专业特色选修课《网络信息安全》

缓冲区溢出和渗透测试

Buffer Overflow and Penetration Testing

嵩天 教授、博士生导师

songtian@bit.edu.cn

北京理工大学网络空间安全学院


Wana DecryptOr 2.0
Ooops, your files have been encrypted!
Chinese (simpli
我的电脑出了什么问题？
您的一些重要文件被我加密保存了。
照片、图片、文档、压缩包、音频、视频文件、exe文件等，几乎所有类型的文件都被加密了，因此不能正常打开。
这和一般文件损坏有本质上的区别。您大可在网上找找恢复文件的方法，我敢保证，没有我们的解密服务，就算老天爷来了也不能恢复这些文档。
有没有恢复这些文档的方法？
当然有可恢复的方法。只能通过我们的解密服务才能恢复。我以人格担保，能够提供安全有效的恢复服务。
但这是收费的，也不能无限期的推迟。
请点击 <Decrypt> 按钮，就可以免费恢复一些文档。请您放心，我是绝不会骗你的。
但想要恢复全部文档，需要付款点费用。
是否随时都可以固定金额付款，就会恢复的吗，当然不是，推迟付款时间越长对你不利。
最好3天之内付款费用，过了三天费用就会翻倍。
还有，一个礼拜之内未付款，将会永远恢复不了。
对了，忘了告诉你，对半年以上没钱付款的穷人，会有活动免费恢复，能否轮
Payment will be raised on
5/16/2017 02:26:59
Time Left
02:22:35:15
Your files will be lost on
5/20/2017 02:26:59
Time Left
06:22:35:15
About bitcoin
How to buy bitcoins?
Contact Us
bitcoin
ACCEPTED HERE
Send $300 worth of bitcoin to this address:
13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94
Copy
Check Payment
Decrypt

本节大纲

- 缓冲区溢出概述
- 缓冲区溢出原理
- 缓冲区溢出的利用

缓冲区溢出是真正的“杀手锏”技术

缓冲区溢出、社会工程学、DDoS至今无解

缓冲区溢出概述

- **缓冲区**

包含相同数据类型实例的一个连续的计算机内存块

或 程序运行期间在内存中分配的一个连续的区域

- **溢出**

  所填充的数据超出了原有的缓冲区边界

- **缓冲区溢出**

缓冲区溢出概述

- **缓冲区溢出的历史**

1988年，Morris蠕虫使fingerd程序溢出

1989年，Spafford提交了一份分析报告，描述了fingerd缓冲区溢出程序的技术细节，引起了重视

1996年， Aleph One详细解释了其中原理

Smashing the stack for fun and profit

缓冲区溢出概述

- **缓冲区溢出被利用的历史**

2001年“红色代码”蠕虫利用微软IIS Web Server缓冲区溢出漏洞攻击超过300 000台计算机

2003年1月，Slammer蠕虫爆发，利用微软SQL Server 2000漏洞

2004年5月，“振荡波”利用Windows系统的活动目录服务缓冲区溢出漏洞

缓冲区溢出概述

- **几个事实**

缓冲区溢出已占所有系统攻击总数的80%以上

各OS和软件存在的缓冲区溢出问题数不胜数

可导致程序运行失败、系统崩溃、执行非授权指令、取得系统特权等后果

本节大纲

- 缓冲区溢出概述
- 缓冲区溢出原理
- 缓冲区溢出的利用

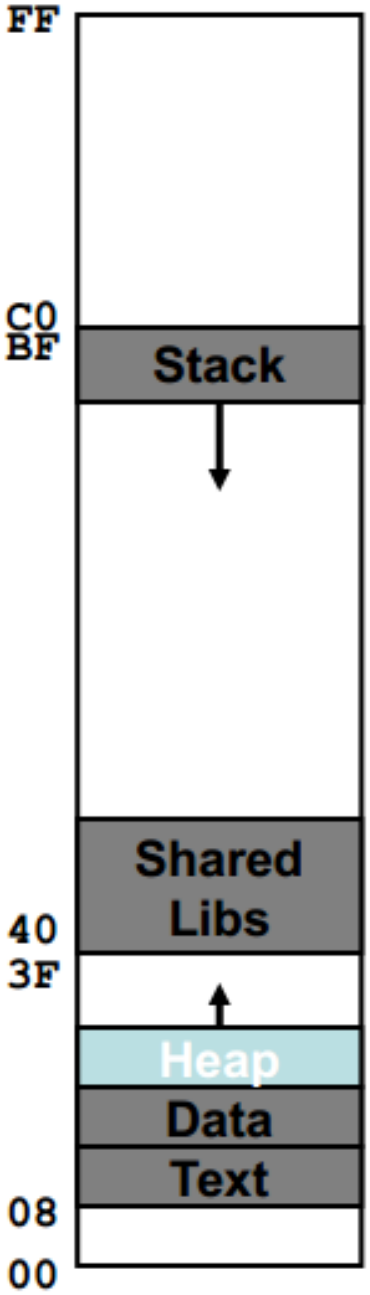
缓冲区溢出原理


内存高地址
命令行参数及环境变量
栈(动态)
堆(动态)
BSS段(静态)
数据段(静态)
代码段(只读)
堆栈区
数据区
代码区
低地址
程序在内存中的存放方式
栈
BSS: Block Started by Symbol
堆
未初始化全局或静态变量
初始化全局或静态变量
程序指令和只读数据

缓冲区溢出原理

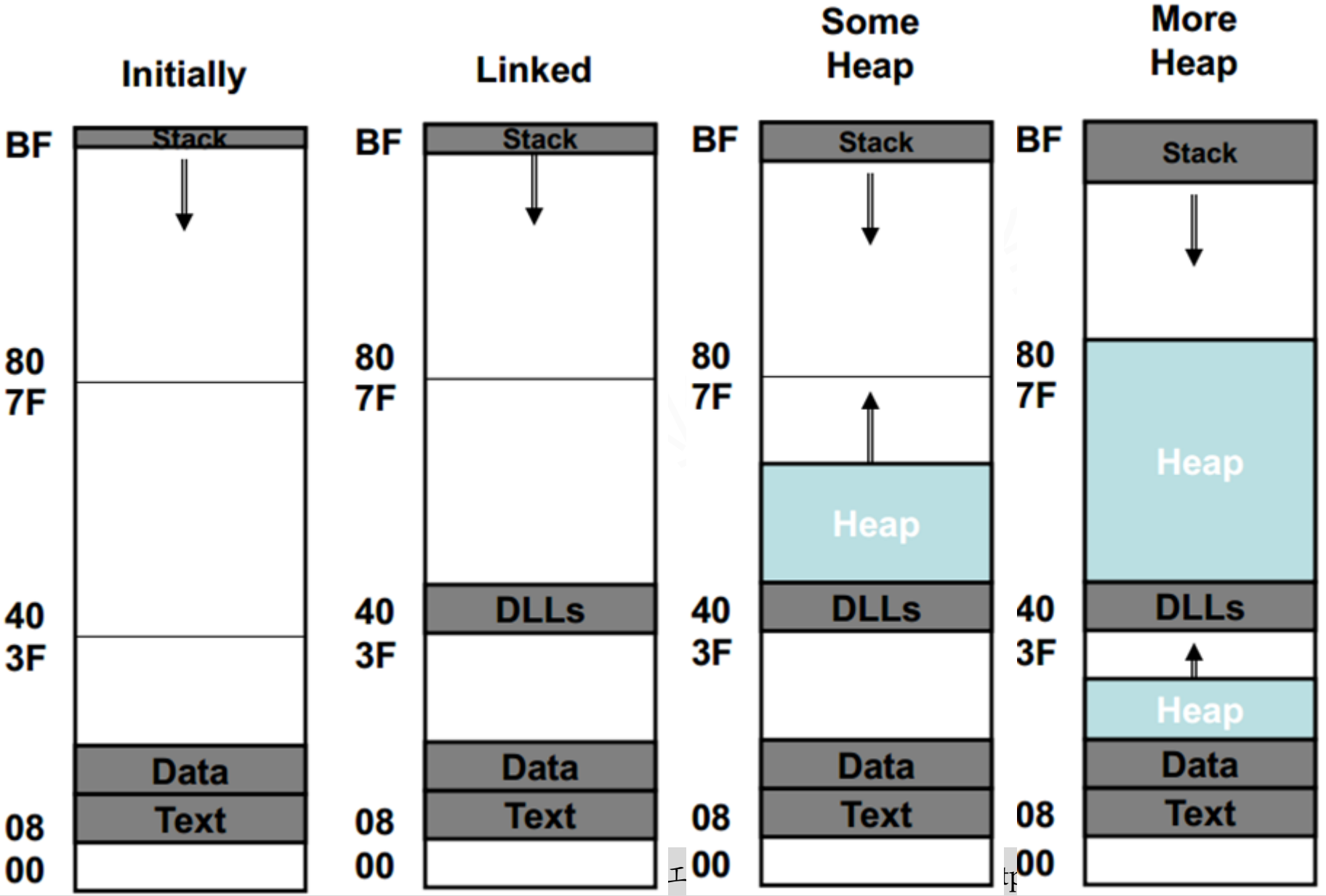
- **几个问题**
  - **静态分配和动态分配的区别?**
  - **static 声明的变量放在那里?**
  - **BSS段和数据段中的变量在初始化上有何区别?**
  - **malloc()函数分配的内存空间在哪里?**
  - **频繁malloc和free对内存空间造成什么影响?**

Linux内存使用



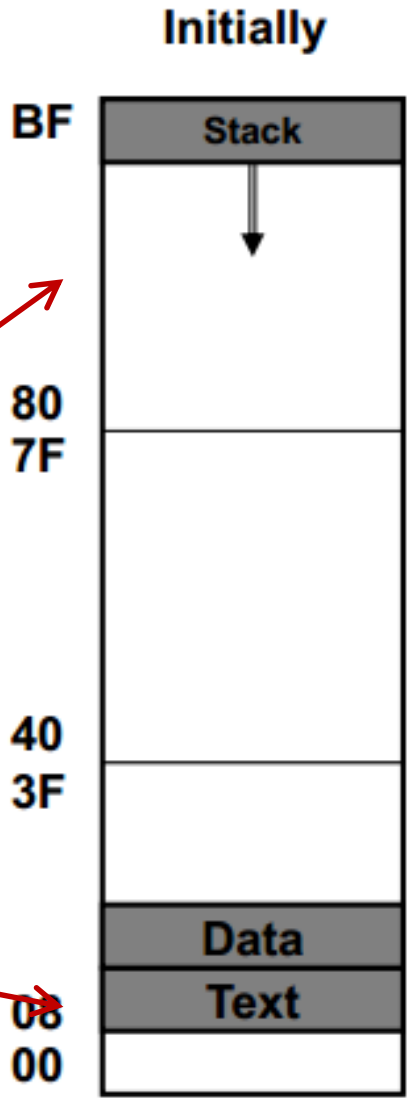
- **32bit系统**
  - **4GB memory**
  - **0-1GB: 用户空间 (text, code, malloc)**
  - **1-3GB: 用户空间 (shared libs, stack)**
  - **3-4GB: 内核空间**

Linux内存使用


Initially
Linked
Some Heap
More Heap
BF
80
7F
40
3F
08
00
Stack
Heap
DLLs
Data
Text

基本程序实例

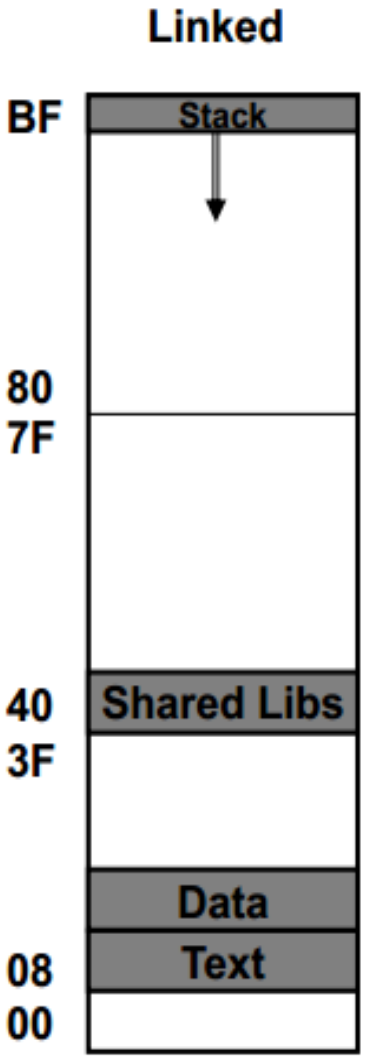
```
(gdb) break main
(gdb) run
  Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
  $3 = (void *) 0xbffffc78
```



- **栈顶**
  - 地址 0xbffffc78
- **main()**
  - 地址 0x804856f (0x0804856f)

动态链接实例

```
(gdb) print malloc
  $1 = {<text variable, no debug info>}
    0x8048454 <malloc>
(gdb) run
  Program exited normally.
(gdb) print malloc
  $2 = {void *(unsigned int)}
    0x40006240 <malloc>
```



- 程序运行前的malloc
  - 地址　0x8048454 (0x08048454)
- 程序运行后的malloc
  - 地址　0x40006240

内存分配实例

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

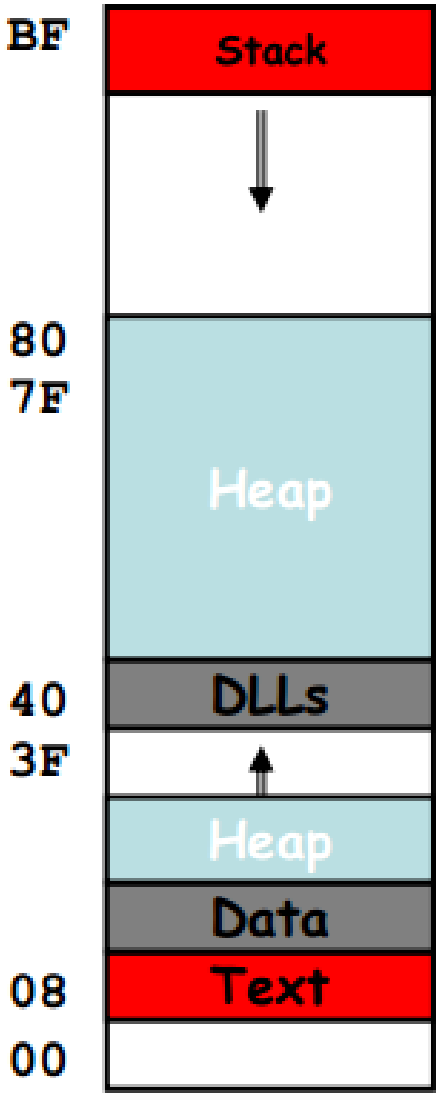
内存分配实例

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }
```

$esp	0xbffffc78
p3	0x500b5008
p1	0x400b4008
Final malloc	0x40006240
p4	0x1904a640
p2	0x1904a538
beyond	0x1904a524
big_array	0x1804a520
huge_array	0x0804a510
main()	0x0804856f
useless()	0x08048560
Initial malloc	0x08048454



缓冲区溢出原理

如果在堆栈中压入的数据超过预先给堆栈分配的容量时，就会出现堆栈溢出，从而使得程序运行失败；如果发生溢出的是大型程序还有可能会导致系统崩溃

系统中所有缓冲区都有溢出的可能性

缓冲区溢出原理

缓冲区溢出包括三种

栈溢出 （最常用）

堆溢出

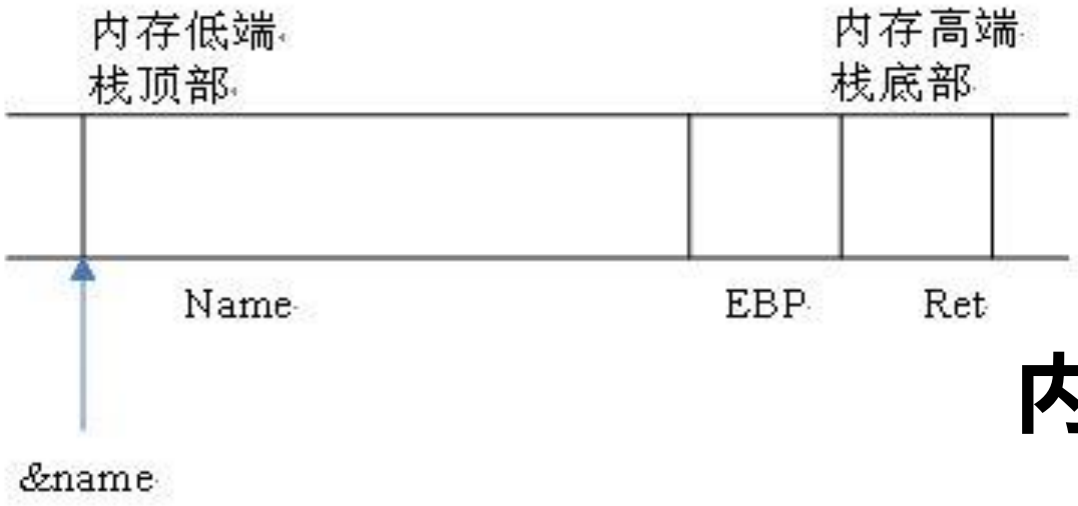
BSS溢出

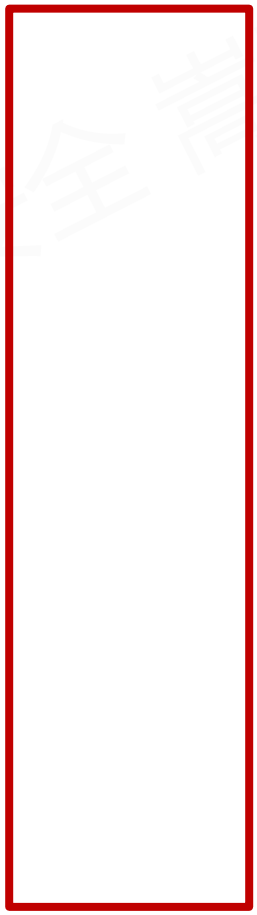
栈溢出实例

```
#include <stdio.h>
int main(){
      char name[16];
      gets(name);
      for(int i=0;i<16&&name[i];i++)
          printf("%c",name[i]);
}
```

栈溢出实例

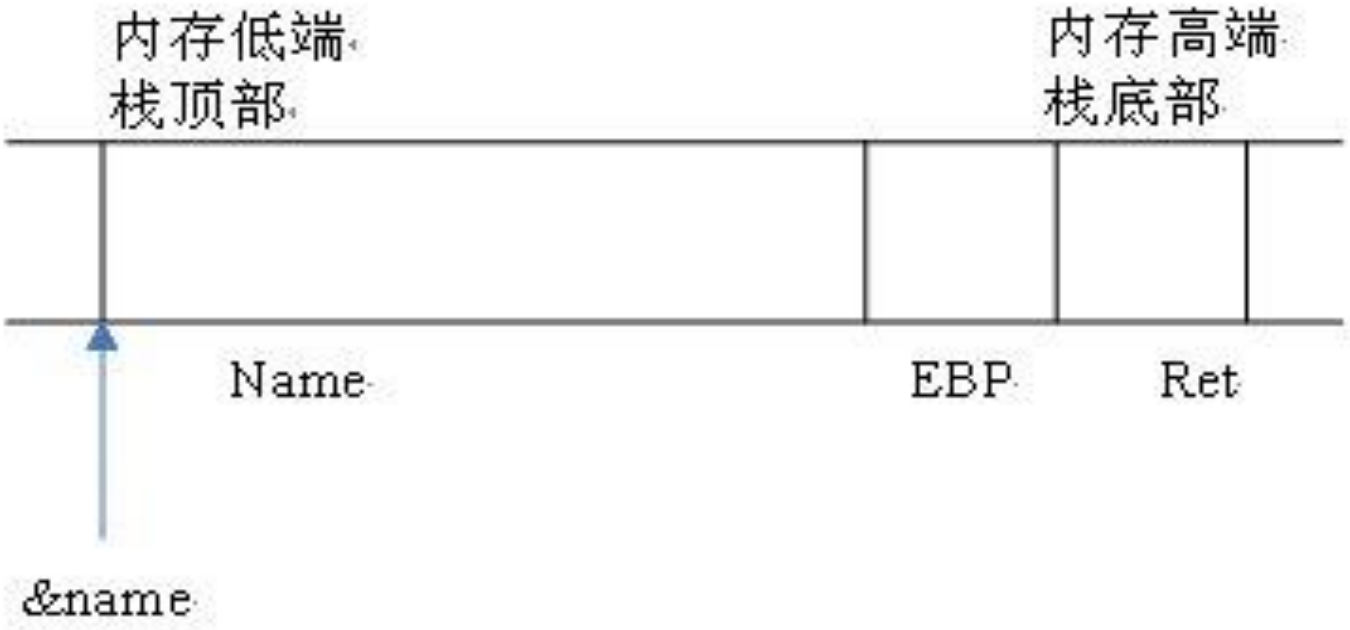
初始栈内存分布





栈溢出实例

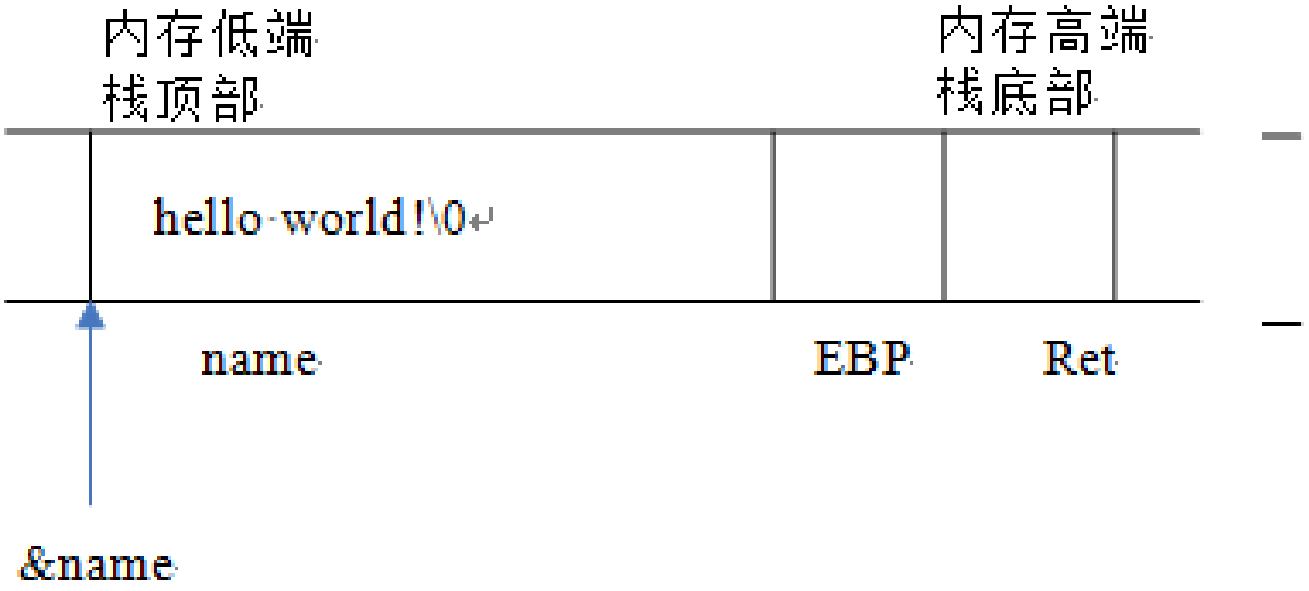
初始栈内存分布



栈溢出实例

执行gets(name)后的栈内存分布

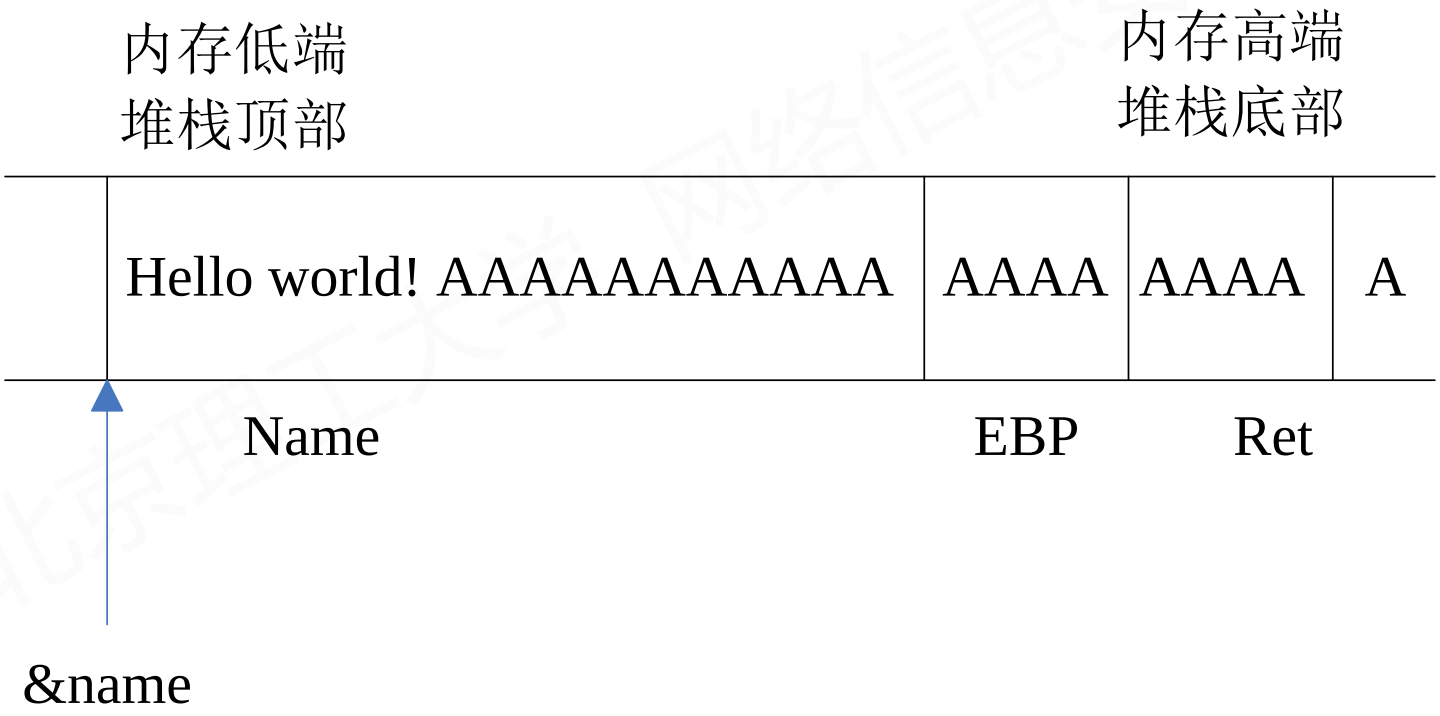
输入是“hello world!”



栈溢出实例

执行gets(name)后的栈内存分布

输入是“hello world！AAAAAA……”




hacker
inside

本节大纲

- 缓冲区溢出概述
- 缓冲区溢出原理

  栈溢出、堆溢出、BSS溢出
- 缓冲区溢出的利用

缓冲区溢出原理


内存高地址
命令行参数及环境变量
栈(动态)
堆(动态)
BSS段(静态)
数据段(静态)
代码段(只读)
堆栈区
数据区
代码区
低地址
程序在内存中的存放方式
栈
BSS: Block Started by Symbol
堆
未初始化全局或静态变量
初始化全局或静态变量
程序指令和只读数据

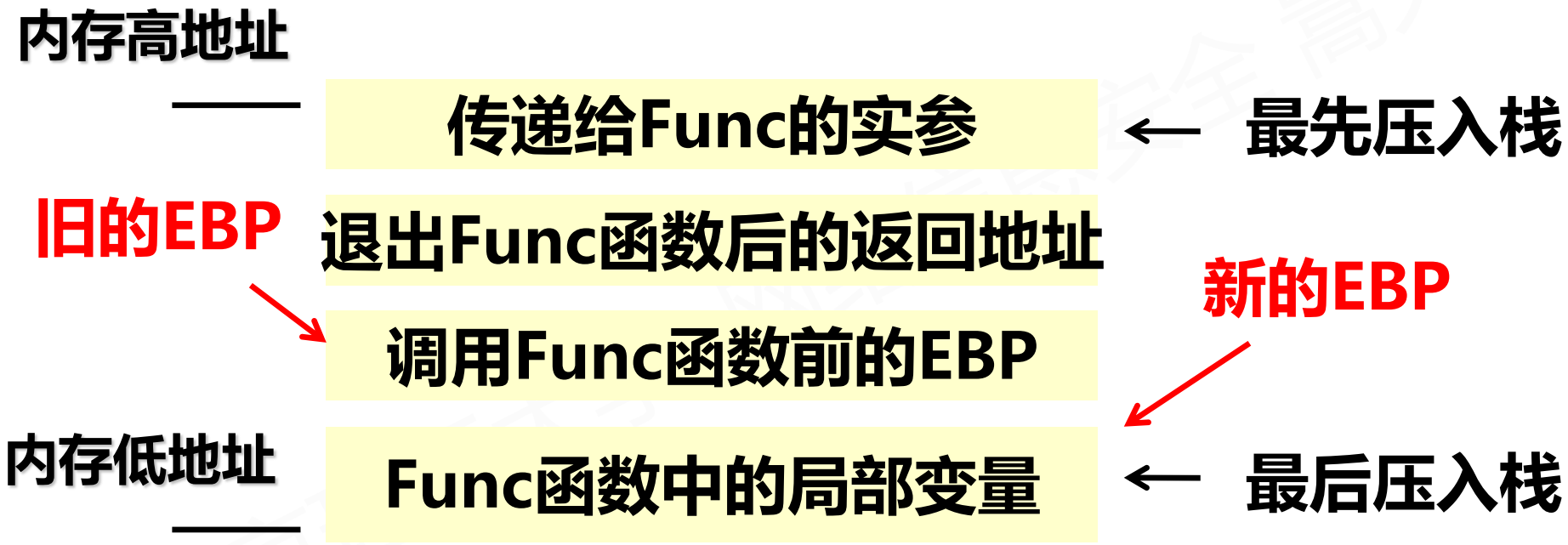
（1）栈溢出

- **2个寄存器**
  - **SP (ESP)**

    栈顶指针，随着数据入栈和出栈而变化
  - **BP (EBP)**

    基地址指针，标记栈中一个相对稳定的位置

栈的使用


内存高地址
传递给Func的实参
← 最先压入栈
旧的EBP
退出Func函数后的返回地址
新的EBP
调用Func函数前的EBP
内存低地址
Func函数中的局部变量
← 最后压入栈

栈溢出

- **栈操作过程**
  - EIP压入栈顶，表示为RET
  - 压入EBP
  - 把当前栈指针ESP赋值给EBP
  - 根据局部变量大小预留空间

栈溢出

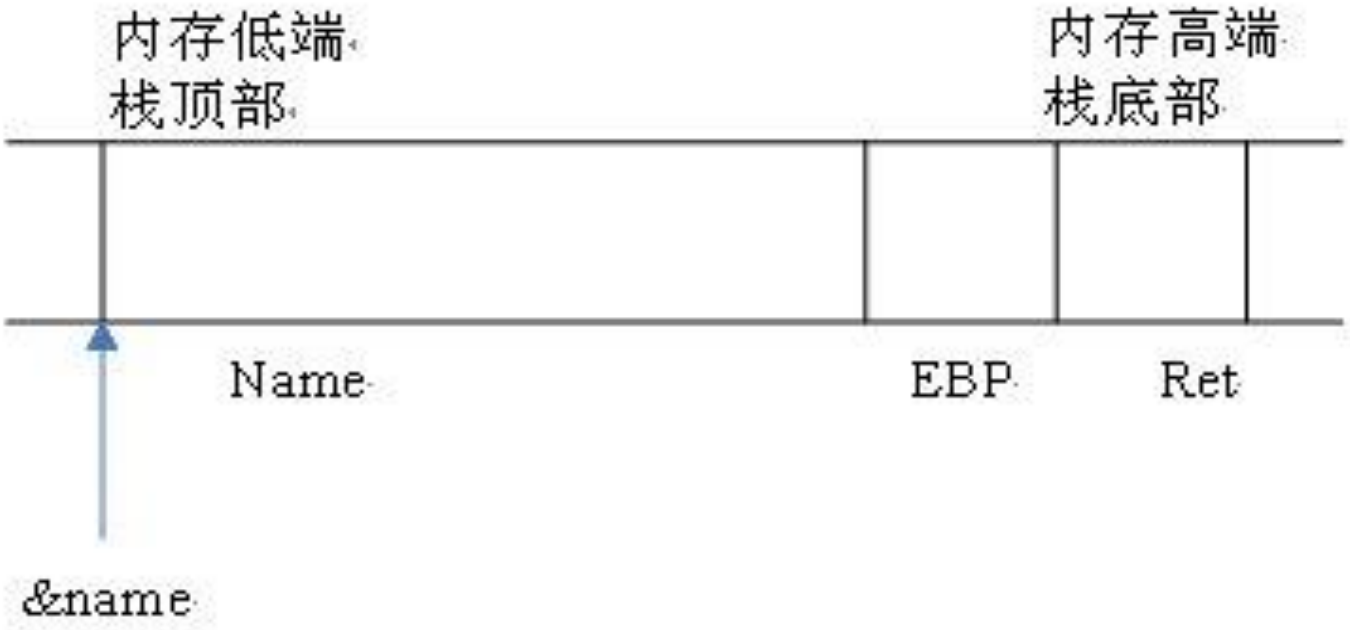
- **回答：**
  - **EBP是什么？**
  - **ESP是什么？**

栈溢出实例

```
#include <stdio.h>
int main(){
    char name[16];
    gets(name);
    for(int i=0;i<16&&name[i];i++)
        printf("%c",name[i]);
}
```

栈溢出实例

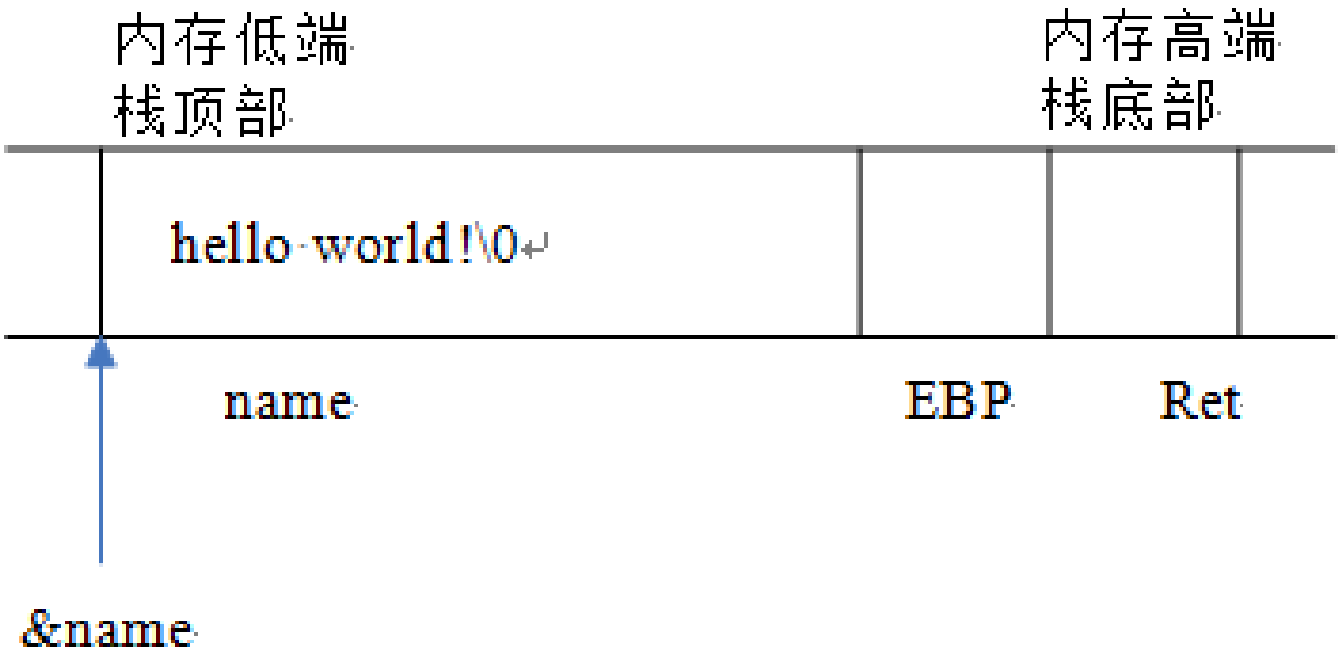
初始栈内存分布



栈溢出实例

执行gets(name)后的栈内存分布

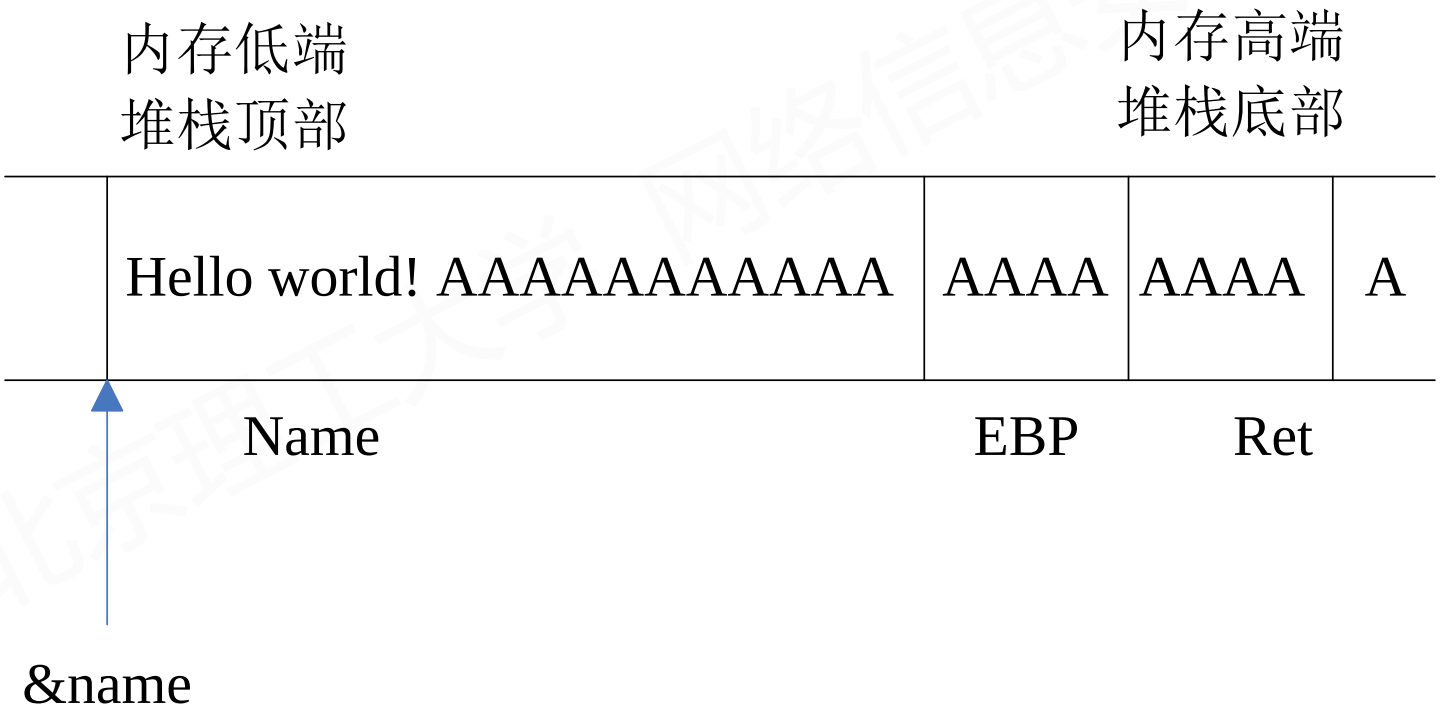
输入是“hello world！”



栈溢出实例

执行gets(name)后的栈内存分布

输入是“hello world！AAAAAA……”



栈溢出实例 echo()

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

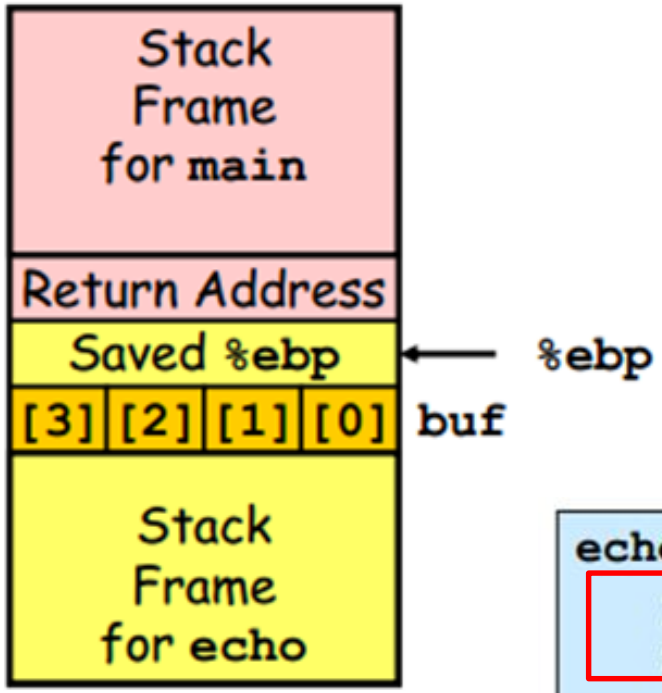
栈溢出实例

```
unix>./bufdemo
Type a string:123
123
```

```
unix>./bufdemo
Type a string:12345
Segmentation Fault
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

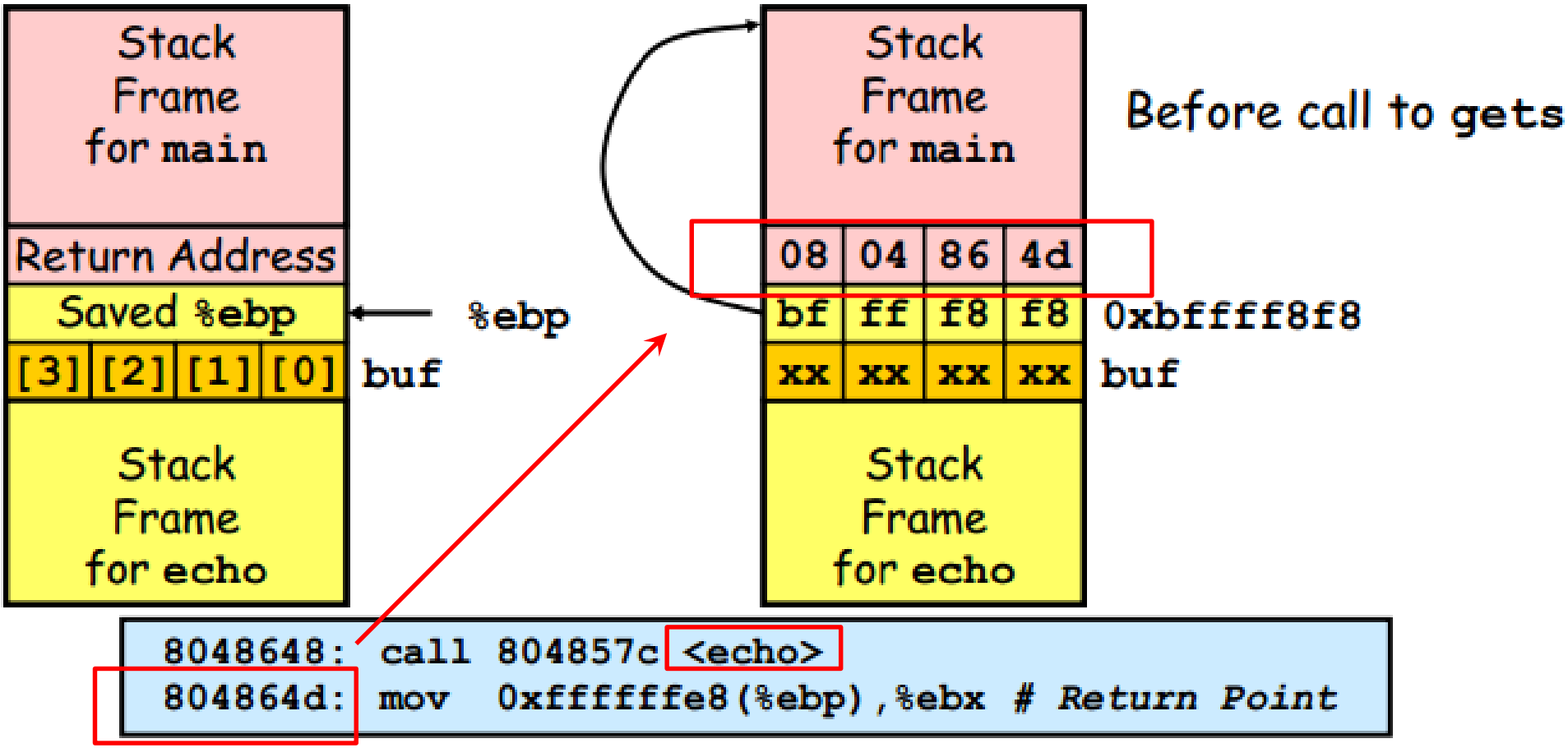
栈溢出实例



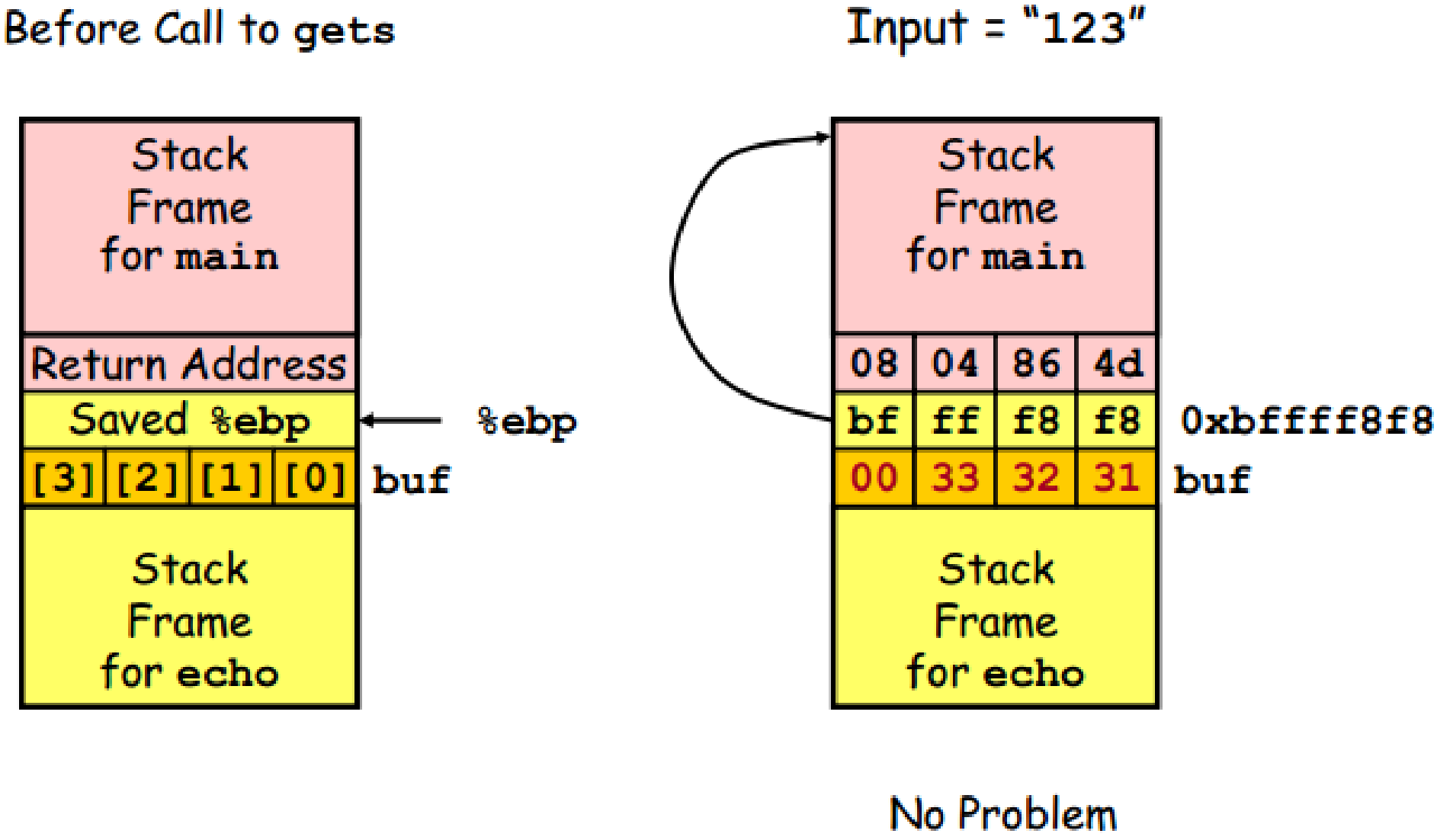

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl %esp,%ebp
    subl $20,%esp           # Allocate stack space
    pushl %ebx              # Save %ebx
    addl $-12,%esp          # Allocate stack space
    leal -4(%ebp),%ebx      # Compute buf as %ebp-4
    pushl %ebx              # Push buf on stack
    call gets               # Call gets
    . . .
```

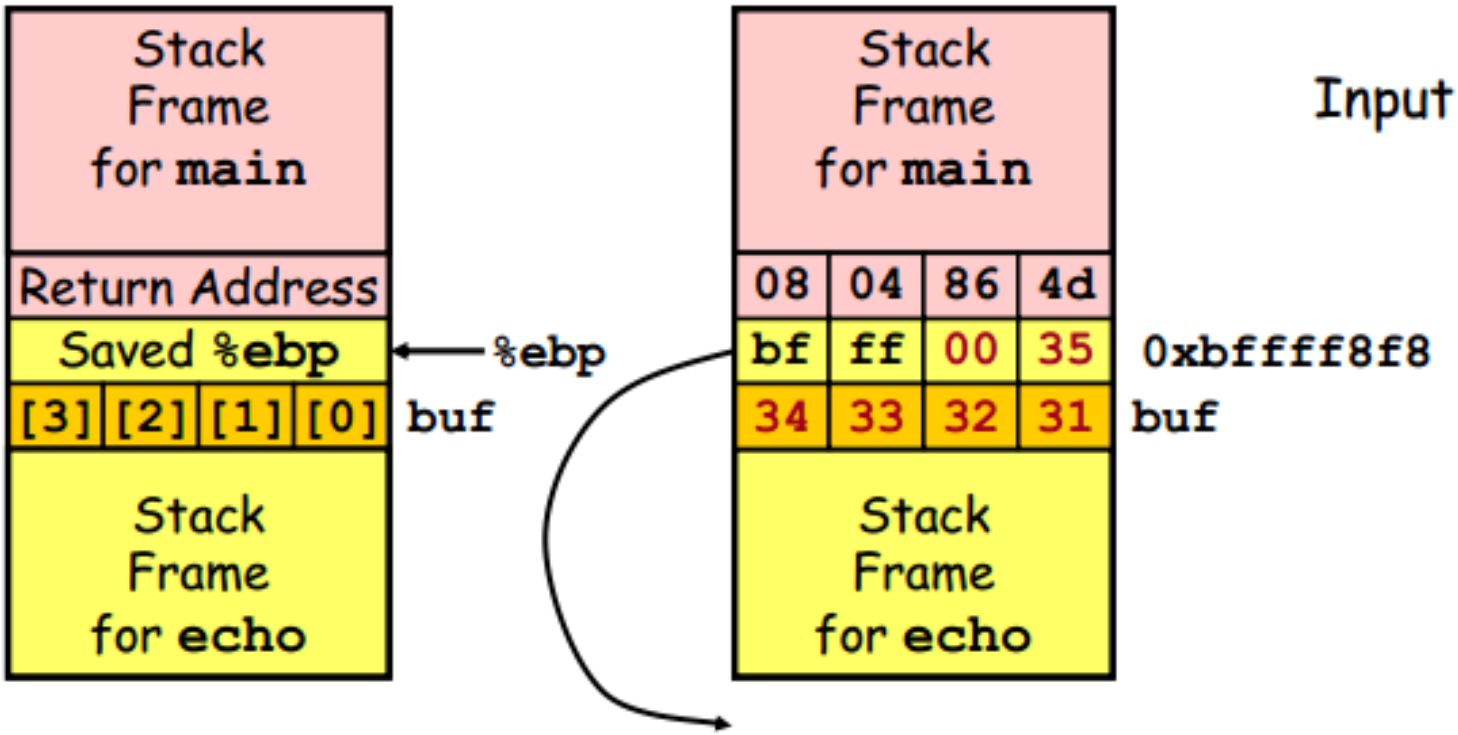
栈溢出实例


Stack Frame for main
Return Address
Saved %ebp
[3] [2] [1] [0]
buf
Stack Frame for echo
%ebp
Stack Frame for main
08 04 86 4d
bf ff f8 f8
0xbffff8f8
xx xx xx xx
buf
Stack Frame for echo
Before call to gets
8048648: call 804857c <echo>
804864d: mov 0xffffffe8(%ebp),%ebx # Return Point

栈溢出实例


Before Call to gets
Input = "123"
Stack Frame for main
Return Address
Saved %ebp
[3] [2] [1] [0]
buf
%ebp
Stack Frame for echo
Stack Frame for main
08 04 86 4d
bf ff f8 f8
0xbffff8f8
00 33 32 31
buf
Stack Frame for echo
No Problem

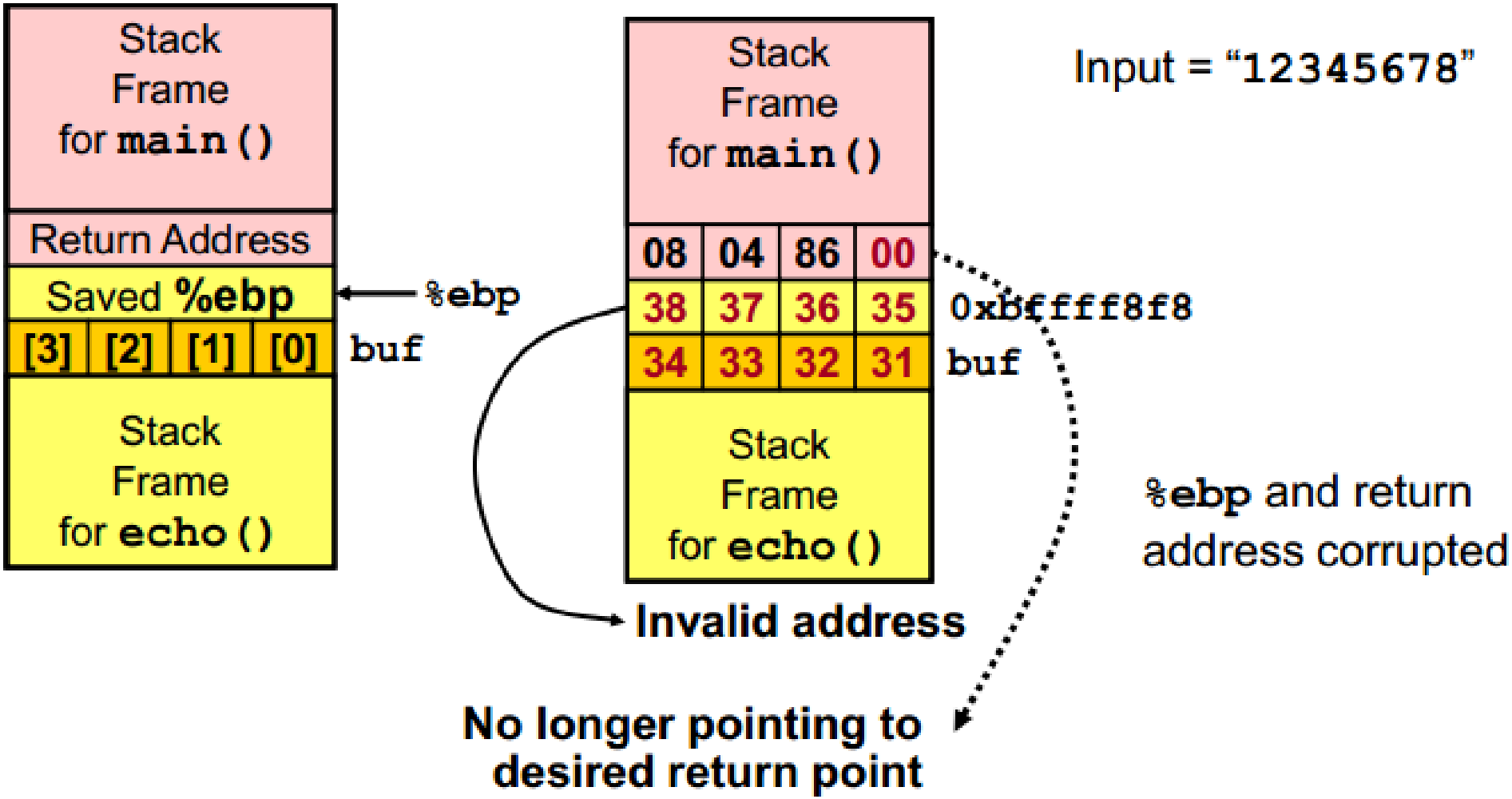
栈溢出实例



echo code:

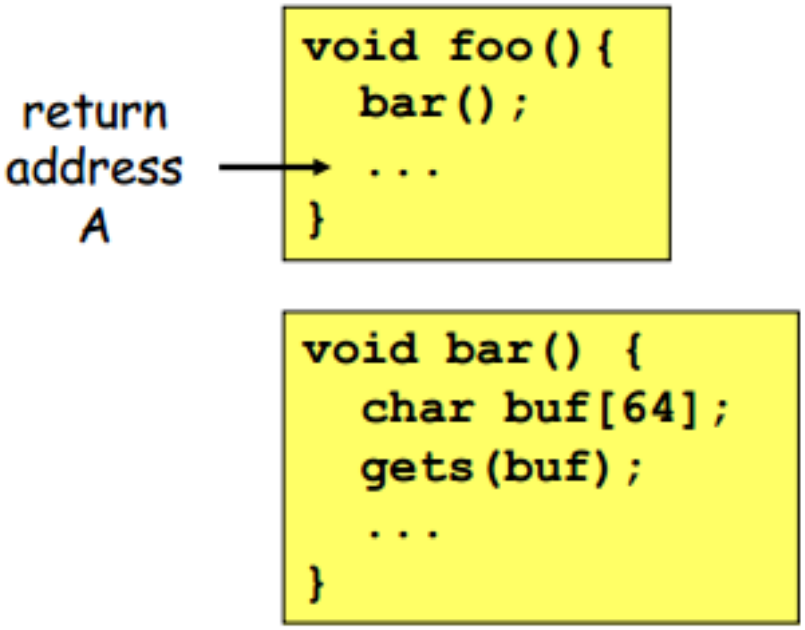
```
8048592:  push   %ebx
8048593:  call   80483e4 <_init+0x50>  # gets
8048598:  mov    0xfffffff8(%ebp),%ebx
804859b:  mov    %ebp,%esp
804859d:  pop    %ebp     # %ebp gets set to invalid value
804859e:  ret
```

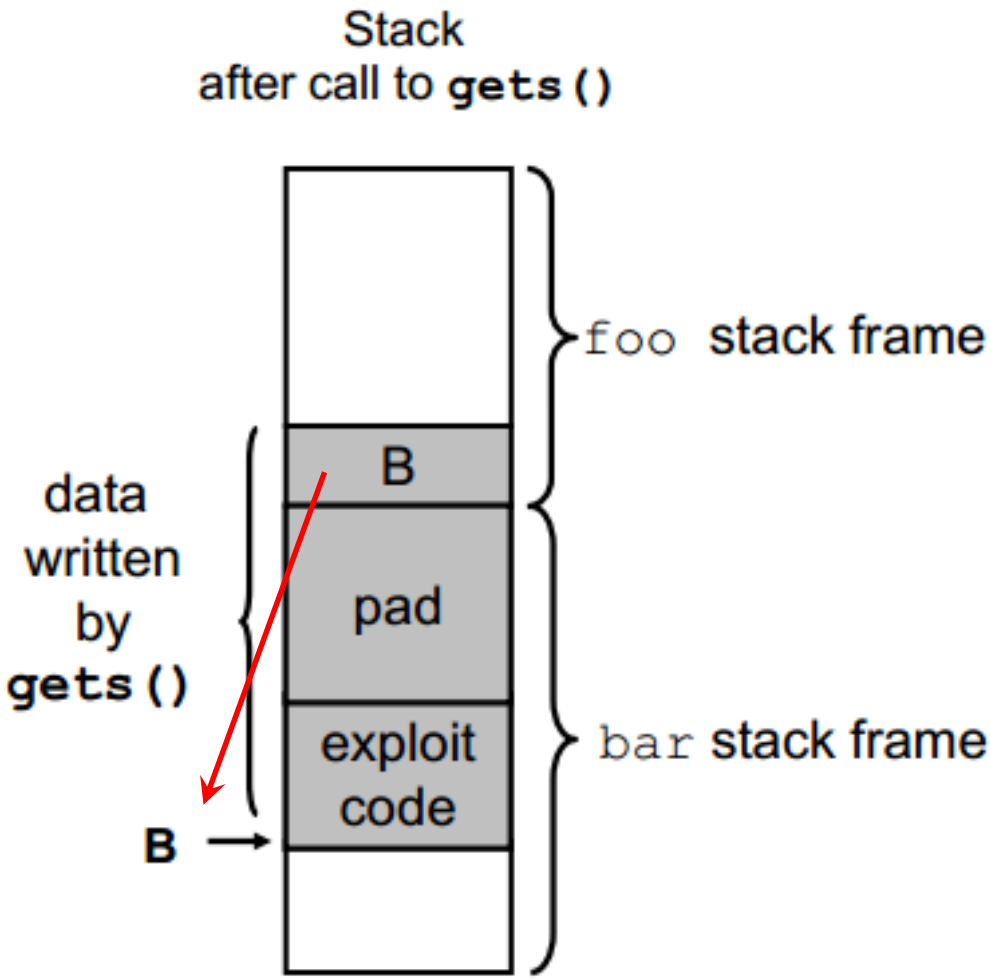
栈溢出实例




```
8048648:  call 804857c <echo>
804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

栈溢出的利用


return
address
A
void foo(){
bar();
...
}
void bar() {
char buf[64];
gets(buf);
...
}


Stack
after call to gets()
foo stack frame
B
pad
exploit
code
bar stack frame
data
written
by
gets()
B

溢出条件

- **使用特定函数**

gets(), strcpy(), scanf(), fscanf(), sscanf()

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

溢出的防范

- **限制字符串读写长度**

  fgets(char *buf, int size, FILE *fp) 或

  gets_s(char *buf, int size) 代替 gets()等

- **限制在栈区域代码执行的权限**

  栈区域内，代码不可执行

（2）堆溢出

- **堆的使用**
  - **malloc/free, new/delete**
  - **堆的操作是分配和回收**
  - **堆如何溢出?**

堆溢出实例

```
# define BUFFER_SIZE 16
int main(){
    char * buf1 = (char * )malloc (BUFFER_SIZE) ;
    char * buf2 = (char * )malloc (BUFFER_SIZE) ;
    memset (buf2, ‘a’, BUFFER_SIZE) ;
    memset (buf1, ‘b’, 2*BUFFER_SIZE);
}
```

buf2结果是：‘aaaaaaaabbbbbbbb’

buf1结果是：‘bbbbbbbbbbbbbbbb’

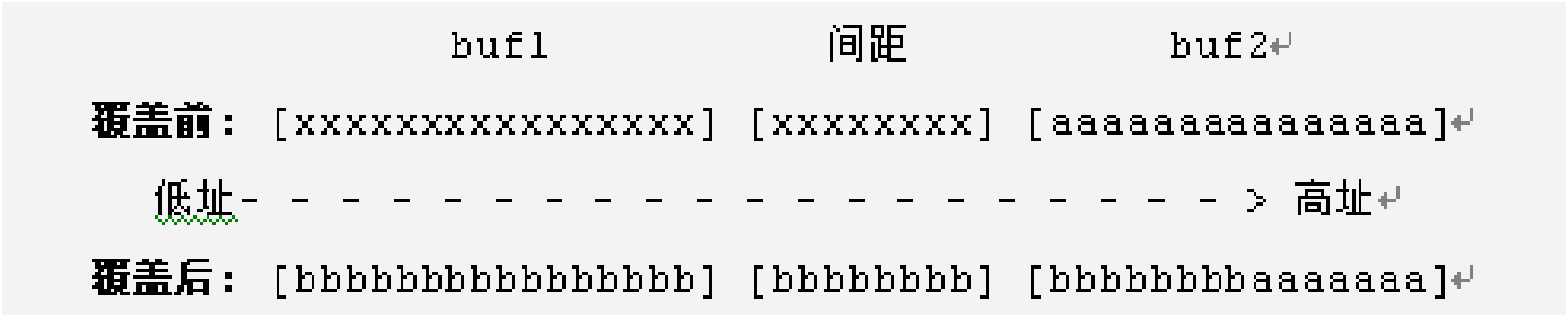
堆溢出实例

- **解释**
  - buf1和buf2是相继分配，但并不紧挨着
  - 之间存在8个字节的间距
  - 内部结构，记录分配的块长度、上一个堆的字节数以及一些标志等。

buf1结果是：　‘aaaaaaaabbbbbbbb’

buf2结果是：　‘bbbbbbbbbbbbbbbb’

堆溢出实例



- **堆溢出不如栈溢出流行**
  - **比栈溢出难度更大**
  - **对内存中变量的组织方式有一定要求**

（3）BSS段溢出

- **BSS段**

static char buf1[16]，buf2[16];

–buf1和buf2连续存储

–数组buf1的溢出内容会改写buf2的值


hacker
inside

本节大纲

- 缓冲区溢出概述
- 缓冲区溢出原理
- 缓冲区溢出的利用

缓冲区溢出应用场景

- **前提**

  不知道源代码、无法登录系统、但有网络端口

- **过程**

  了解监听网络端口的程序(开源或可执行)

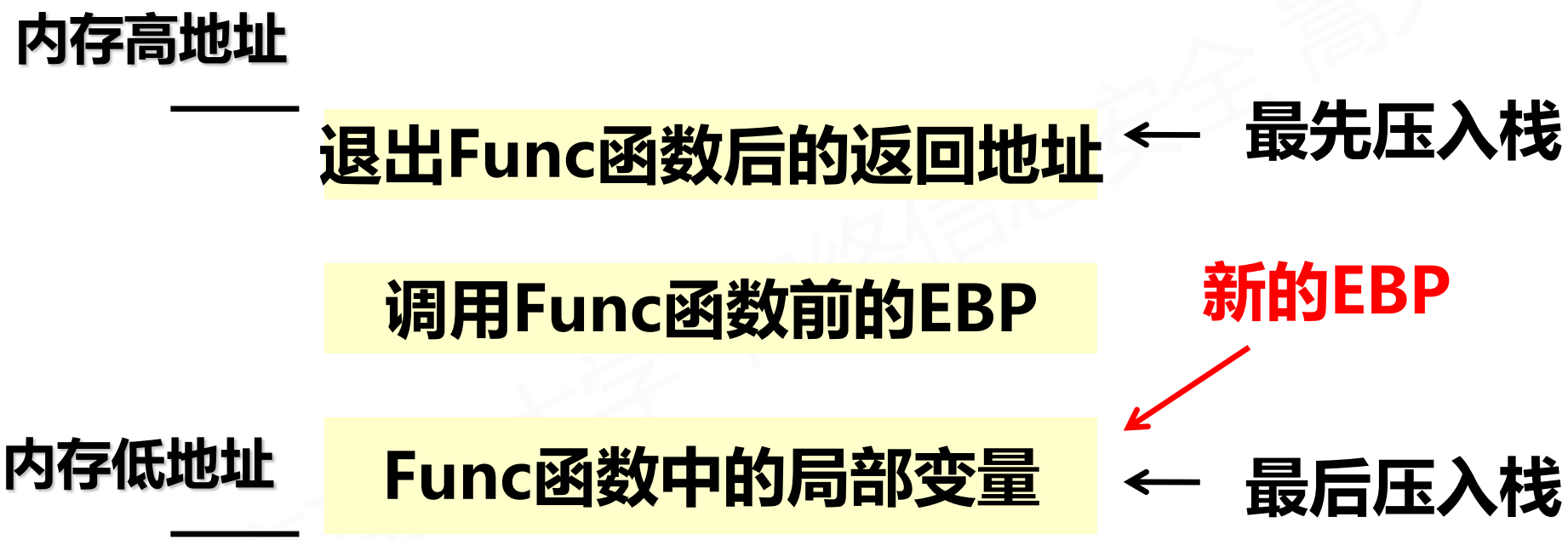
  查找漏洞、本地尝试漏洞利用

  构造网络数据开展远程攻击

缓冲区溢出原理


内存高地址
命令行参数及环境变量
栈(动态)
堆(动态)
BSS段(静态)
数据段(静态)
代码段(只读)
堆栈区
数据区
代码区
低地址
程序在内存中的存放方式
栈
BSS: Block Started by Symbol
堆
未初始化全局或静态变量
初始化全局或静态变量
程序指令和只读数据

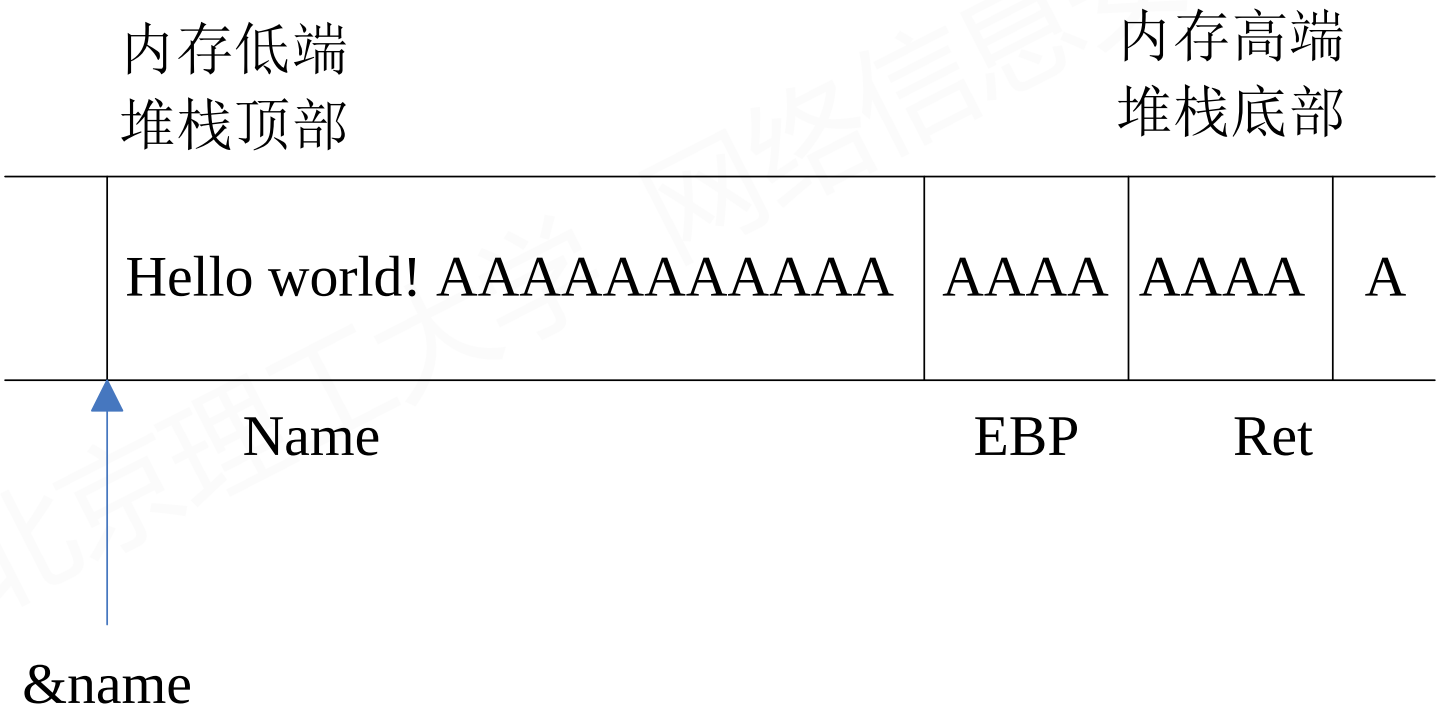
栈的使用


内存高地址
退出Func函数后的返回地址
最先压入栈
调用Func函数前的EBP
新的EBP
内存低地址
Func函数中的局部变量
最后压入栈

栈溢出实例

执行gets(name)后的栈内存分布

输入是“hello world！AAAAAA……”



缓冲区溢出利用

- **步骤1**

  在程序的地址空间安排适当代码

- **步骤2**

  使控制流跳转到攻击代码

代码植入过程

- **已有代码利用**

  攻击代码希望执行exec(‘/bin/sh’)

  在libc库中存在这样的代码exec(arg)，

  其中arg指向一个字符串的指针参数

  只需要修改该指针参数即可达到目的

代码植入过程

- **构造代码利用(exploit)**

  (1) 构造可被攻击平台执行的指令序列

  (2) 找到缓冲区空间存放该指令序列

  (3) 跳转到该指令序列执行

shellcode

- **填充代码构成**

  shellcode, 返回地址, 填充数据

- **shellcode**

  完成特殊任务的自包含的二进制代码

  shellcode一般用于获得系统特殊权限

shellcode

- **Shellcode种类　　exec()及系统调用**

系统调用的函数名称	完成的功能
open()	读文件
open()，create()，link()，unlink()	写文件
fork()	创建进程
system()，popen()	执行程序
socket()，connect()，send()	访问网络
chmod()，chown()	改变文件属性
setuid()，getuid()	改变权限限制

shellcode

- **Shellcode** **不同硬件平台的shellcode不一样**

exec(“/bin/sh”)

```
char shellcode[] =
“\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x4
6\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4
e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd
8\x40\xcd\x80\xe8\xdc\xff\xff\xff”
```

shellcode

- **返回地址**

  指shellcode的入口地址

  借助内存分配的规律（栈从0xbfffffff开始）

  不同的返回地址均处于某个较小的地址区间

  为了提高成功率，往往使用重复地址内容

shellcode

- **填充数据**

  辅助shellcode和返回地址

  一般使用对执行结果没有影响的操作

  使用最多的是NOP，值为0x90

代码植入的构造类型

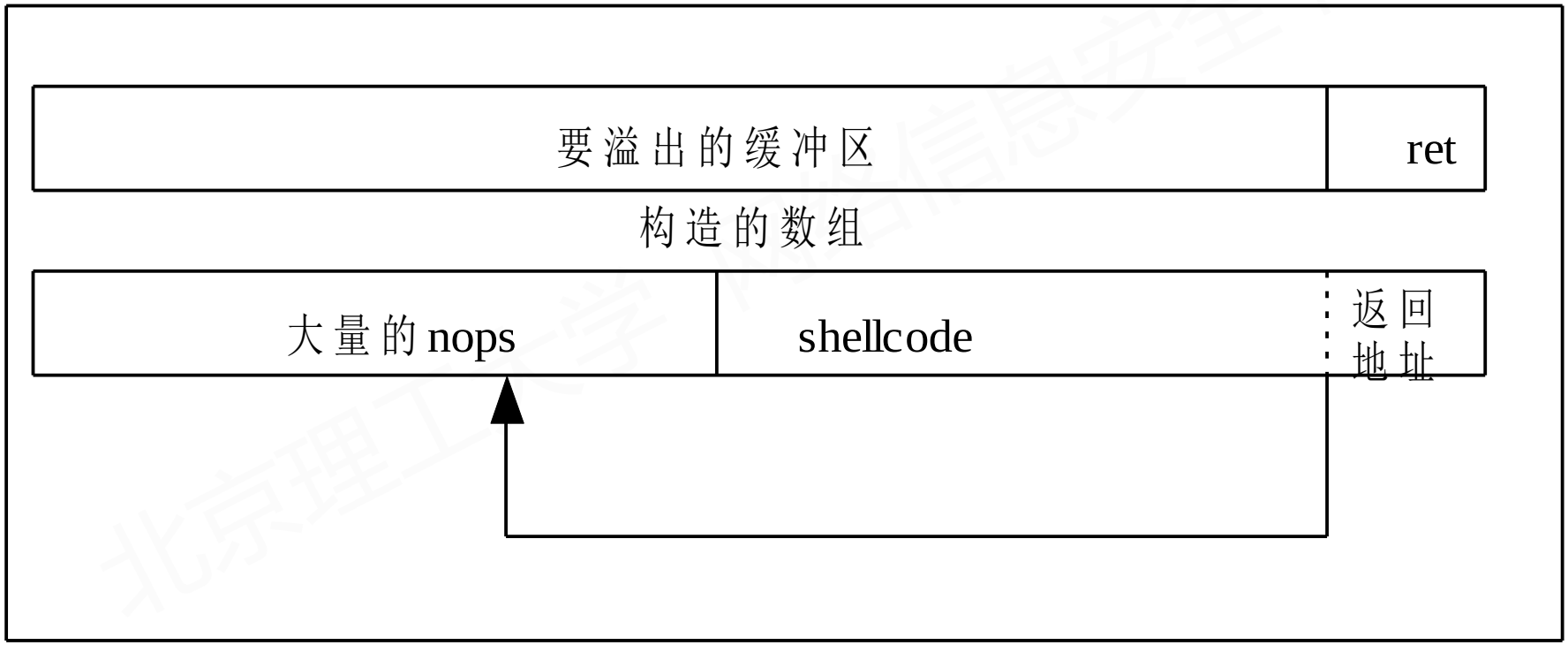
- **多种模式，常用：**
  - **NSR模式**
  - **RNS模式**
  - **AR模式**

N: 填充数据　　**S: shellcode**

R: 返回地址　　**A: 环境变量**

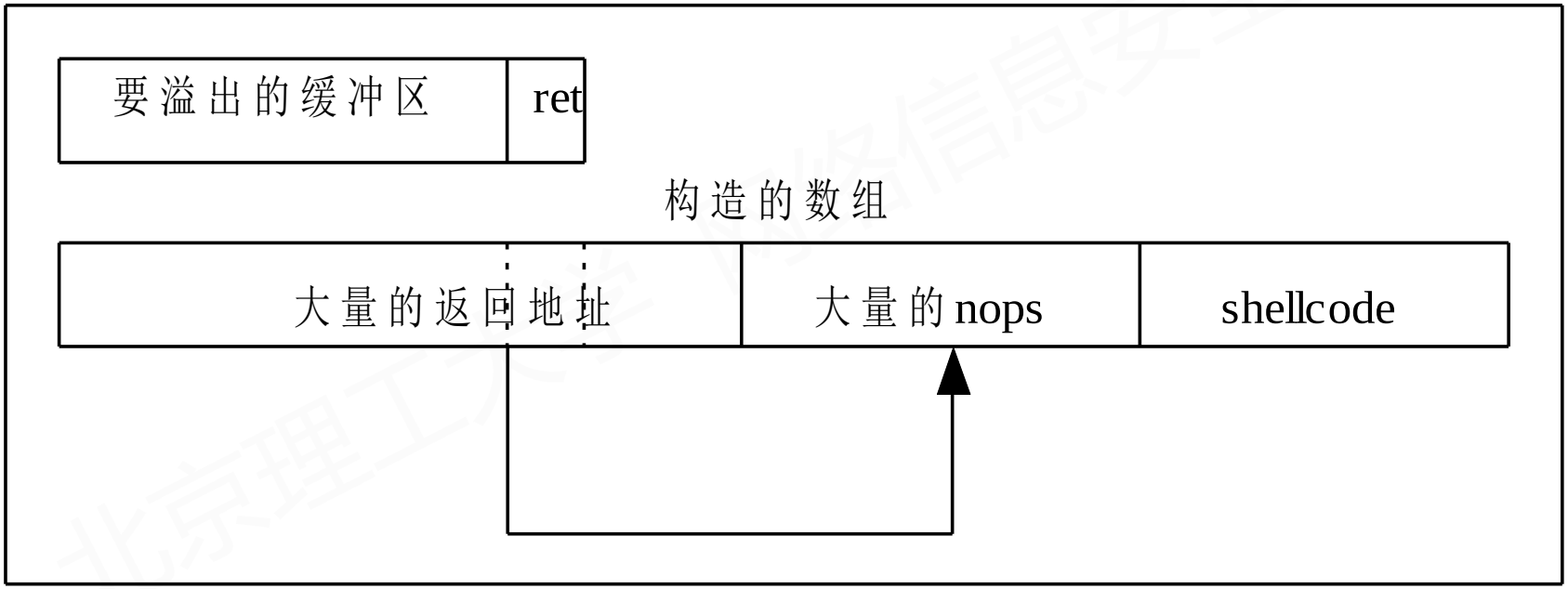
代码植入的构造类型

- **NSR模式**



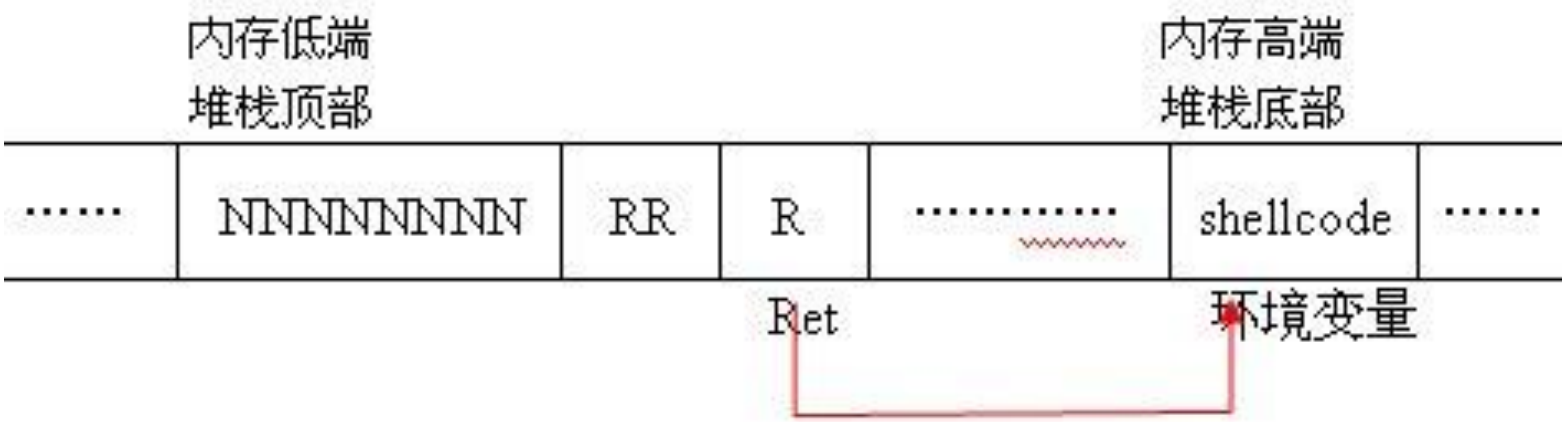
代码植入的构造类型

- **RNS模式**



代码植入的构造类型

- **AR模式**



必须事先将shellcode放置到环境变量中

本节总结

- 经过本节的学习，我们知道
  - 缓冲区溢出的基本原理
  - 栈溢出、堆溢出、BSS溢出
  - 漏洞利用及Shellcode