Homework #1: Buffer Overflows

Due February 10, 11:00 PM

---

1 Homework Overview

This homework will give you first-hand experience with *buffer overflow attacks*. This attack exploits a buffer overflow vulnerability in a program to make the program bypass its usual execution and instead jump to alternative code (which typically starts a shell). Specifically, the attack overflows the vulnerable buffer to introduce the alternative code on the stack and appropriately modifies the return address on the stack (to point to the alternative code). There are several defenses against this attack (other than fixing the overflow vulnerability), such as address space randomization, compiling with stack-guard, dropping root privileges, etc.

In this lab, you are given a set-root-uid program with a buffer-overflow vulnerability for a buffer allocated on stack. You are also given *shellcode*, i.e., binary code that starts a shell. Your task is to exploit the vulnerability to corrupt the stack so that when the program returns, instead of going to where it was called from, it calls the shellcode, thereby creating a shell with root privilege. You will also be guided through several protection schemes implemented in Ubuntu to counter this attack.

Note: There is a lot of helpful information in Section 7; be sure to read it before you get started. Also, if you get stuck, "Smashing the Stack for Fun and Profit" and the lecture notes and slides will help.

2 Getting Set Up

This homework assignment uses Docker. If you haven't used Docker before, you may want to review Prof. Mike Marsh's tutorial – it has great information about how to install Docker as well as how to use it in general. If you are running Windows, you may find it especially helpful for installation. (Beyond installation, there is more information in the tutorial than you will need just to do this homework.) You can find it here: https://gitlab.cs.umd.edu/mmarsh/docker-tutorial

Once you have Docker installed, you can **use the preconfigured Docker image we have given you**, available here:

https://umd.box.com/s/ldcf370t67tm7e25wm0ihq0nqezok36k.

The username and password are both `cmsc414`.

Once you download the docker image, you will need to load it as:

```
$ docker image load -i /path/to/ubuntu-hw1.tar
```

where `/path/to/` is replaced by the location of `ubuntu-hw1.tar`. This command loads the docker image from the tarball, which you can view using the command `docker images`. You only need to load the image once.

After you download the starter files, you can run your code as follows, assuming your .c files and Makefile are in the current diretory:

```
$ docker run -it -v $(pwd):/opt ubuntu:hw1
cmsc414@622bc485fad2:/$ cd /opt
cmsc414@622bc485fad2:/opt$ make
cmsc414@622bc485fad2:/opt$ ./name_of_your_executable
```

The `-it` option tells docker to start an interactive shell, and the `-v /path/on/host:/path/on/container` option mounts a directory on the host to a directory on the container. In the example command above, we used the `$(pwd)` bash command to get the path of the current working directory and mounted all files in it to the directory `/opt` on the container.

This is the image we will use for testing your submissions. If it doesn't work on that image, you will get no points. It makes no difference if your submission works on another Ubuntu version (or another OS).

The amount of code you have to write in this lab is small, but you have to understand the stack. Using `gdb` (or some equivalent) is essential. The article, *Smashing The Stack For Fun And Profit*, is very helpful and gives ample details and guidance. Read it if you're stuck.

Throughout this document, the prompt for an ordinary (non-root) shell is "$", and the prompt for a root shell is "#".

2.1 Starter files

Starter files are available at: https://umd.box.com/s/ta5ocijgcqph0aieczykalg4mqzk1s5q

3 Compiling

We will be using `gcc` to compile all of the programs in this project. There are a few non-standard ways we will be using `gcc`, like turning off some protection mechanisms, with some command line arguments. We are providing a makefile that handles all of this for you (just type "make"), but it's a good idea to look at the options we provided and read about the options we chose.

3.1 Working with a debugger

`gdb` will be your best friend in this project. To get useful information from `gdb` regarding the names of functions and variables, include the `-g` commandline argument to `gcc`.
However, `GDB` is command line driven and has a steep learning curve. You may find it easier to use `DDD`, a graphical front end for `GDB`, or a GUI based debugger from an IDE such as Clion or Visual Studio. Please note that the TAs can't help you use or configure other debuggers.

3.2 Compiling to 32-bit

This semester, we will be using the latest, 64-bit version of Ubuntu. For the sake of this project, however, we will be running in 32-bit. Our makefile handles this for you.

3.3 Disabling compiler protections

The `gcc` compiler implements two security mechanisms that help protect against buffer overflows and code injection. Also, our version by default turns on a protection to help randomize addresses. For the sake of this project, we will be turning off canaries and in one case, a non-executable stack..

1. **Canaries**: `gcc` implements the idea in the "Stack Guard" paper by introducing canaries in each stack frame. You can disable this protection by compiling with the commandline argument `-fno-stack-protector`.

2. **Non-executable stack**: In the updated VM we are using, `gcc` by default will make the stack non-executable, thereby making it more difficult to launch arbitrary code. You can disable this with the commandline argument `-z execstack`.

4 Task 1: A Vulnerable Program

You will be exploiting a program that has a buffer overflow vulnerability. You are not allowed to modify the program itself; instead, you will be attacking it by cleverly constructing malicious *inputs* to the program.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* this header exports functions to execute the exploit and read/write
 * to/from it */
#include "comms.h"

void sensitive_function()
{
```

```
    puts("Full points! (Hey wait a minute.. How did you get here?)");
    exit(EXIT_SUCCESS);
}

void buffer_overflow()
{
    char secret[32];
    char buffer[48];

    sprintf(secret, "authorized personnel only");

    read_from_exploit(buffer, 64);

    if(strcmp(secret, "TOP SECRET") == 0)
        sensitive_function();
}

static char greeting[128];

int main()
{
    int local = 5;

    exec_exploit("./exploit1.x");

    read_from_exploit(greeting, sizeof(greeting)-1);

    write_to_exploit(greeting);

    puts("Waiting for input...");
    buffer_overflow();

    puts("Zero points. (Program terminated successfully; overflow failed.)");



    return EXIT_SUCCESS;
}
```

The vulnerable program for Task 1, `vulnerable1.c`, is given above. Use the makefile to compile it, generating two executables, "exploit1.x" and "vulnerable1.x."

The above program prints a greeting supplied from the exploit program and later, has a buffer-overflow vulnerability in function `buffer_overflow()`. The vulnerable program, which runs as root, executes an unpriviledged instance of exploit1 and communicates with it via the `write_to_exploit` and `read_from_exploit`. The function signatures for `write_to_exploit` and `read_from_exploit` are in comms.h. You do not need to interact with comms.c or comms.h. `write_to_exploit` has the same function signature and behavior as `printf`, except that it writes to exploit's stdin instead of to the console. `read_from_exploit` reads the specified number of bytes into the buffer provided. The exploit, shown below, sends and receives messages to the vulnerable program through stdin and stdout.

```
#include <unistd.h>
```

```c
#include <stdio.h>  // for fwrite()
#include <string.h> // for memset()
#include <stdlib.h> // for EXIT_SUCCESS

#define BUFFER_SIZE 64

int main()
{

    char buffer[BUFFER_SIZE];
    memset(buffer, 0, BUFFER_SIZE);  // Initialized to all zeroes

    fprintf(stderr, "send debug statements to stderr\n");

    /* you may change this string */
    char greeting[128] = "Exploit initiated communication with parent\n";

    write(STDOUT_FILENO, greeting, sizeof(greeting) - 1);

    /* you may do whatever you want with the response */
    char response[512];
    fgets(response, sizeof(response), stdin);

    // TODO: Populate the buffer here, as you see fit
    write(STDOUT_FILENO, buffer, BUFFER_SIZE);

    return EXIT_SUCCESS;
}
```

Because `stdin` and `stdout` now redirect to the vulnerable program, the `write` and `fgets` calls write to and read from the vulnerable program. To print out debugging information, print to `stderr`. The final write in the exploit program is read by the vulnerable program in the `buffer_overflow` function, which results in a buffer overflow.

An attacker can exploit vulnerabilities in this program and potentially launch a shell. Moreover, because the program is a set-root-uid program (compiled as root using `sudo`), the attacker may be able to get a root shell. Doing so is your next task, but we will do so as a series of smaller steps.
For this task:

4.1 Part 1

Write a program, `exploit1.c`, that uses a buffer overflow to defeat the string comparison in `vulnerable1.c`. The goal of this part is to see where local variables are stored on the stack and how they may be modified via a buffer overflow.

4.2 Part 2

Write a program, `exploit2.c`, that exploits `vulnerable2.c` with a buffer overflow and printf data leak to overwrite the instruction pointer saved in the stack to jump into `sensitive_function`. (`vulnerable2.c` is not reproduced here but is found in the starter files.)

There are two concepts here that you will want to consider. The first is whether useful information may be obtained by carelessly printing user information with printf like functions (hint: yes). The second is how to use a buffer overflow to overwrite the saved instruction pointer on the stack.

4.3 Part 3

Write a program, `exploit3.c`, that exploits `vulnerable3.c` with a buffer overflow and printf data leak to overwrite the instruction pointer saved in the stack to jump into `sensitive_function`. This is largely similar to part 2. However, `sensitive_function` now requires a particular argument to be passed to it. (Again, `vulnerable3.c` is not reproduced here but is found in the starter files.)

4.4 Part 4

Write a program, `exploit4.c`, that exploits `vulnerable4.c` with a buffer overflow and printf dataleak to push shellcode onto the stack and execute it. This part requires all of the machinery as part 2, but now you must introduce new executable instructions into the process's address space and correctly set the instruction pointer so that it jumps to that location. The makefile compiles vulnerable4.c with an executable stack. (Again, `vulnerable4.c` is not reproduced here but is found in the starter files.)

5 Task 2 (Extra credit): Re-enabling common defenses

As mentioned in Section 3.3, `gcc` by default puts some protective measure in place to mitigate buffer overflows and code injections. We got around these with commandline options to turn off canaries (`-fno-stack-protector`) and to make the stack executable (`-z execstack`). For extra credit, get **Task 1 Part 4** to work without one or both of these options (more credit for both).
If you choose to do this task, submit all relevant files in a subdirectory called `task2/` and include a file named `task2/readme.txt` that describes which defense(s) you attacked and how you went about launching your attack. Place your exploit code in a file named `task2/exploit.c` and include any other files that were necessary in launching this attack, if there were any.

6 What to submit

You will submit this homework via ELMS.

Submit the following files (We have provided a `subcheck.pl` file that will check to make sure that your submission directory contains these files; however, it does not automatically test your code, compile it, etc.). You can test to see if your directory `dir` has all of the required files by running `./subcheck.pl dir` from the command line.

Required:

1. `task1/exploit1.c`
2. `task1/exploit2.c`
3. `task1/exploit3.c`
4. `task1/exploit4.c`

Optional (extra credit):

4. Files for extra credit Task 2 stored in a `task2/` directory:

- `task2/exploit.c`
- `task2/readme.txt`: Your description of which defenses you got around (non-executable stack and/or canaries) and how.
- Any additional files you need to launch this attack.

Prepare your submission tarball with the following command, assuming your `exploit*.c` files are located in `/opt/hw1-buffer/task1`:

```
$ tar -C /opt/hw1-buffer -czvf <lastname>.<firstname>.tgz task1/exploit1.c \
task1/exploit2.c task1/exploit3.c task1/exploit4.c
```

Submit `<lastname>.<firstname>.tgz` to ELMS.

Note: Only the latest submission counts.

Grading Rubric:

Task	Points
1.1	10
1.2	30
1.3	30
1.4	30
2 (optional, extra credit)	10 (for breaking one defense), 15 (for both)

7 Some extra background information

This section contains additional background on creating shell code and injecting code. We have included the files discussed here in the `background/` directory in the starter files.

7.1 Shellcode

A **shellcode** is binary code that launches a shell. Consider the following C program:

```
/* start_shell.c */

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

The machine code obtained by compiling this C program can serve as a shellcode. However it would typically not be suitable for a buffer-overflow attack (e.g., it would not be compact, it may contain `0x00`) entries). So one usually writes an assembly language program, and assembles that to get a shellcode.

We provide the shellcode that you will use in the stack. It is included in `call_shellcode.c`, but let's take a quick divergence into it now:

```
/* call_shellcode.c  */

/* A program that executes shellcode stored in a buffer */
#include <stdlib.h>
#include <stdio.h>
#include <string.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"                 /* cdql                      */
  "\xb0\x0b"             /* movb    $0x0b,%al         */
  "\xcd\x80"             /* int     $0x80             */
;
int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)( ))buf)( );
}
```

This program contains the shellcode in a `char[]` array. Compile this program, run it, and see whether a shell is invoked. Also, compare this shellcode with the assembly produced by `gcc -S start_shell.c`.

A few places in this shellcode are worth noting:

- 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 used here (`cdql`) is simply a shorter instruction. Third, the system call `execve()` is called when we set `%al` to `11`, and execute "`int $0x80`".

7.2 Guessing runtime addresses for vulnerable program

Consider an execution of our vulnerable program, `vuln`. For a successful buffer-overflow attack, we need to guess two runtime quantities concerning the stack at `bof()`'s invocation.

1. The distance, say R, between the overflowed buffer and the location where `bof()`'s return address is stored. The target address should be positioned at offset R in `badfile`.
2. The address, say T, of the location where the shellcode starts. This should be the value of the target address.

See Figure 1 for a pictorial example.

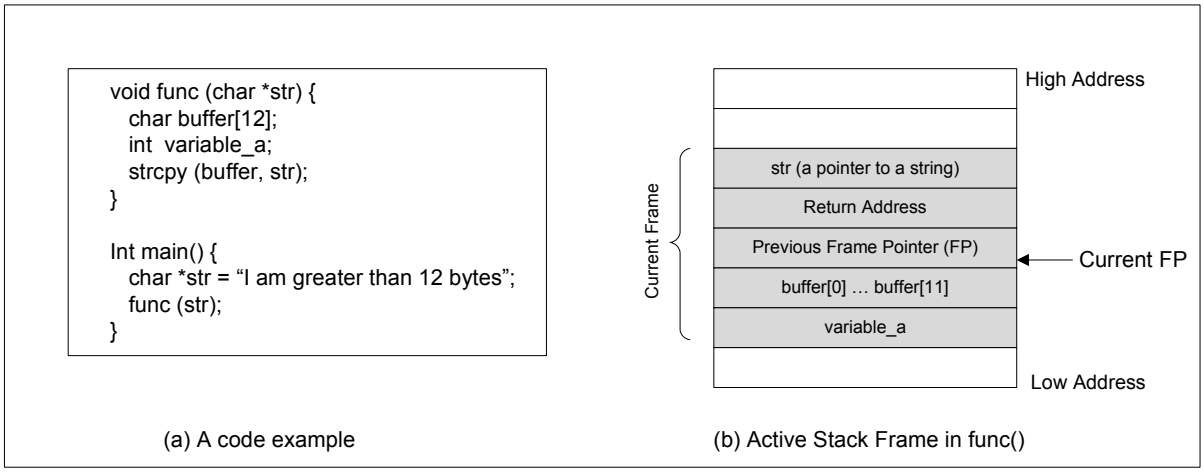

Figure 1: Buffer overflow stack example.

If the source code for a program like `vuln` is available, it is easy to guess R accurately, as illustrated in the previous figure. Another way to get R is to run the executable in a (non-root) debugger. The value obtained for R by these methods should be close, if not the same as, as the value when the vulnerable program is run during the attack.

If neither of these methods is applicable (e.g., the executable is running remotely), one can always *guess* a value for R. This is feasible because 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. Therefore the range of R that we need to guess is actually quite small. Furthermore, we can cover the entire range in a single attack by overwriting all its locations (instead of just one) with the target address.

Guessing T, the address of the shellcode, can be done in the same way as guessing R. If the source of the vulnerable program is available, one can modify it to print out T (or the address of an item a fixed offset away, e.g., buffer or stack pointer). Or one can get T by running the executable in a debugger. Or one can *guess* a value for T.

If address space randomization is disabled, then the guess would be close to the value of T when the vulnerable program is run during the attack. This is because (1) the stack of a process starts at the same address (when address randomization is disabled); and (2) the stack is usually not very deep.

Here is a program to print out the value of the stack pointer (source).

```
/* SeeSP.c */
#include <stdio.h>
#include <inttypes.h>

int main(void)
{
    register uintptr_t sp asm ("sp");
    printf("SP: 0x%016" PRIxPTR "\n", sp);
    return 0;
}
```

7.3 Improving the odds

To improve the chance of success, you can add a number of NOPs to the beginning of the malicious code; jumping to any of these NOPs will eventually get execution to the malicious code. Figure 2 depicts the attack.

7.4 Storing a long integer in a buffer

In your exploit program, you may need to store a `long` integer (4 bytes) at position `i` of a `char` buffer `buffer[]`. Since each buffer entry is one byte long, the integer will occupy positions `i` through `i+3` in `buffer[]`. Because `char` and `long` are of different types, you cannot directly assign the integer to `buffer[i]`; instead you can cast `buffer+i` into a `long` pointer and then assign the integer, as shown below:

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

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

Bibliography

1. Aleph One. Smashing The Stack For Fun And Profit. *Phrack 49*, Volume 7, Issue 49. Available here: https://umd.box.com/s/obc65v0lkfdt9755rzk8tgybwd2r235y

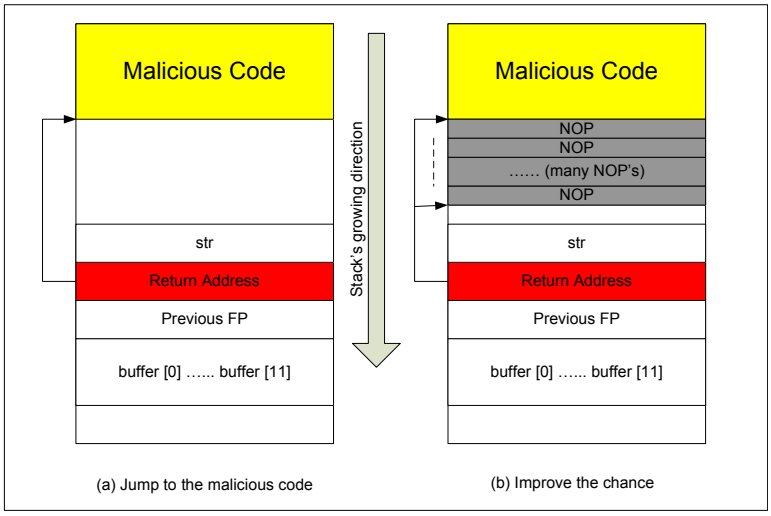

Figure 2: NOP sled example.