IERG 4130 - Introduction to Cyber Security (Fall 2022)
Lab 1: Buffer-Overflow & Format String Vulnerability

Total: 100' (with 10' bonus)
Due Date: Oct. 27, 11:59 pm
Reference: https://seedsecuritylabs.org/
Note: The following tasks are adjusted and may be different from the original SEED labs.
You can download the used code files in Blackboard.
The BONUS won't let you exceed the full mark but can help make up the deductions.

1. Set-UID Program and Linux Capability (10')

1.1 Overview

In this lab, we are particularly interested in how environment variables affect the behavior of Set-UID programs, which are usually privileged programs. In a capability system, when a program is executed, its corresponding process is initialized with a list of capabilities (tokens). When the process tries to access an object, the operating system checks the process' capabilities, and decides whether to grant the access or not.

1.2 Lab Tasks

1.2.1 Task 1: Environment Variable and Set-UID Programs

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 the 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. Write following program that can print out all the environment variables in current process.

```
#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.

```
// Asssume the program's name is foo
$ sudo chown root foo
$ sudo chmod 4755 foo
```

Step 3. In your 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 already exist):

- `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 your 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.

1.2.2 Task 2: Experiencing Capabilities

In operating systems, there are many privileged operations that can only be conducted by privileged users. Examples of privileged operations include configuring network interface cards, backing up all the user files, shutting down the computers, etc. Without capabilities, these operations can only be carried out by superusers, who often have many more privileges than what are needed for the intended tasks. Therefore, letting superusers conduct these privileged operations is a violation of the Least-Privilege Principle.

Privileged operations are very necessary for operating systems. All Set-UID programs involve privileged operations that cannot be performed by normal users. To allow normal users to run these programs, Set-UID programs turn normal users into powerful users (e.g. root) temporarily, even though the involved privileged operations do not need all the power. This is dangerous: if the program is compromised, adversaries might get the root privilege.

Capabilities divide the powerful root privilege into a set of less powerful privileges. Each of these privileges is called a capability. With capabilities, we do not need to be a superuser to conduct privileged operations. All we need is to have the capabilities that are needed for privileged operations. Therefore, even if a privileged program is compromised, adversaries can only get limited power. This way, the risk of the privileged program can be lowered quite significantly.

When a privileged program is executed, the running process will carry those capabilities that are assigned to the program. In some sense, this is similar to the Set-UID files, but the major difference is the amount of privileged carried by the running processes.

Step 1. We will use an example to show how capabilities can be used to remove unnecessary power assigned to certain privileged programs. First, let us log in as a normal user, run the following command and describe your observation:

```
$ ping www.cuhk.edu.hk
```

The program should run successfully. If you look at the file attribute of the program `/bin/ping`, you will find out that ping is actually a Set-UID program with the owner being root, i.e., when you execute `ping`, your effective user id becomes root, and the running process is very powerful. If there are vulnerabilities in ping, the entire system can be compromised. The question is whether we can remove these privileged from `ping`. Let us turn `/bin/ping` into a non-Set-UID program. This can be done via the following command:

```
$ sudo chmod u-s /bin/ping
```

Note: Binary files like ping may locate in different places in different distributions of Linux, use 'which ping' to locate your ping program.
Now, run `'ping www.cuhk.edu.hk'`, and see what happens. Interestingly, the command will not work. This is because ping needs to open RAW socket, which is a privileged operation that can only be conducted by root (before capabilities are implemented). That is why ping has to be a Set-UID program. With capability, we do not need to give too much power to ping.

Step 2. Let us only assign the cap net raw capability to ping and use `getcap` to display the capabilities, and see what happens (describe your observation):

```
$ sudo setcap cap_net_raw=ep /bin/ping
$ getcap /bin/ping
$ ping www.cuhk.edu.hk
```

Step 3. We can also use `setcap -r` to remove the capabilities. Let us remove the capability and see what happens (describe your observation):

```
$ sudo setcap -r /bin/ping
$ ping www.cuhk.edu.hk
```

2. Buffer-Overflow Vulnerability (50')

2.1 Overview

The learning objective of this lab is for students to gain 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 used by a malicious user to alter the flow control of the program, leading to the execution of malicious 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 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 buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

2.2 Lab Tasks

2.2.1 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. Later on, we will enable them one by one, and see whether our attack can still be successful.

Address Space Randomization. Ubuntu and several other Linux-based systems use 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. 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 will 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 -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
```

Configuring /bin/sh (Ubuntu 16.04 VM only). 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 rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

2.2.2 Task 3: Running Shellcode

Before starting the attack, let us get familiar with the 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 how to launch a shell by executing a shellcode stored in a buffer. Please compile and run the following code, and see whether a shell is invoked. You can download the program from Blackboard.

```
/* call_shellcode.c */
/* You can get this program from the lab's website */
/* A program that launches a shell using shellcode */
#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)( );
}
```

Compile the code above using the following `gcc` command. Run the program and describe your observations. Please do not forget to use the `execstack` option, which allows code to be executed from the stack; without this option, the program will fail.

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

The shellcode above invokes the `execve()` system call to execute `/bin/sh`. A few places in this shellcode are worth mentioning. 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. 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. Third, the system call `execve()` is called when we set `%al` to 11, and execute "`int $0x80`".

2.2.3 The Vulnerable Program

You will be provided with the following program, which has a buffer-overflow vulnerability in Line ①. Your job is to exploit this vulnerability and gain the root privilege.

```
/* Vunlerable program: stack.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
   char buffer[128];

   /* The following statement has a buffer overflow problem */
   strcpy(buffer, str); ①

   return 42;
}

int main(int argc, char **argv)
{
   char str[404];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   fread(str, sizeof(char), 404, badfile);
   bof(str);
   printf("Returned Properly\n");
   return 1;
}
```

Compile the above vulnerable program. Do not forget to include the -fno-stack-protector and "`-z execstack`" options to turn off the StackGuard and the non-executable stack protections. After the compilation, we need to make the program a root-owned Set-UID program. We can achieve this by first changing the ownership of the program to root (Line ①), and then changing 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 -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack ①
$ sudo chmod 4755 stack ②
```

The above program has a buffer overflow vulnerability. It first reads 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 500 bytes, but the buffer in `bof()` is only 24 bytes long. Because `strcpy()` does not check boundaries, a 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 the 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.

2.2.4 Task 4: Exploiting the Vulnerability

We provide you with a partially completed exploit code called "`exploit.c`". The goal of this code is to construct contents for `badfile`. In this code, 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" /* 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 */
;

void main(int argc, char **argv)
{
   char buffer[404];
   FILE *badfile;

   /* Initialize buffer with 0x90 (NOP instruction) */
   memset(&buffer, 0x90, 404);
```

```
    /* You need to fill the buffer with appropriate contents
       here */
    /* ... Put your code here ... */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 404, 1, badfile);
    fclose(badfile);
}
```

After you finish 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:

Important: Please compile your vulnerable program first. Please note that the program exploit.c, which generates the `badfile`, can be compiled with the default StackGuard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in `stack.c`, which is compiled with the StackGuard protection disabled.

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

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)
```

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 Blackboard as well.

2.2.5 Task 5: Defeating dash's Countermeasure

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 ①, which compares real and effective user/group IDs.

```
//
https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.dif
f.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())) { ①
++         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 rm /bin/sh
$ sudo ln -s /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 ① and run the program as a Set-UID program (the owner should be root); please describe your observations. We then uncomment Line ① 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); ①
   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 one in Task 4 ---
   "\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 attack from Task 4 again and see if you can get a root shell. Please describe and explain your results.

2.2.6 Task 6: Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524, 288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. First, we turn on Ubuntu's address randomization using the following command. We run the same attack developed in Task 4. Please describe and explain your observation.

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

We then use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the `badfile` can eventually be correct. You can use the following shell script to run the vulnerable program in an infinite loop. If your attack succeeds, the script will stop; otherwise, it will keep running. Please be patient, as this may take a while. Let it run overnight if needed. Please describe your observation.

```
#!/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
```

2.2.7 Task 7: Turn on the StackGuard Protection

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we disabled the StackGuard protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 4 in the presence of StackGuard. To do that, you should compile the program without the *-fno-stack-protector* option. For this task, you will recompile the vulnerable program, `stack.c`, to use GCC StackGuard, execute task 4 again, and report your observations. You may report any error messages you observe.

In GCC version 4.3.3 and above, StackGuard is enabled by default. Therefore, you have to disable StackGuard using the switch mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable StackGuard.

2.2.8 Task 8: Turn on the Non-executable Stack Protection

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the noexecstack option, and repeat the attack in Task 4. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult. You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the nonexecutable 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.

3. Format String Vulnerability (40’ + bonus 10’)

3.1 Overview

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 `prinf()` 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 students to gain the first-hand experience on format-string vulnerability by putting what they have learned about the vulnerability from class into actions. Students will be given a program with a format-string vulnerability; their 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.

3.2 Lab Tasks

To simplify the tasks in this lab, we turn off the address randomization using the following command:

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

3.2.1 The Vulnerable Program

You are given a vulnerable program that has a format string vulnerability. The program has a function `myprintf()`, which takes a user input using `fgets()`, and then prints out the input using `printf()`. The way `printf()` is used (Line ①) is vulnerable to format string attacks.

```
#include <stdio.h>

void myprintf()
{
      char input[100];
      int var = 0x55667788;

      /* print out information for experiment purpose */
      printf("Target address: %x\n", (unsigned) &var);
      printf("Data at var's address: 0x%x\n", var);

      printf("Please enter a string: ");
      fgets(input, sizeof(input)-1, stdin);

      printf(input); //The vulnerable place ①

      printf("Data at target address: 0x%x\n", var);
}

void main() {myprintf();}
```

Compilation. Compile the above program. Moreover, some of our attacks require us to know the memory address of a target area, so for the sake of simplicity, we turn off the system address randomization. We run the following commands:

```
$ gcc -o vul vul.c
$ sudo sysctl -w kernel.randomize_va_space=0
```

Running and testing the program. Yon can type any message; the program is supposed to print out whatever is typed by you. However, a format string vulnerability exists in the program's `myprintf()` function, which allows us to get the program to do more than what it is supposed to do, including giving us a root access to the machine. In the rest of this lab, we are going to exploit this vulnerability

```
$ ./vul
Target address: xxxxxxxx
Data at target address: 0x55667788
Please enter a string: your message here
your message here
Data at target address: 0x55667788
```

3.2.2 Task 9: Understanding the Layout of the Stack

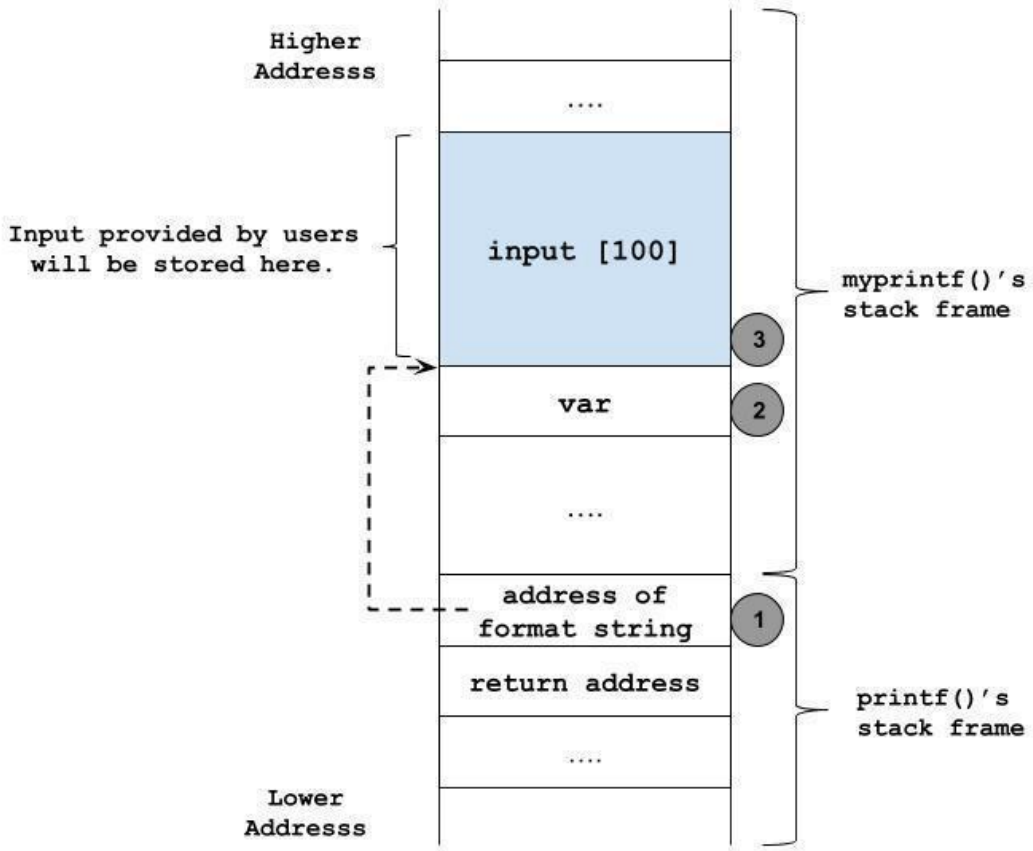

Figure 1: The stack layout when printf() is invoked from inside of the myprintf() function.

To succeed in this lab, it is essential to understand the stack layout when the printf() function is invoked inside myprintf(). Figure 1 depicts the stack layout. You need to conduct some investigation and calculation, and then answer the following questions:

- Question 1: What are the memory addresses at the locations marked by ❶, ❷, and ❸?
- Question 2: What is the distance between the locations marked by ❶ and ❸?

3.2.3 Task 10: Crash the Program

The objective of this task is to provide an input to the `vul` program, such that when the program tries to print out the user input in the `myprintf()` function, it will crash.

```
$ ./vul
Target address: xxxxxx
Data at target address: 0x55667788
Please enter a string: //your input here
Segmentation fault
```

3.2.4 Task 11: Print Out Data on the Stack

Assume that there is a secret value stored inside the program, and we would like to use the format string vulnerability to get the program to print out the secret value. For this task, we assume that the `var` variable in the vulnerable program contains a secret.

```
Target address: xxxxxx
Data at target address: 0x55667788
Please enter a string: //your input here
63.b7fba5a0.f0b5ff.bfffec5e.55667788.252e7825.78252e78.2e78252e.252
e7825
Data at target address: 0x55667788
```

3.2.5 Task 12: Change the Program's Data in the Memory (Bonus 10')

Attention: **The BONUS won't let you exceed the full mark but can help make up the deductions.**

The objective of this task is to modify the vulnerable program's memory using the format string vulnerability. Now we assume that var holds an important number that should not be tampered with by users. Its current value is 0x55667788, and we want to change it to another value.

Try to change the value to `0x22112211`. In this task, we need to change the content of the target variable to a specific value of `0x22112211`. Your task is considered a success only if the variable's value becomes `0x22112211`.

4. Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide an explanation for the observations that are interesting or surprising. Please also list the important code snippets (if any) followed by an explanation. Simply attaching code without any explanation will not receive credits. (**Please also write down your student id in the report**)