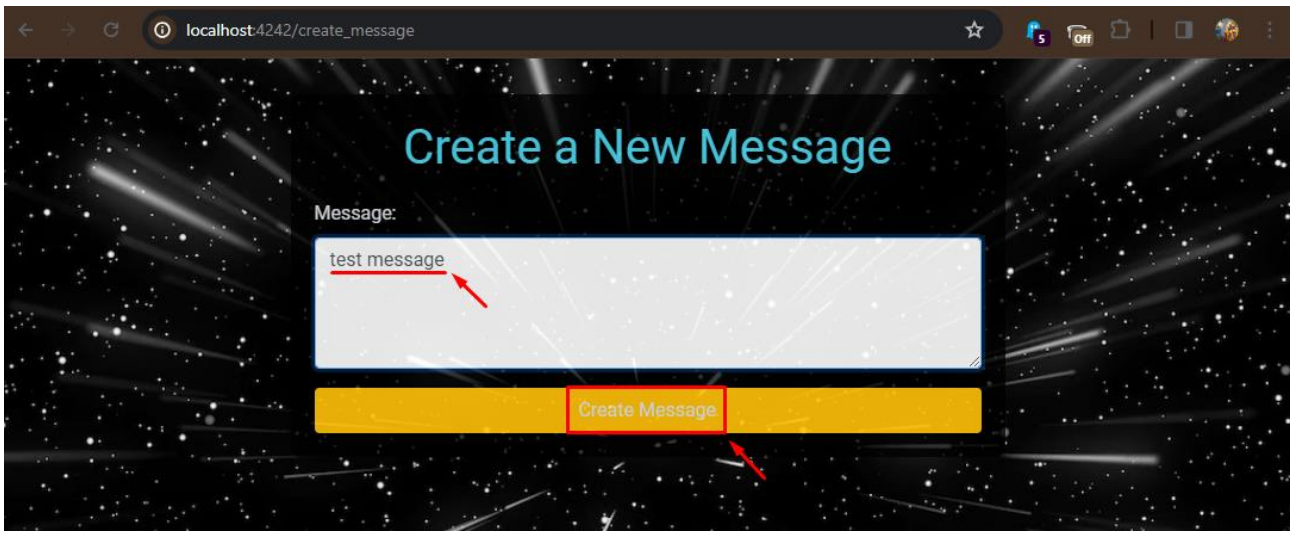


Figure 4.2.18 - Contents of "/send_message" page.

After sending a “test message” we notice that we are getting redirected to a “/message” page where the message we just sent to the Jedi is displayed. With a closer look on the URL, we observe that our displayed message has an “id” and a “token” parameter.

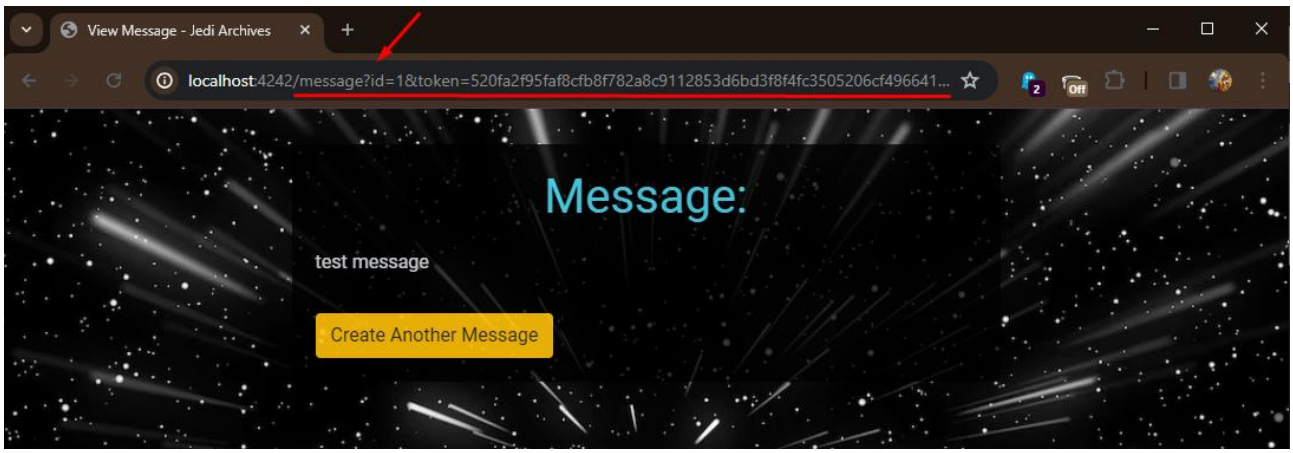


Figure 4.2.19 - “id” and “token” parameters.

Let's experiment a little bit so we can understand how these messages work. After sending a new message it seems like a new “id” is assigned in ascending order and a new “maybe” unique token. Pressing the "Create Another Message" button will allow us to return to the "/create_message" page and send more messages.

If we try to change the “id” of the message from “id=2” to “id=1” so we can access the previous message we sent, we get the following error as depicted in the figure.

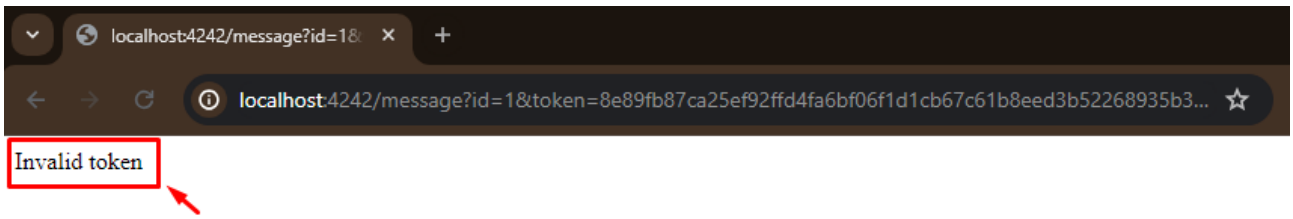


Figure 4.2.20 - Error message while trying to access previous messages by modifying the "id" parameter.

While searching at the "/create_message" page source code I found a comment that may be useful for the continuation of the challenge. It indicates that we can find a configuration of how the messages work on the “/message_conf” page as seen in Figure 4.2.21.

```
<body>
    <div class="container">
        <h1>Create a New Message</h1>
        <form method="post">
            <div class="form-group">
                <label for="data">Message:</label>
                <textarea id="data" name="data" rows="4" class="form-control"></textarea>
            </div>
            <button type="submit" class="btn">Create Message</button>
            <!-- Messages conf needs update soon. Find it at /message_conf -->
        </form>
    </div>
```

Figure 4.2.21 - Useful comment in the page's source code.

At http://localhost:4242/message_conf, there is a configuration showcasing how the messaging system works. So, let's break it down and try to understand this configuration. I transferred the code into “PyCharm” which is a “Python IDE” to facilitate more detailed analysis and enhance visibility.

Figure 4.2.22 - Contents of "message_conf" file in PyCharm IDE.

```python


@app.route( rule: '/create_message', methods=['GET', 'POST'])
def create_message():
    if request.method == 'POST':
        data = request.form.get('data', 'no data provided.')
        id = len(messages)
        messages.append(data)
        token = generate_token(id)
        return redirect(f'/message?id={id}&token={token}')
    return render_template('create_message.html')

@app.route('/message')
def view_message():
    id = request.args.get('id', '-1')
    token = request.args.get('token', '')

    try:
        id = int(id)
    except ValueError:
        return 'Invalid ID format'

    if id < 0 or id >= len(messages):
        return 'Message not found'

    expected_token = generate_token(id)
    if token != expected_token:
        return 'Invalid token'

    message = messages[id]
    return render_template( template_name_or_list: 'view_message.html', message=message)


if __name__ == '__main__':
    app.run(debug=True)
```

Figure 4.2.23- Contents of "message_conf" file in PyCharm IDE.

The Python file provided is a Flask web application that allows users to create and view messages. It also includes a mechanism for generating and validating tokens to ensure that only users with the correct token can view a specific message.

The flag seems to be stored as the first message of a “messages” list.

The “generate_token” function generates a secure token using the cryptographic algorithm HMAC with SHA-256 hash function, based on the secret key and a given ID. However, some key components appear to be vulnerable.

As seen in Figure 4.2.24. the “SECRET_KEY” variable is a string. Actually, it should be calling “str(uuid4())” to generate a “UUID” and we can also verify that from the “from uuid import uuid4” on imports that has grey color. This indicates that it's not used anywhere in the code.

So, we conclude that since the “SECRET_KEY” is a static string, the token that is generated is only determined by the “id” of each message.

Because of this, I created my own Python script with a static SECRET_KEY = "secret-{uuid4}" that allows me to generate the token for any message by simply selecting the appropriate "id."

We already know that the flag is the first message of a “messages” list. To regenerate this message’s token, I set the "message_id=0".

solve.py

```python
import hmac
import hashlib

# Use the same simplified SECRET_KEY
SECRET_KEY = "secret-{uuid4}"

# Message ID for which you want to generate the token
message_id = "0"

# Recreating the token using the same method as in the Flask app
recreated_token = hmac.new(SECRET_KEY.encode(), message_id.encode(), hashlib.sha256).hexdigest()

print(recreated_token)
```

Figure 4.2.24 – Python script for regenerating a message's token.

After running the “solve.py” script we get the following regenerated token:
“6a33f9165c4f27ee039e44350f87ecbb1ddb2bddd1ef6a0cbef7684fa9cd8a7c”

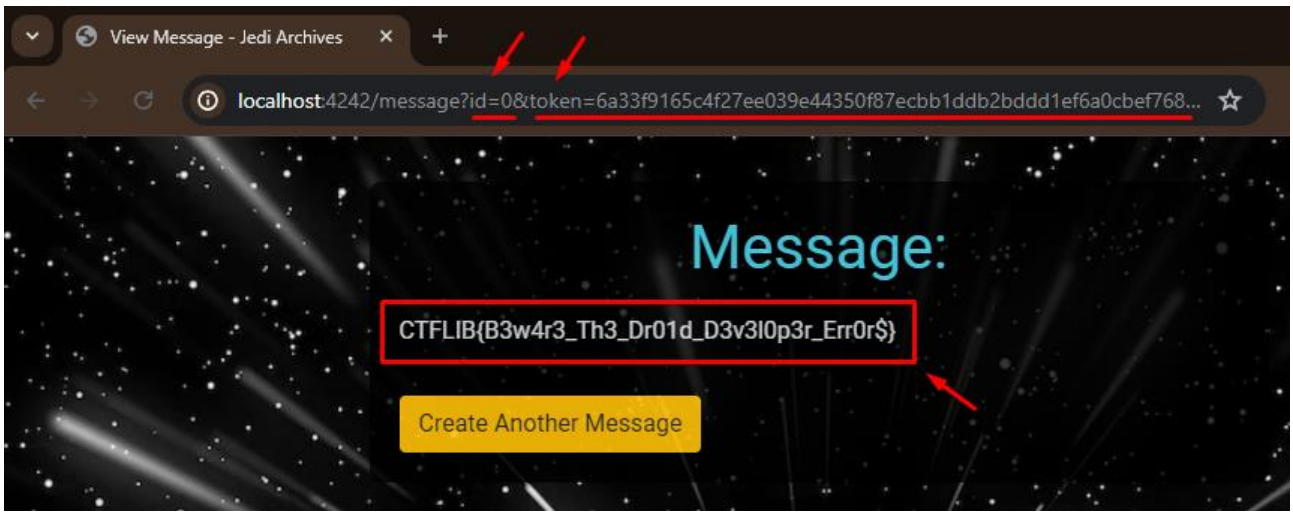


Figure 4.2.25 - Retrieving the first message of the list / Successfully retrieved the hidden Flag.

We try to retrieve the first message and BOOM; we got the hidden FLAG:

CTFLIB{B3w4r3_Th3_Dr01d_D3v3l0p3r_Err0r$}

4.2.5 Creation

Here is the Dockerfile of the application as seen in Figure 4.2.26.

Dockerfile

```
# Use HAProxy as the base image
FROM haproxy:2.2.16

# set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
ENV APP_PORT 4242

# Install dependencies required for building Python 3.8.10 from source
RUN apt-get update && apt-get install -y \
    build-essential \
    libffi-dev \
    libssl-dev \
    zlib1g-dev \
    liblzma-dev \
    libbz2-dev \
    libreadline-dev \
    libsqlite3-dev \
    wget \
    curl \
    llvm \
    libncurses5-dev \
    libncursesw5-dev \
    xz-utils \
    tk-dev \
    && rm -rf /var/lib/apt/lists/*

# Download, compile, and install Python 3.8.10
RUN curl -O https://www.python.org/ftp/python/3.8.10/Python-3.8.10.tar.xz \
    && tar -xf Python-3.8.10.tar.xz \
    && cd Python-3.8.10 \
    && ./configure --enable-optimizations \
    && make -j 8 \
    && make altinstall \
    && cd .. \
    && rm -rf Python-3.8.10.tar.xz Python-3.8.10
```

Figure 4.2.26 - First part of the Dockerfile.

```
# Install pip for Python 3.8
RUN curl -O https://bootstrap.pypa.io/get-pip.py \
    && python3.8 get-pip.py \
    && rm get-pip.py

# Set the working directory in the container
WORKDIR /app

# Copy the requirements file into the image
COPY ./app/requirements.txt /app/requirements.txt
# config-backup.txt is in the src/app directory
COPY ./app/config-backup.txt /app/config-backup.txt


# Install python requirements
RUN pip install -r requirements.txt --no-cache-dir

# Copy the Flask application and HAProxy configuration
COPY ./app /app
COPY ./config/haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg

# Copy and grant execution permissions to the serve script
COPY ./app/serve.sh /serve.sh
RUN chmod +x /serve.sh

# Expose the application's port
EXPOSE 8081 80 4242

# Start the services using the script
CMD ["/bin/sh", "/app/serve.sh"]
```

Figure 4.2.27 - Second part of the Dockerfile.

This Dockerfile defines haproxy:2.2.16 as the container's base image. Then it installs and compiles Python and sets up all the necessary dependencies (for flask, gunicorn). It copies the application source code and the configuration files in the image and sets up execution permissions to a serve script. Finally, it exposes the application's port, and sets the serve script to run at container startup, which initializes HAProxy and the Flask application.

serve.sh

```
#!/bin/sh

# Generate global variables
export APP_SECRET_KEY=$(python3.8 -c 'import secrets;print(secrets.token_hex(32))' 2>&1)

# Start HAProxy in the background
haproxy -f /usr/local/etc/haproxy/haproxy.cfg &

# Serve application with Gunicorn using Gevent workers
python3.8 -m gunicorn --keep-alive 10 -k gevent --bind 0.0.0.0:$APP_PORT -w 20 app:app
```

Figure 4.2.28 - serve.sh script for initializing HAProxy and gunicorn services.

The serve.sh script initiates the HAProxy service which is a proxy server to manage incoming web traffic to our Flask application. Then the Flask application runs with the Gunicorn service that is using Gevent workers for more efficient request handling.

haproxy.cfg

```
global
    daemon
defaults
    mode    http
    timeout  client  50000
    timeout  server  50000
    timeout  connect 50000
frontend web
    bind *:80
    # Deny requests to /secret, mitigating direct access to potentially sensitive data
    http-request deny if { path_beg /secret }
    default_backend websrvs
backend websrvs
    http-reuse always
    # Define server with address 127.0.0.1:4242, checking its health for availability
    server srv1 127.0.0.1:4242 check
```

Figure 4.2.29 - HAProxy server configuration.

Then we create the vulnerable HAProxy server configuration file that essentially blocks access to the "/secret" page by establishing an access control list as seen in Figure 4.2.29.

```python
# Read the flag from flag.txt, or provide a default one if not found
1 usage
def read_flag():
    try:
        with open('flag.txt', 'r') as file:
            return file.read().strip()
    except FileNotFoundError:
        return "CTFLIB{ex4mpl3-fl4g}"

# Initialize the messages list with the first message containing the Flag
FLAG = read_flag()
messages = [FLAG]
# Here us the vulnerability, using format string incorrectly
SECRET_KEY = "secret-{uuid4}"
```

Figure 2 1 - Messaging system configuration.

```python
def generate_token(id):
    # Generate a secure token for message authentication, but using an insecure SECRET_KEY
    return hmac.new(SECRET_KEY.encode(), str(id).encode(), hashlib.sha256).hexdigest()
```

Figure 4.2.30 - Messaging system configuration.

Then we create the vulnerable messaging system configuration for the second part of the application. Users are able to create and view messages. Also, we implement a mechanism for generating and validating tokens to ensure that only users with the correct token can view a specific message. The hidden flag is stored as the first message of a “messages” list. The “generate_token” function generates a token using the cryptographic algorithm HMAC with SHA-256 hash function, based on a secret key and a message ID.

```python
@app.route( rule: '/create_message', methods=['GET', 'POST'])
def create_message():
    # Create a message and generate a token for it
    if request.method == 'POST':
        data = request.form.get('data', 'no data provided.')
        id = len(messages)
        messages.append(data)
        token = generate_token(id)
        return redirect(f'/message?id={id}&token={token}')
    return render_template('create_message.html')

@app.route('/message')
def view_message():
    # Validate token and display message if valid
    id = request.args.get('id', '-1')
    token = request.args.get('token', '')

    try:
        id = int(id)
    except ValueError:
        return 'Invalid ID format'

    if id < 0 or id >= len(messages):
        return 'Message not found'

    expected_token = generate_token(id)
    if token != expected_token:
        return 'Invalid token'

    message = messages[id]
    return render_template( template_name_or_list: 'view_message.html', message=message)
```

Figure 4.2.31 - Messaging system configuration.

Figure 4.2.31. illustrates the two main routes of the messaging system. The first route is for creating messages and generating authentication tokens. The second route is for viewing the messages. When a message is sent, its saved and the user is redirected to view it. If the token or the message ID is invalid during viewing, error messages are displayed.

The appropriate front-end pages for every back-end functionality are constructed (index.html, secret.html, create_message.html, view_message.html).

5. Conclusion

In conclusion, this thesis presents the vital importance of the Capture the Flag (CTF) challenges in the field of cybersecurity education, proving to be an innovative and impactful way to enhance cybersecurity expertise. By engaging learners in real-world scenarios, CTF challenges serve not just as a tool for skill enhancement but as a bridge to strategic and analytical thinking that are essential in problem-solving. This creative approach has an appealing gamified nature and refers to a wide range of people, from beginners to professionals, offering a fun and exciting learning experience that ignites their love and passion for the cybersecurity field.

Furthermore, as the landscape of cyber threats continues to evolve in complexity and scale, the competitive and collaborative environment fostered by CTF challenges becomes invaluable. It not only allows individuals to test their skills but also promotes a culture of continuous learning and resilience. This makes CTF challenges an important method in developing robust cybersecurity skills which place a strong emphasis on both theoretical knowledge and practical experience.

Although CTF challenges are an excellent way to give students realistic, hands-on experience with real-world events and enhance their cybersecurity education, combining them with other essential components like formal coursework, certifications, and ongoing professional development creates a more complete and well-rounded educational experience.

6. References

[1] Hack TheBox Challenges

[2] CTFLib website – by CTFLib team from the department of Digital Systems of the University of Piraeus

[3] NoSQL injection – by Portswigger

[4] Access control vulnerabilities and privilege escalation – by Portswigger

[5] Insecure Direct Object Reference (IDOR) — Web-based Application Security, Part 6 – by Spanning

[6] File upload vulnerabilities – by Portswigger

[7] File Upload General Methodology – by Hacktricks

[8] Path traversal – by Portswigger

[9] OS Command Injection – by Portswigger

[10] Apache HTTP Server 2.4.49 - Path Traversal & Remote Code Execution (RCE) – by Lucas Souza – Credits to Ash Daulton and the cPanel Security Team | May 10, 2021

[11] Five Ways to Simulate Apache CVE-2021-41773 Exploits – by Suleyman Ozarslan, PhD & Picus Labs | October 06, 2021

[12] Apache HTTP Server Path Traversal & Remote Code Execution (CVE-2021-41773 & CVE-2021-42013) – by Mayank Deshmukh, Senior Web Application Signatures Engineer – October 27, 2021 - 8 min read

[13] YouTube video for Apache HTTP Server Path Traversal and Remote Code Execution vulnerability (CVE-2021-41773 & CVE-2021-42013) – by vulnmachines

[14] CVE-2021-41773 – Path traversal and file disclosure vulnerability in Apache HTTP Server 2.4.49 – by LudovicPatho

[15] XML external entity (XXE) injection – by Portswigger

[16] Server-side request forgery (SSRF) – by Portswigger

[17] Burp Suite – Application Security Testing Software – by PortSwigger

[18] PayloadAllTheThings GitHub repository for XML External Entity – by swisskyrepo

[19] JWT attacks – by Portswigger

[20] Server-side template injection – by Portswigger

[21] PayloadAllTheThings GitHub repository for Server-Side Template Injection – by swisskyrepo

[22] HTTP request smuggling – by Portswigger

[23] Hash-based Message Authentication Code (HMAC) – by Rahul Awati on TechTarger security

[24] Haproxy : Security Vulnerabilities, CVEs – by CVEdetails

[25] Vulnerability Details: CVE-2021-40346 – by CVEdetails

[26] CVE-2021-40346-POC GitHub repository for CVE-2021-40346 proof of concept – by donky16

[27] Critical Vulnerability in HAProxy (CVE-2021-40346): Integer Overflow Enables HTTP Smuggling – by JFrog