

CS Project

Return-to-libc Attack

Păștin Melisa-Alexia

CTI-EN, 3rd year

0. Introduction

Buffer overflow vulnerabilities remain one of the most significant security risks in software development, allowing attackers to manipulate program execution and potentially gain unauthorized access to system resources. To mitigate such threats, modern operating systems have introduced several countermeasures, including non-executable stacks, StackGuard, and address space layout randomization (ASLR). However, attackers have developed techniques to bypass these protections, such as the Return-to-libc attack.

This project explores the mechanics of the Return-to-libc attack, an advanced exploitation method that does not rely on injecting shellcode, but instead hijacks the control flow of a vulnerable program to execute existing functions within the C standard library (libc). Specifically, the attack manipulates the stack to invoke the `system()` function, enabling execution of arbitrary commands, typically to escalate privileges and spawn a root shell.

To complete this project, I was given a vulnerable Set-UID program that contained a classic buffer overflow flaw. My task was to craft a malicious input that used the Return-to-libc technique to gain root access. I also worked on exercises that involved understanding and disabling protection mechanisms such as StackGuard, non-executable stacks, and ASLR. This hands-on experience gave me a comprehensive view of both how attacks are carried out and how they can be defended against.

1. Preparation tasks

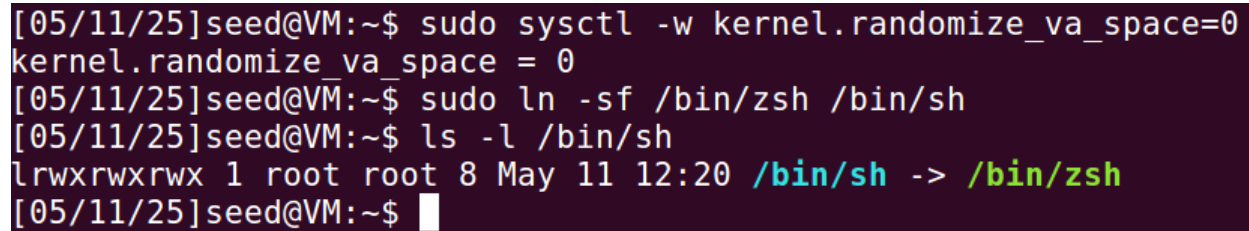
This section documents the steps taken to configure the environment for the Return-to-libc attack. Key security mechanisms were disabled to simplify the exploitation process, and the vulnerable program was compiled with specific settings. Screenshots are included to verify critical steps.

Ubuntu enables ASLR by default to randomize memory addresses such as those of the stack and heap. This makes it harder to predict addresses, which is critical in buffer overflow attacks. To disable ASLR, I ran the following command in the terminal:

```
sudo sysctl -w kernel.randomize_va_space=0
```

In Ubuntu 16.04, `/bin/sh` is linked to `/bin/dash`, which includes a security feature that disables Set-UID privileges when executed in such a process. Since our exploit relies on invoking `/bin/sh` with elevated privileges via `system()`, I had to re-link it to a shell without that countermeasure. I used the following command:

```
sudo ln -sf /bin/zsh /bin/sh
```



```
[05/11/25]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[05/11/25]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[05/11/25]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 May 11 12:20 /bin/sh -> /bin/zsh
[05/11/25]seed@VM:~$
```

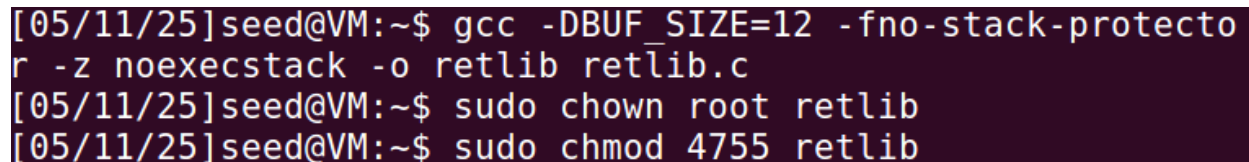
Screenshot 1 (the third command in the screenshot was used to check the correct execution of the second command)

Next, I compiled the vulnerable program `retlib.c`. The program contains a classic buffer overflow vulnerability: it reads 300 bytes into a much smaller buffer (`BUF_SIZE`), which causes an overflow.

```
gcc -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
```

Then, I changed the ownership of the binary to root and set the Set-UID bit so that it runs with root privileges:

```
sudo chown root retlib
sudo chmod 4755 retlib
```



```
[05/11/25]seed@VM:~$ gcc -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
[05/11/25]seed@VM:~$ sudo chown root retlib
[05/11/25]seed@VM:~$ sudo chmod 4755 retlib
```

Screenshot 2

2. Task 1: Finding out the addresses of libc functions

The goal of this task was to determine the runtime memory addresses of two key libc functions: `system()` and `exit()`. These addresses are needed for a later buffer overflow exploit. When Address Space Layout Randomization (ASLR) is disabled, shared libraries such as libc load at fixed addresses, making it feasible to discover function addresses using debugging tools like gdb.

Firstly, I created an empty badfile, required for execution using the command:

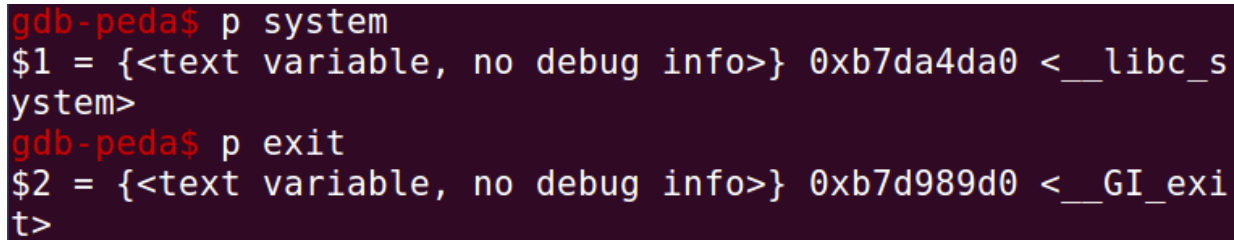
```
touch badfile
```

Then, the following steps were performed to retrieve the memory addresses of `system()` and `exit()` using gdb:

```
gdb -q retlib
(gdb) run
(gdb) p system
```

```
(gdb) p exit
```

```
(gdb) quit
```



```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7da4da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7d989d0 <__GI_exit>
```

Screenshot 3

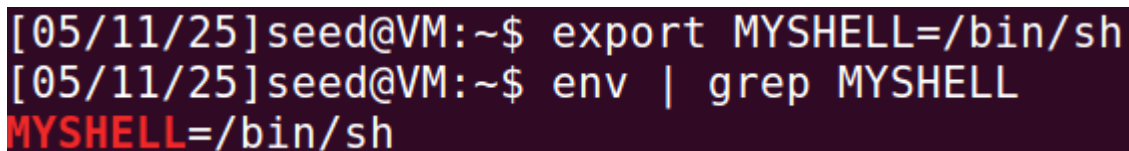
So I found out that:

- The memory address of system() is: 0xb7da4da0
- The memory address of exit() is: 0xb7d989d0

3. Task 2: Putting the shell string in the memory

The goal of this task was to perform a buffer overflow attack and redirect the program's control flow to the system() function, executing an arbitrary command. Specifically, I tried to get the system() function to execute the /bin/sh program, which would provide us with a shell prompt. To achieve this, we needed to place the string /bin/sh into memory and pass its address as an argument to the system() function.

I chose to use an environment variable to hold the /bin/sh string. Since environment variables are inherited by child processes, defining one in the current shell ensures that it appears in the memory of the vulnerable program when it runs. I exported a new environment variable named MY_SHELL and assigned it the value /bin/sh. To verify that the string was correctly placed in memory, I used the env command to inspect the variable.



```
[05/11/25]seed@VM:~$ export MY_SHELL=/bin/sh
[05/11/25]seed@VM:~$ env | grep MY_SHELL
MY_SHELL=/bin/sh
```

Screenshot 4

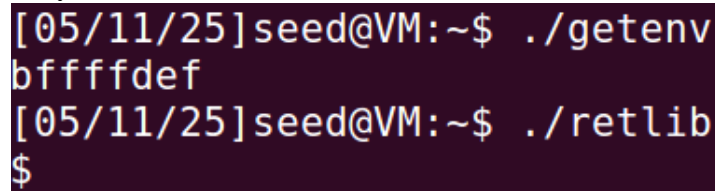
Using a small C program (getenv.c), I retrieved and printed the address of the MY_SHELL environment variable. This gives me the memory address of the string /bin/sh, which is needed to pass as an argument to system().

The address printed by the program will typically be consistent if Address Space Layout Randomization (ASLR) is disabled. However, the address may vary slightly when executed within the vulnerable program (retlib) due to changes in the program's environment, such as the program's name or other factors. Although the address could change slightly, it will usually be close to the address printed by the simple C program.

I ran the vulnerable program `retlib` with the goal of causing a buffer overflow. By using the address of the `MYHELL` variable as an argument to the `system()` function, I intended to execute `/bin/sh` and spawn a shell.

After compiling `retlib.c` and ensuring the Set-UID program was correctly set up, I ran the program to trigger the buffer overflow.

After executing `retlib`, the attack was successful and I gained access to a shell prompt (`/bin/sh`), indicating that the `system()` function was called with the `/bin/sh` command.



```
[05/11/25]seed@VM:~$ ./getenv
bffffdef
[05/11/25]seed@VM:~$ ./retlib
$
```

Screenshot 5

So the address is: `0xbffffdef`

4. Task 3: Exploiting the buffer-overflow vulnerability

The goal of this task was to exploit the buffer overflow vulnerability in the `retlib` program. Specifically, I created a file named `badfile` that contained the addresses of key functions (`/bin/sh`, `system()`, and `exit()`) in order to perform a buffer overflow attack and gain access to a shell.

I generated the contents of `badfile` using C code. This file was crafted to manipulate the return address of the vulnerable function in `retlib`, redirecting it to the `system()` function, which would then execute `/bin/sh` and spawn a shell.

In `exploit.c`, I defined a buffer (`buf`) that stored the relevant function addresses: the address of `system()`, the argument to be passed (`/bin/sh`), and finally the address of `exit()` to ensure a clean program termination. I overwrote the return address in the vulnerable function with the address of `system()`, and placed the address of `/bin/sh` at the appropriate location to be used as its argument. Lastly, I wrote this constructed buffer into `badfile`.

The provided `exploit.c` had some missing parts that I needed to complete. One of these was filling in the correct function addresses. I used the values I had obtained earlier during Tasks 1 and 2: the addresses of `system()` and `exit()` from Task 1, and the `/bin/sh` address from Task 2. For reference, the addresses I used were:

`/bin/sh: 0xbffffdef`

`system(): 0xb7da4da0`

`exit(): 0xb7d989d0`

The other one refers to the values of `X`, `Y` and `Z`. To successfully craft the exploit in this lab, it was essential to determine the correct offset values, labeled as `X`, `Y`, and `Z`, used to overwrite specific locations on the stack. By disassembling the vulnerable `bof()` function, we observed the typical function prologue: `push ebp, mov ebp, esp`, followed by `sub esp, 0x18`. The `sub esp, 0x18` instruction reserves 24 bytes on the stack for local variables, including the 40-byte character

buffer. Since the return address is stored just above this buffer, overflowing the buffer with 24 bytes reaches the saved return address. Therefore, we set $X = 24$ to overwrite the return address with the address of `system()`. On a 32-bit system, each address occupies 4 bytes, so the subsequent two values, Y and Z , are placed at offsets 28 and 32 respectively. In our modified code, these values are explicitly written as: `buf[24] = system()`, `buf[28] = exit()`, and `buf[32] = "/bin/sh"`.

```
Dump of assembler code for function bof:
0x080484bb <+0>:      push    ebp
0x080484bc <+1>:      mov     ebp,esp
0x080484be <+3>:      sub     esp,0x18
```

Screenshot 6

In a return-to-libc attack, the order of values placed on the stack is critical to ensure correct execution. The first value to overwrite the return address must be the address of `system()`, which hijacks control flow to execute a shell. Immediately after that, the address of `exit()` is placed to ensure the program terminates cleanly once the shell exits. Finally, the address of the string `"/bin/sh"` is added last, as it will be used as the first argument to `system()`. This specific ordering — `system()`, `exit()`, and then `"/bin/sh"` — ensures the exploit mimics a valid function call sequence. In our modified code, we explicitly assign these addresses in the proper stack order to form a reliable and effective payload for the return-to-libc attack.

Once I completed the `exploit.c` program, I compiled it to generate the badfile. This created an executable named `exploit`, which, when executed, produced the badfile containing the required addresses. After running the `exploit` executable, I then executed the `retlib` program. Due to the buffer overflow vulnerability, `retlib` read the crafted input from badfile and redirected its control flow to `system("/bin/sh")`, effectively granting me a shell.

```
[05/11/25]seed@VM:~$ gcc -o exploit exploit.c
[05/11/25]seed@VM:~$ ./exploit
[05/11/25]seed@VM:~$ ./retlib
$
```

Screenshot 7

After that, I proceeded with the two attack variations:

Attack variation 1: In this variation, the goal is to determine whether including the address of the `exit()` function in the badfile is necessary for the exploit to work. I tested this by commenting out the line that stored the address of `exit()` in the buffer. After making this change, I recompiled and executed the `exploit` program, followed by running `retlib` as before.

After running the attack without including the address of `exit()`, I successfully obtained a shell. This indicated that the `exit()` function was not strictly necessary for the exploit to succeed.

Although `exit()` is typically used to cleanly terminate a program after execution, in this case, the exploit worked without it. This suggested that the program could return from `system("/bin/sh")` and continue execution without any issues, even though `exit()` was not explicitly called.

In typical buffer overflow exploits, `exit()` is often included to ensure the program terminates gracefully after spawning the shell. It prevents the program from executing unintended instructions or leaving background processes in an inconsistent state.

However, in my case, the attack still worked without the `exit()` address, which suggested that after executing `system("/bin/sh")`, the process either terminated naturally or continued in a way that didn't cause a crash. This may have happened because the system call to `system()` handled process control well enough to allow the exploit to succeed and the shell to be spawned.

Since `system("/bin/sh")` launches a new shell, and that shell takes control, the program may not have needed `exit()` to continue running safely. Therefore, although the exploit succeeded without it, including `exit()` would still be good practice to ensure clean and predictable program termination.

Attack variation 2: This variation explored what would happen when I changed the filename of `retlib` to something different (e.g., `newretlib`), while keeping the content of `badfile` unchanged. My goal was to observe whether the attack would still succeed under this change.

In this variation, the attack failed. The failure appeared to be due to how memory addresses—such as the location of environment variables—are influenced by factors like the program's name and the length of the filename.

In particular, the address of the environment variable (like `MYSHELL`) that holds the string `/bin/sh` likely changed when I renamed `retlib` to `newretlib`. The memory layout of the process shifted due to factors such as the filename length, which affects the placement of environment variables in memory; changing the filename's length can cause these variables to be located at different memory addresses.

Since the exploit depended on hardcoded memory addresses—such as the location of `MYSHELL`—renaming the executable altered those addresses. As a result, the original `badfile`, which relied on specific memory addresses, no longer worked because the memory layout had shifted.

This experiment showed that the program's memory layout, including the position of key variables like `MYSHELL`, was influenced by the program's name. Changing the filename disrupted the address assumptions in my exploit, causing it to fail.

5. Task 4: Turning on address randomization

In this task, I investigated the impact of address randomization on the success of a Return-to-libc attack. To begin, I enabled address randomization on Ubuntu using the following command:

```
sudo sysctl -w kernel.randomize_va_space=2
```

I then re-executed the Return-to-libc attack using the previously constructed `badfile`. However, this time, the `retlib` program failed with a segmentation fault, and the shell was not spawned. This indicated that the program attempted to jump to or access an invalid memory location.

The corresponding values in the `badfile` specifically, the addresses of `system()`, `exit()`, and `/bin/sh` were based on static memory locations observed when ASLR was disabled. When ASLR was

enabled, these addresses became randomized, meaning the hardcoded values in the badfile no longer pointed to valid memory locations. However, the offsets (X=24, Y=28, Z=32) remained correct because they depend on the stack layout of the vulnerable program, which is fixed at compile time and unaffected by ASLR.

The failure of the exploit occurred because ASLR randomized the addresses of `system()`, `exit()`, and `/bin/sh`, not the offsets themselves. The values written at offsets X, Y, and Z became invalid, but the positions (X=24, Y=28, Z=32) were still the correct locations to overwrite the return address, return address for `system()`, and argument for `system()`, respectively. This distinction highlights that ASLR disrupts the predictability of memory addresses, not the structure of the stack frame, rendering the exploit ineffective.

```
[05/11/25]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[05/11/25]seed@VM:~$ gcc -o exploit exploit.c
[05/11/25]seed@VM:~$ ./exploit
[05/11/25]seed@VM:~$ ./retlib
Segmentation fault
```

Screenshot 8

To confirm this, I used `gdb` to inspect the randomized addresses:

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb755ada0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb754e9d0 <__GI_exit>
```

Screenshot 9

My hypothesis is that the exploit failed because ASLR caused the runtime addresses of `system()`, `exit()`, and `"/bin/sh"` to change unpredictably with each execution. Since these addresses were hardcoded in the badfile, the values for X, Y, and Z, which relied on these addresses, no longer matched the actual memory locations during execution, causing the program to crash. ASLR's address randomization protection successfully prevented the Return-to-libc attack by altering the memory locations of critical functions and strings. The exploit, which relied on fixed memory addresses, could no longer succeed because these addresses no longer aligned due to ASLR, highlighting its effectiveness in protecting against Return-to-libc attacks.

6. Conclusions

This lab project gave me a hands-on experience in exploring buffer overflow vulnerabilities and Return-to-libc attacks. It allowed me to dive into both offensive techniques and modern defensive mechanisms, building a solid understanding of these security concepts. The tasks were

structured to gradually increase in complexity, helping me to gain a deeper understanding of real-world security challenges. Below are my key takeaways from each stage of the lab:

Task 1: Finding out the addresses of libc functions

I began by analyzing the behavior of a vulnerable program to understand how the stack layout changes during execution. Using tools like gdb, I examined the addresses of environment variables and function calls. This gave me valuable insight into how functions return, how memory is laid out on the stack, and how an attacker could potentially manipulate this for exploit.

Task 2: Putting the shell string in the memory

In this phase, I constructed a malicious input file (badfile) to exploit the buffer overflow vulnerability. By carefully placing the addresses of `system()`, `exit()`, and the `"/bin/sh"` string at precise offsets (X, Y, and Z), I successfully hijacked the program's execution flow. This demonstrated to me how attackers can redirect program execution without needing to inject shellcode, relying only on existing libc functionality.

Task 3: Exploiting the buffer-overflow vulnerability

After compiling and executing my exploit code, I successfully launched a Return-to-libc attack. The vulnerable program spawned a root shell, confirming that my payload was correctly constructed. I also tested two variations of the attack:

- Without `exit()`: The attack still succeeded, showing that while including `exit()` can help gracefully terminate the program, it's not strictly necessary.
- Changing the executable's filename: The attack failed, demonstrating how small changes like renaming the executable can invalidate the exploit by shifting memory addresses, emphasizing the fragility of such attacks.

Task 4: Turning on address randomization

In the final task, I enabled Ubuntu's address space layout randomization (ASLR) with `kernel.randomize_va_space=2`. When I re-ran the exploit, it resulted in a segmentation fault, and the attack failed. This showed me that ASLR is an effective defense against Return-to-libc attacks by randomizing the memory addresses of key components. This makes it nearly impossible to predict function locations without additional vulnerabilities, such as information leaks.

Through this lab, I learned that Return-to-libc attacks can bypass defenses like non-executable stacks but rely on precise control over memory layout. Even small changes, such as renaming the executable or shifting the stack, can break the exploit, showing the vulnerability of such attacks. Modern protections like ASLR are highly effective at disrupting these attacks, underscoring the importance of enabling them in real systems. Ultimately, I gained a solid understanding of system internals, such as the stack and libc, which is crucial both for creating and defending against such exploits.