Dallin Christensen
CS 465

Buffer Overflow 2 (Lab 9)

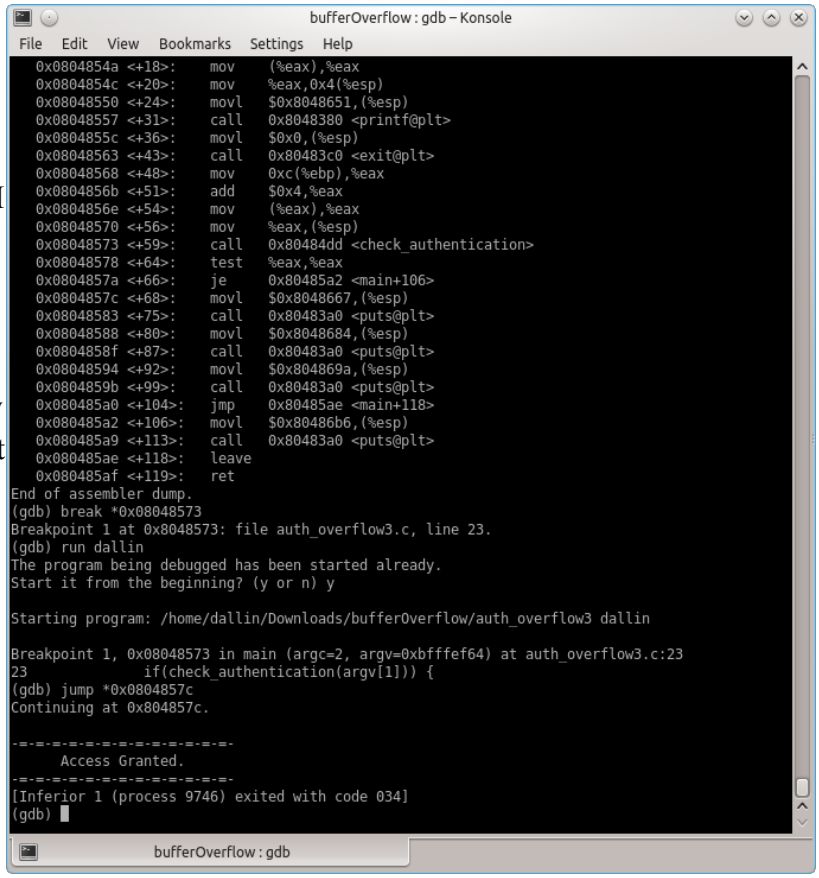
For this lab I chose option 3 (Stack Smashing Intro based on materials found in Jon Erickson's The Art of Exploitation).

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).

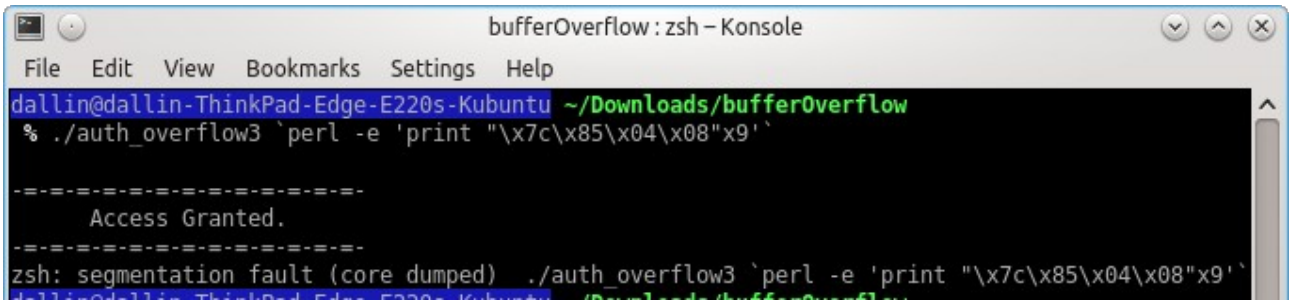
Part 2 - Goal: Use the debugger to obtain access by overwriting the return address

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.



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.