Thomas Hart

CS 465

Project 9 – Buffer Overflow Attack

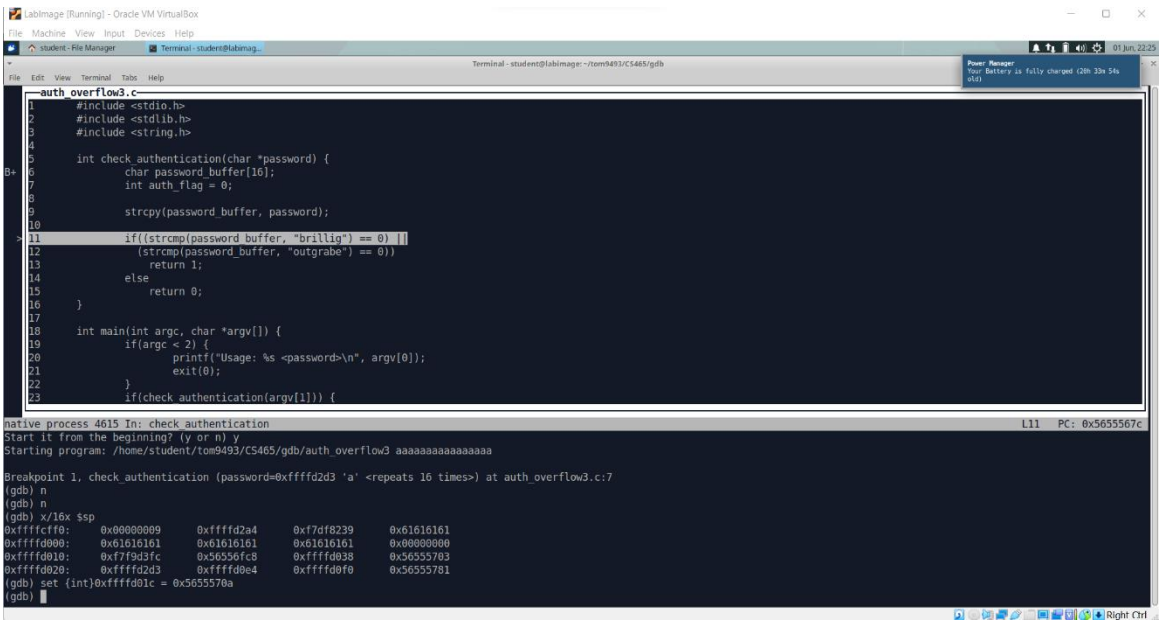
Section A

Local variables
Ebp
Esp
Return address

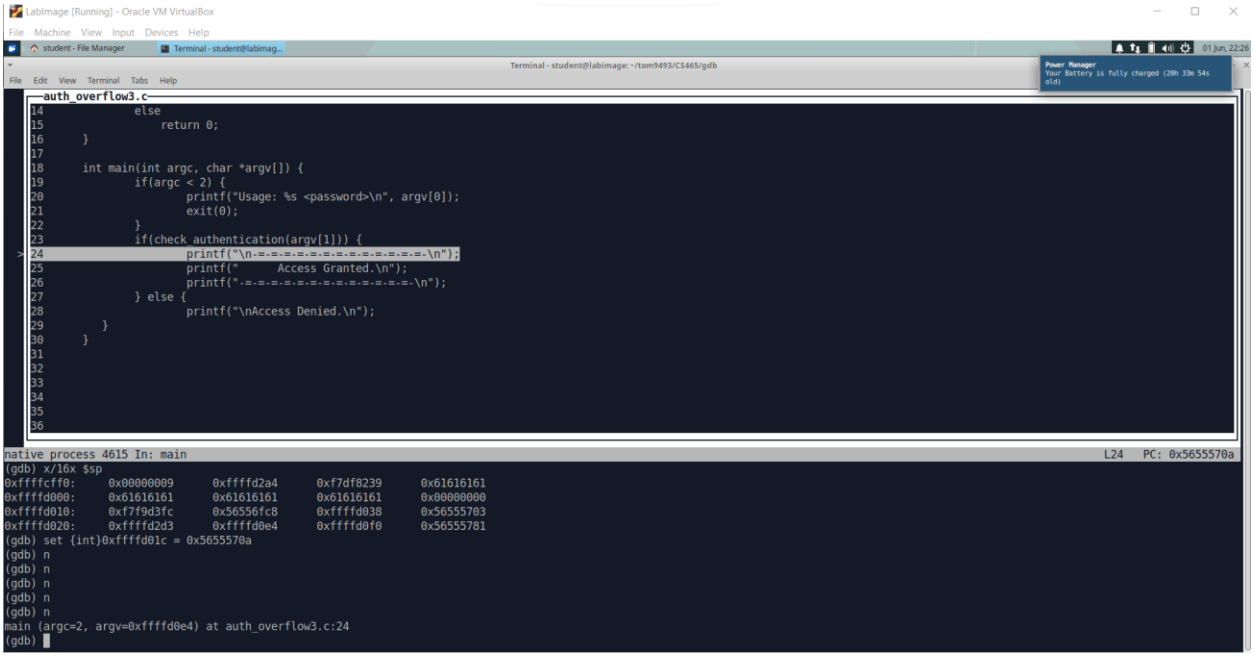
```
0xffffcff0:     0xffffd00c      0xffffd2d4      0xf7df8239      0xf7fa0808
0xffffd000:     0xf7f9d000      0xf7f9d000      0x00000000      0x61616161
0xffffd010:     0x61616161      0x61616161      0x00616161      0x00000000
0xffffd020:     0x00000002      0xffffd0e4      0xffffd048      0x0804854c
```

This is the portion of the stack relevant to the auth_overflow1 program. The repeating 61's show where the "password_buffer" variable begins and ends. As shown, it's only meant to be 16 characters, but because strcpy() does not stop us from overflowing the buffer, we can insert more. The very next address <0xffffd9ec> holds the "auth_flag" variable. If this value returns anything except 0, the program will think a valid password was entered. When we overflow the buffer, we can overwrite this. With these things in mind, entering 16 or fewer characters which are not the given passwords will not give us access. Entering more than this (as long as they're not all zero) will change the "auth_flag" variable from zero, granting us access. If we type 29 characters, it will further overflow the buffer and begin to mess things up. The program will try to access things not accessible, and a segmentation fault will occur after we receive the "Access Granted" message. If we type 33 or more characters, we will immediately get a segmentation fault. Our program will try to access something illegal and break as soon as the input is overflowed onto the stack.

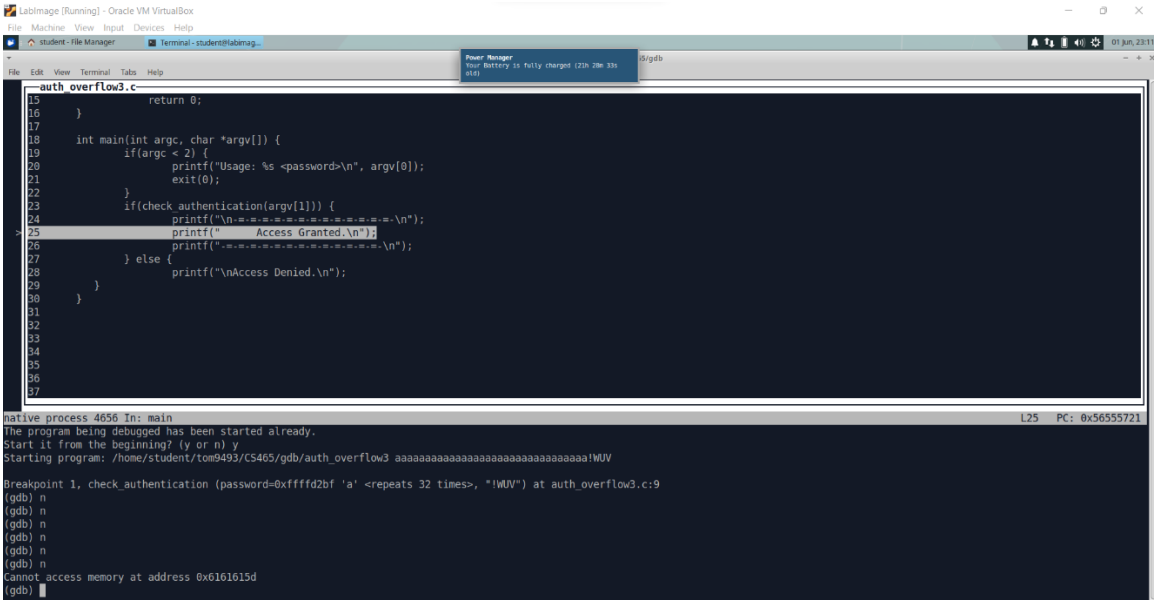
Section B



This first picture is a screenshot of the “check_authentication” stack after entering “aaaaaaaaaaaaaaa” as the password, which should not grant us access. I used “disassemble main” to find the address of the first print statement, which will be the place in code we want to get to since that’s where we’ve technically been granted access. I used “info frame” in the “check_authentication” function to find the return address of the function (held at address 0xffffd01c). Originally, it goes back to the conditional statement to check if the function returned one or zero. That’s at the address 0x56555703. After inspecting main, the address I want to return to is 0x5655570a. I used the “set” command (as seen in the screenshot) to change the return address. The screenshot below shows that it worked, and I was able to gain access without a valid password.



Section C



Section D

We use the strcpy() function in “check_authentication” to overflow the buffer with NOP commands, followed by the shell code, followed by padding before we reach the return address, then we overwrite the return address with a new address that will point somewhere inside the NOP commands. The NOP commands act as a window that we can return our program to without having to know the exact address of the shell code. All the NOP commands will execute (doing nothing) and then the shell code will execute. With this particular stack, however, I couldn’t figure out how to do the injection with the tiny window given between the top of the stack (esp) and bottom of the stack (ebp). Since the return address follows directly after ebp, the stack needs to be at least as big as the shell code, but this stack was not. From where the buffer begins and the return address (0xffffd00c to 0xffffd02c), the shell code does not fit. If I could increase the size of the buffer, this would be manageable. Maybe there was a different way to do this without changing auth_overflow3.c, but I couldn’t figure it out.

```
native process 4853 In: check_authentication
(gdb) x/30x $esp
0xffffd000:     0x00000009      0xffffd2b0      0xf7df8239      0xf7fa0808
0xffffd010:     0xf7f9d000      0xf7f9d000      0x00000000      0x00000000
0xffffd020:     0xf7f9d3fc      0x56556fc8      0xffffd048      0x56555703
0xffffd030:     0xffffd2df      0xffffd0f4      0xffffd100      0x56555781
0xffffd040:     0xf7fe5960      0xffffd060      0x00000000      0xf7de0fa1
0xffffd050:     0xf7f9d000      0xf7f9d000      0x00000000      0xf7de0fa1
0xffffd060:     0x00000002      0xffffd0f4      0xffffd100      0xffffd084
0xffffd070:     0x00000002      0xffffd0f4
(gdb)
```