**Second-Order IDOR (Blackbox)**

Now that we have seen how to approach second-order IDOR vulnerabilities from a whitebox approach, let us discuss differences and additional challenges we need to overcome if we do not have access to the web application's source code and need to identify second-order IDORs from a black box approach.

**Identifying Object References**

For this section, our sample web application is a slightly modified version of the lab from the previous section. However, we do not have access to the web application's source code this time. Therefore, we need to identify an object reference for a potential IDOR by exploring the web application.

When accessing one of our files, we can observe that the file GET parameter in the URL looks like a hash:

A screenshot of a computer

Description automatically generated

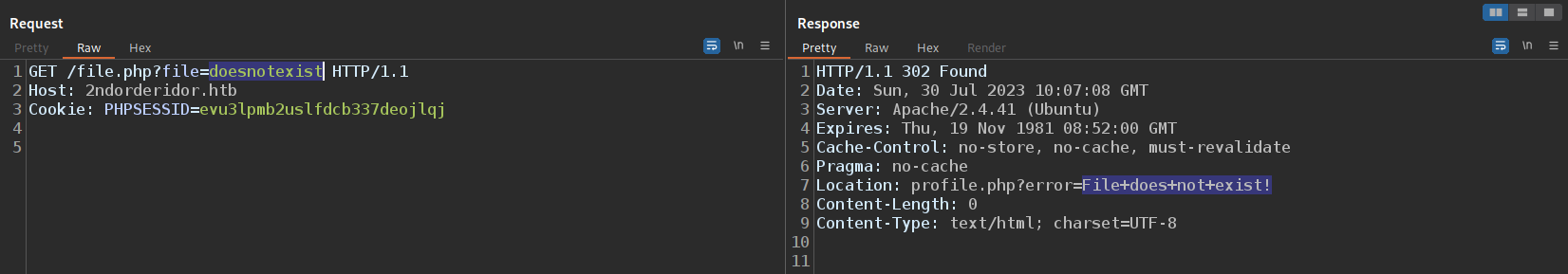
Moreover, we can observe that there is a file preview in our profile that displays the first few characters of the file we last accessed:

A screenshot of a computer

Description automatically generated

To enumerate files, we must apply the methodology discussed in the [Bypassing Encoded References](https://academy.hackthebox.com/module/134/section/1187) section of the Web Attacks module. More specifically, we need to determine how the hash is computed. Some internet research should reveal that the above hash is the MD5 hash of the value 2. Thus, we can create a small script that iterates through all values in a particular range and attempts to access the corresponding files.

First, let us explore how the web application reacts if we attempt to access a file that does not exist:



The error message File does not exist! is subsequently displayed on the profile page. With this information, we can write a script that detects valid file IDs. An example script may look like this:

Code: python

import hashlib, requests

URL = "http://172.17.0.2/file.php"

COOKIE = {"PHPSESSID": "evu3lpmb2uslfdcb337deojlqj"}

for file\_id in range(1000):

id\_hash = hashlib.md5(str(file\_id).encode()).hexdigest()

r = requests.get(URL, params={"file": id\_hash}, cookies=COOKIE)

if not "File does not exist!" in r.text:

print(f"Found file with id: {file\_id} -> {id\_hash}")

Running the script, we can see the discovered file IDs:

yovecio@htb[/htb]$ python3 discover\_fileids.py

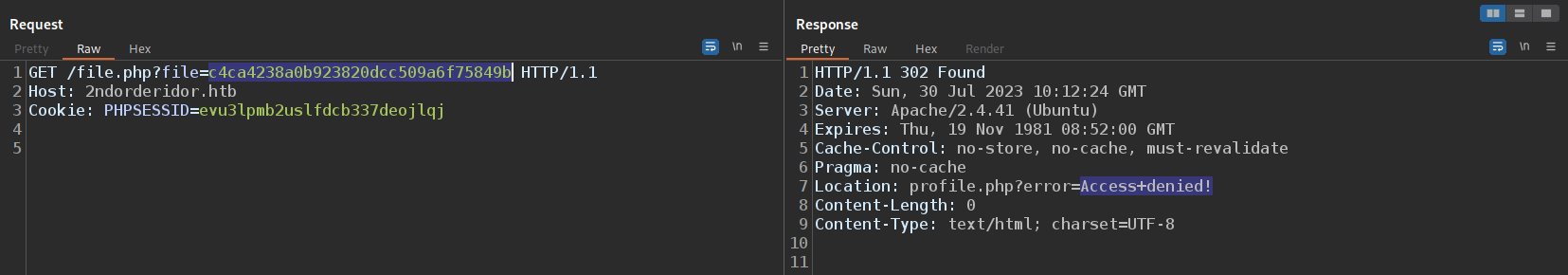
Found file with id: 1 -> c4ca4238a0b923820dcc509a6f75849b

Found file with id: 2 -> c81e728d9d4c2f636f067f89cc14862c

Found file with id: 3 -> eccbc87e4b5ce2fe28308fd9f2a7baf3

Found file with id: 4 -> a87ff679a2f3e71d9181a67b7542122c

From the previous enumeration of our files, we know that the files with file IDs 2, 3, and 4 are ours. With that in mind, let us attempt to access file ID 1. Unfortunately, doing so reveals that the web application implements an authorization check that prevents us from accessing the file owned by another user:



**Exploiting the Second-Order**

To exploit the second-order, we need to think about other functions in the web application that may be affected by our failed file access. In our sample web application, the file is loaded into the recently accessed database such that the first few characters of the file are displayed in our profile, even though the file is owned by another user, as there is no additional authorization check:

A screenshot of a computer

Description automatically generated

While the sample web application is small enough that it is almost impossible not to "accidentally" discover the second-order IDOR vulnerability, real-world web applications tend to be significantly more complex, with multiple features that affect each other.

Therefore, discovering second-order IDOR vulnerabilities in real-world web applications is typically quite challenging and requires a good understanding of how they work, in addition to thinking about how different web application functions might interplay and affect each other to intentionally provoke a second-order IDOR vulnerability.