南京郵電大學
Nanjing University of Posts and Telecommunications

课后作业

1. PE文件格式是什么？在WINDOWS系统中有哪些类型的文件属于PE文件？

- PE的意思就是Portable Executable(可移植、可执行)，它是Win32可执行文件的标准格式

- Win32可执行文件，如*.EXE、*.DLL、*.OCX等，都是PE格式

2. 简述WINDOWS平台下病毒运行的一般流程

- 一般来说，病毒往往先于HOST程序获得控制权。运行Win32病毒的一般流程示意如下：
- ①用户点击或系统自动运行HOST程序；
- ②装载HOST程序到内存；
- ③通过PE文件中的AddressOfEntryPoint+ImageBase，定位第一条语句的位置(程序入口)；
- ④从第一条语句开始执行(这时执行的其实是病毒代码)；
- ⑤病毒主体代码执行完毕，将控制权交给HOST程序原来的入口代码；
- ⑥HOST程序继续执行。

3. 怎么样获得PE文件中重要数据结构?

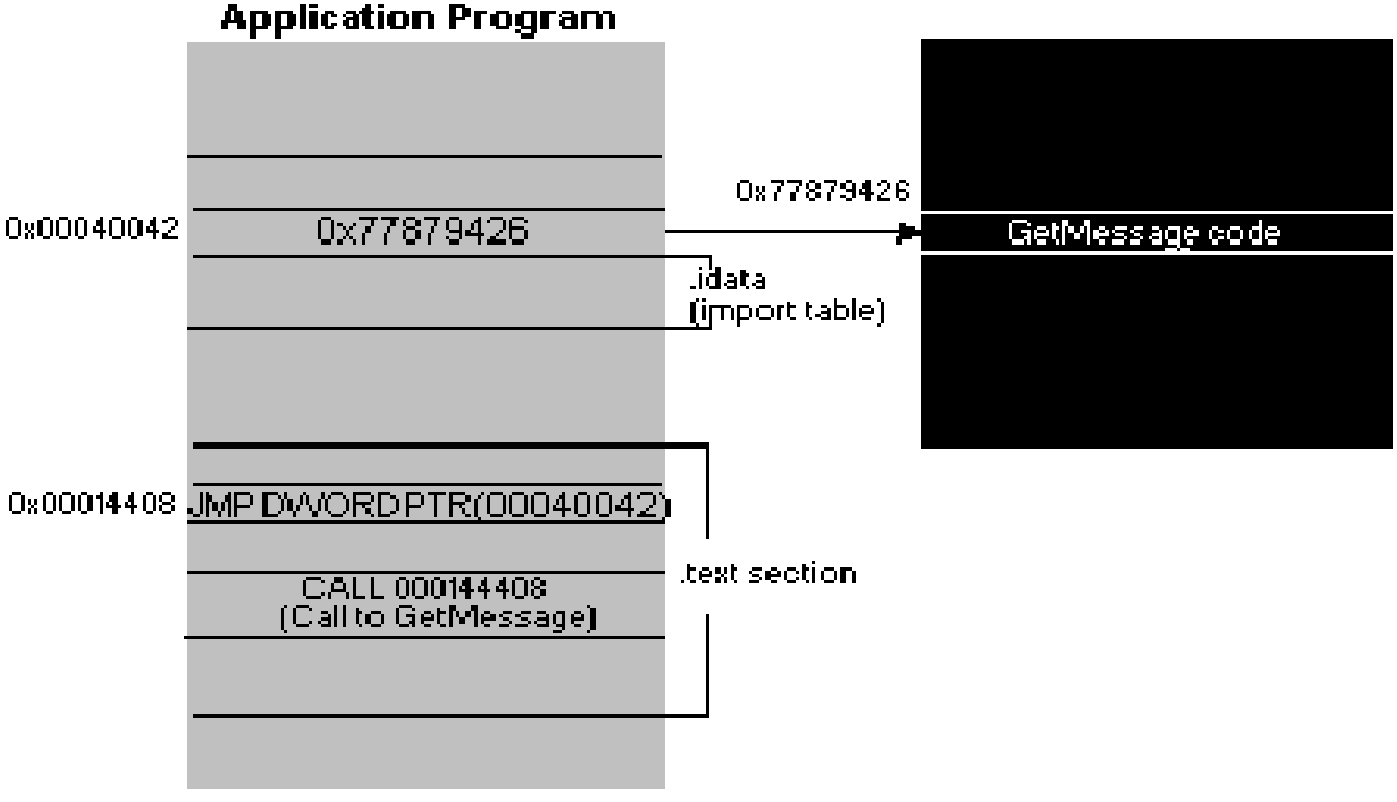
1)从 DOS header 定位到 PE header

2)从 optional header 读取 data directory 的地址。

3)IMAGE_DATA_DIRECTORY 结构尺寸乘上找寻结构的索引号: 寻import symbols的位置信息，必须用IMAGE_DATA_DIRECTORY 结构尺寸(8 bytes)乘上1（import symbols在data directory中的索引号）。

4)将上面的结果加上data directory地址，就得到包含所查询数据结构信息的IMAGE_DATA_DIRECTORY 结构项

4. 一个程序调用外部**DLL**中的函数时并不直接调用那个**DLL**中的函数。相反，**CALL**指令转到了同一个**.text**节中的**JMP DWORD PTR [XXXXXXXX]**类型的指令。



5. DLL/EXE要引出一个函数给其他DLL/EXE使用，有两种实现方法

- 通过函数名引出
- 比如某个DLL要引出名为"GetSysConfig"的函数，如果它以函数名引出，那么其他DLLs/EXEs若要调用这个函数，必须通过函数名-GetSysConfig。
- 仅仅通过序数引出
- 序数是唯一指定DLL中某个函数的16位数字，在所指向的DLL里是独一无二的。例如在上例中，DLL可以选择通过序数引出，假设是16，那么其他DLLs/EXEs若要调用这个函数必须以该值作为GetProcAddress调用参数。这就是所谓的仅仅靠序数引出。

反调试技术简介（续）

本次课程支撑的毕业要求指标点

- 毕业要求5-3：

能够分析比较所使用的技术、资源和工具的优势和不足，并理解与表述问题解决方案的局限性。

识别调试器 – 检测断点

- **4.2 检测硬件断点**

  OllyDbg的寄存器窗口按下右键，点击View debug registers可以看到DR0、DR1、DR2、DR3、DR6和DR7这几个寄存器。DR0、DR1、DR2、DR3用于设置硬件断点，由于只有4个硬件断点寄存器，所以同时最多只能设置4个硬件断点。DR4、DR5由系统保留。如果没有硬件断点，那么DR0、DR1、DR2、DR3这4个寄存器的值都为0。 ·

```
BOOL CheckDebug()
{
    CONTEXT context;
    HANDLE hThread = GetCurrentThread();
    context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    GetThreadContext(hThread, &context);
    if (context.Dr0 != 0 || context.Dr1 != 0 || context.Dr2 != 0 || context.Dr3!=0)
    {
        return TRUE;
    }
    return FALSE;
}
```

识别调试器 – 时钟检测

- **4.3 时钟检测**
  - 被调试时，进程的运行速度大大降低。例如，单步调试会大幅降低程序的运行速度，所以时钟检测是检测调试器存在的最常用方式之一。
  - 原理：记录一段操作前后的时间戳，然后比较两个时间戳，如果存在滞后，则可以认为存在调试器。
  - 常用的时钟检测方法是利用rdtsc指令(操作码0x0F31)，它返回一个系统重新启动以来的时钟数，并且将其作为一个64位的值存入EAX中。运行两次rdtsc指令，然后比较两次读取之间的差值 ·

```
BOOL CheckDebug()
{
    DWORD time1, time2;
    __asm
    {
        rdtsc
        mov time1, eax
        rdtsc
        mov time2, eax
    }
    if (time2 - time1 < 0xff)
    {
        return FALSE;
    }
    else
    {
        return TRUE;
    }
}
```

识别调试器 – 父进程判断

- **4.4 父进程判定**

  一般双击运行的进程的父进程都是**explorer.exe**，但是如果进程被调试，父进程则是调试器进程。也就是说如果父进程不是explorer.exe则可以认为程序正在被调试。explorer.exe创建进程的时候会把**STARTUPINFO**结构中的值设为0，而非explorer.exe创建进程的时候会忽略这个结构中的值，也就是结构中的值不为0。所以可以利用STARTUPINFO来判断程序是否在被调试。

```
BOOL CheckDebug()
{
    STARTUPINFO si;
    GetStartupInfo(&si);
    if (si.dwX!=0 || si.dwY!=0 || si.dwFillAttribute!=0 || si.dwXSize!=0 || si.dwYSize!=0 || si.dwXCountChars!=0 || si.dwYCountChars!=0)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

干扰调试器 – TLS回调

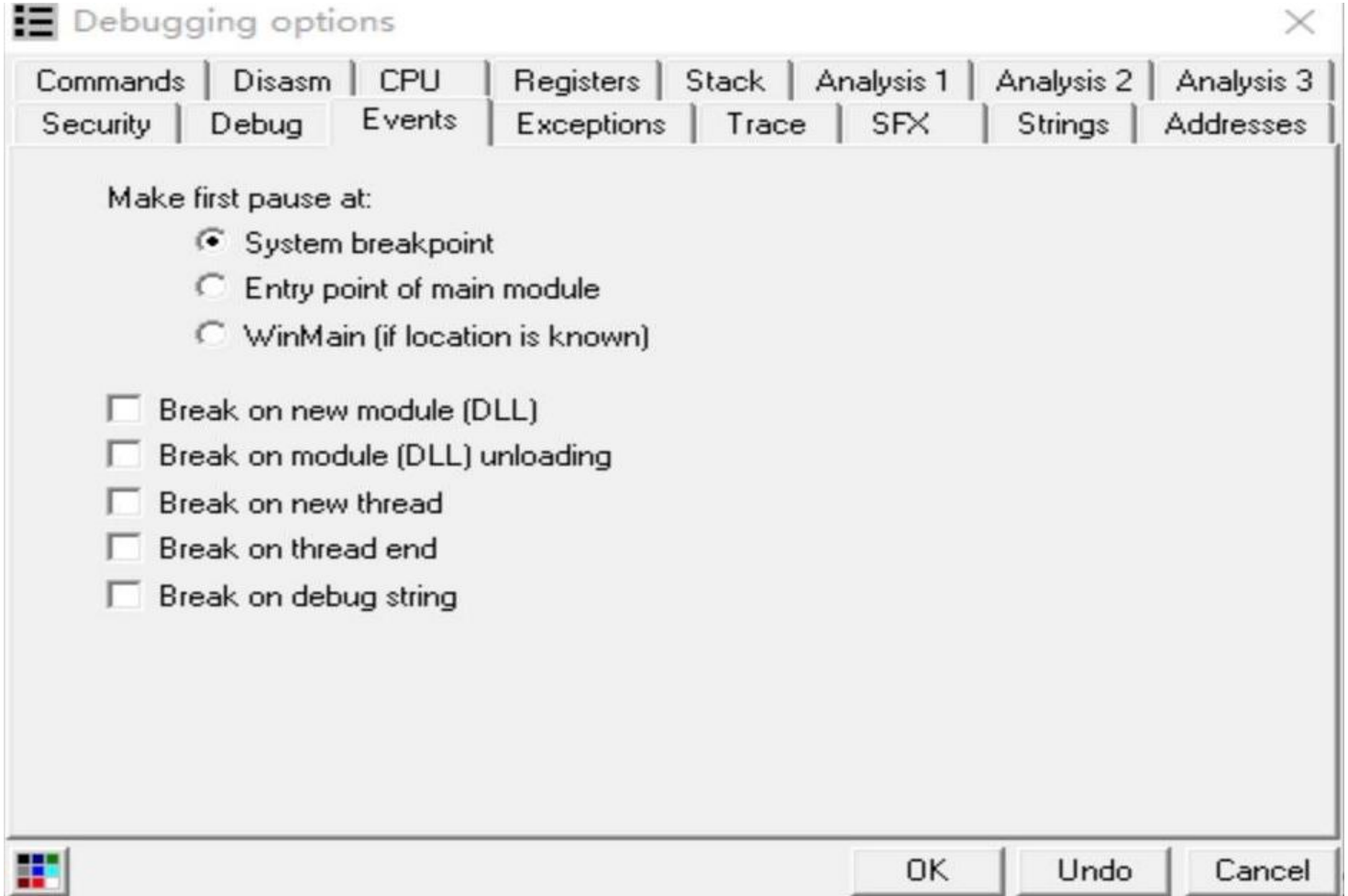
- **5.1 TLS回调**

  Thread Local Storage(TLS)，即线程本地存储，是Windows为解决一个进程中多个线程同时访问全局变量而提供的机制。TLS回调函数在进程创建时被调用，会早于进程的Main入口函数，利用这一特性进行反调试。

```
void NTAPI __stdcall TLS_CALLBACK1(PVOID DllHandle, DWORD Reason, PVOID Reserved)
{
    if (IsDebuggerPresent())
    {
        printf("TLS_CALLBACK: Debugger Detected!\n");
    }
    else
    {
        printf("TLS_CALLBACK: No Debugger Present!\n");
    }
}

int main(int argc, char* argv[])
{
    printf("233\n");
    return 0;
}
```

干扰调试器 – TLS回调


Debugging options
Commands
Disasm
CPU
Registers
Stack
Analysis 1
Analysis 2
Analysis 3
Security
Debug
Events
Exceptions
Trace
SFX
Strings
Addresses
Make first pause at:
System breakpoint
Entry point of main module
WinMain (if location is known)
Break on new module (DLL)
Break on module (DLL) unloading
Break on new thread
Break on thread end
Break on debug string
OK
Undo
Cancel

干扰调试器 – 利用中断

- **5.2 利用中断**
  - 双字节操作码0xCD03也可以产生INT 3中断，这是干扰WinDbg调试器的有效方法。在调试器外，0xCD03指令产生一个STATUS_BREAKPOINT异常。
  - 然而在WinDbg调试器内，由于断点通常是单字节机器码0xCC，因此WinDbg会捕获这个断点然后将EIP加1字节。这可能导致程序在被正常运行的WinDbg调试时，执行不同的指令集。（OllyDbg可以避免双字节INT 3的攻击）

```
BOOL CheckDebug()
{
    __try
    {
        __asm
        {
            __emit 0xCD
            __emit 0x03
        }
    }
    __except(1)
    {
        return FALSE;
    }
    return TRUE;
}
```

SEH结构体异常处理

- 异常处理结构体（Structure Exception Handler，SEH）是Windows异常处理机制所采用的的重要数据结构

- 每个SEH结构体包含两个DWORD指针：SEH链表指针和异常处理函数句柄

- 当GUI应用程序触发一个消息时，系统将把该消息放入消息队列，然后去查找并调用窗体的回调函数，即消息处理函数

- 与之类似，异常也可视为一种消息，应用程序发生异常时就触发了该消息，系统会将异常放入SEH结构体中，调用它的回调函数，即异常处理函数

干扰调试器 – 陷阱标志位

- **5.3 陷阱标志位**

  EFLAGS寄存器的第八个比特位是陷阱标志位。如果设置了，就会产生一个单步异常。

.

```
BOOL CheckDebug()
{
    __try
    {
        __asm
        {
            pushfd
            or word ptr[esp], 0x100
            popfd
            nop
        }
    }
    __except(1)
    {
        return FALSE;
    }
    return TRUE;
}
```

干扰调试器 – 陷阱标志位

- **5.4 RaiseException函数**

  RaiseException函数产生的若干不同类型的异常可以被调试器捕获。只有当没有附加调试器的时候，异常处理程序才会执行。

```
BOOL TestExceptionCode(DWORD dwCode)
{
    __try
    {
        RaiseException(dwCode, 0, 0, 0);
    }
    __except(1)
    {
        return FALSE;
    }
    return TRUE;
}

BOOL CheckDebug()
{
    return TestExceptionCode(DBG_RIPEXCEPTION);
}
```

花指令简介

反汇编基本算法浅析

1. 线性扫描反汇编算法：

从程序的入口点开始反汇编，然后对整个代码段进行扫描，反汇编扫描过程中所遇到的每条指令。线性扫描算法的缺点在于在冯诺依曼体系结构下，无法区分数据与代码，从而导致将代码段中嵌入的数据误解释为指令的操作码，以致最后得到错误的反汇编结果。

反汇编基本算法浅析

2. 行进递归反汇编算法：

相比线性扫描算法，行进递归算法通过程序的控制流来确定反汇编的下一条指令，遇到非控制转移指令时顺序进行反汇编，而遇到控制转移指令时则从转移地址处开始进行反汇编。行进递归算法的缺点在于准确确定间接转移目的地址的难度较大。

花指令介绍

花指令是程序中的无用指令或者垃圾指令，故意干扰各种反汇编静态分析工具，但是程序不受任何影响，缺少了它也能正常运行

加花指令后，IDA Pro等分析工具对程序静态反汇编时，往往会出现错误或者遭到破坏，加大逆向静态分析的难度，从而隐藏自身的程序结构和算法，从而较好的保护自己

垃圾指令类 - 不可执行

```
jmp Label1

  db thunkcode1;垃圾数据

Label1:

  ......
```

1. jmp可以用call，ret，loop等替换；
2. 该垃圾数据通常是一条多字节指令的操作码，例如在thunkcode1处放入0e8h，由于0e8h是call指令的操作码，因此对0e8h进行解码时就会将它后面的4个字节看作是调用目标地址，从而造成反汇编过程中的错误，达到隐藏恶意代码的目的。

垃圾指令类 - 不可执行 - 多节

```
JMP Label1
Db thunkcode1
Label1:
.....
JMP Label2
Db thunkcode2
Label2:
.....
```

典型形式的条件跳转混淆可能仅仅造成几条指令的反汇编错误，但是多重顺序嵌套的条件跳转混淆则能够使更多指令的反汇编得到错误的结果

垃圾指令类 - 不可执行 - 多层

```
JMP Label1
  Db thunkcode1
Label2:
  ……
  JMP Label3
  Db thunkcode3
Label1:
  …….
  JMP Label2
  Db thunkcode2
Label3:
  ……
```

在上面的形式中，可以将条件跳转到跳转目的地址之间的所有字节进行填充来破解混淆，于是有了条件混淆的一种新的形式，即多层乱序嵌套

花指令使用实例

```
1  void flower_code()
2  {
3          __asm
4          {
5                  jz label;
6                  jnz label;
7                  //相当于汇编中的db
8                  __emit 0e8h;
9  label:
10                 mov ax, 8;
11                 xor ax, 7;
12         }
13 }
14
15
```

```
Address: flower_code(void)
Viewing Options
0120353B 57              push    edi
0120353C 8D BD 40 FF FF FF  lea     edi,[ebp-0C0h]
01203542 B9 30 00 00 00     mov     ecx,30h
01203547 B8 CC CC CC CC     mov     eax,0CCCCCCCCh
0120354C F3 AB           rep stos   dword ptr es:[edi]
   __asm
   {
      jz label;
0120354E 74 03           je      label (1203553h)
      jnz label;
01203550 75 01           jne     label (1203553h)
      //相当于汇编中的db
      __emit 0e8h;
01203552 E8 66 B8 08 00  call    0128EDBD
      xor ax, 7;
01203557 66 83 F0 07        xor     ax,7
   }
}
0120355B 5F              pop     edi
0120355C 5E              pop     esi
0120355D 5B              pop     ebx
0120355E 81 C4 C0 00 00 00  add     esp,0C0h
```

南京郵電大學
Nanjing University of Posts and Telecommunications

手动脱壳简述

寻找程序入口点（Original Entry Point，OEP）

- 软件加壳就是隐藏了OEP（或者用了假的OEP/花指令等，例如直接转到ExitProcess等处），只要找到程序真正的OEP，可以实现脱壳。一般的查壳工具无法直接识别出OEP

- 壳实质上是一个子程序，它在程序运行时首先取得控制权并对程序进行压缩，同时隐藏程序真正的OEP

- ESP定律：即堆栈平衡定律，是应用频率最高的脱壳方法之一，可以应对简单的压缩壳(壳的种类可以由PEID等工具进行分析）。最后一次异常等方法也常用于识别OEP

ESP定律

• 在程序自解压过程中,多数壳会先将当前寄存器状态压栈,如使用PUSHAD,而在解压结束后,会将之前的寄存器值出栈,如使用POPAD 。

• 基于PUSHAD和POPAD的对称性，可以利用硬件断点定位真正的OEP：当壳把代码解压前和解压后，必须要平衡堆栈，让执行到OEP的时候，使ESP=0012FFC4。这就是ESP定律

• 例如，PUSHAD的时候将寄存器值压入了0012FFC0到0012FFA4的堆栈中。通过在0012FFA4下硬件断点，等POPAD恢复堆栈，即可停在OEP处

手动脱壳

- 定位到OEP之后，使用LordPE等工具把程序的镜像dump出来

- 但dump得到的镜像无法运行，因为无法自动获取导入函数的地址

- 为此，需要修复导入函数地址表（Import Address Table, IAT）。通常使用importrec工具进行修复，从原文件（加壳的文件）提取信息后，对脱壳文件进行修复

Stolen Code

- 某些壳在处理OEP代码的时候，把OEP处固定的代码NOP掉，然后把这些代码（即stolen code）放到壳代码的空间中去，而且常伴随着花指令，使原程序的起始代码从壳空间开始执行，然后再JMP回原程序空间

- 如果脱掉壳,这一部分代码就会遗失,也就达到了反脱壳的目的。 这就是stolen code技术

- 如果dump以后修复IAT，这里OEP依旧是错误的，程序无法运行

- 原因：前面几行代码被放在壳空间中，所以不会被转储，因此也得不到执行

- 解决方法：寻找真正的OEP，找到缺失的代码

缓冲区溢出

什么是缓冲区溢出？

- 缓冲区是一块连续的计算机内存区域，可保存相同数据类型的多个实例
- 缓冲区可以是堆栈(自动变量)、堆(动态内存)和静态数据区(全局或静态)
- 在C/C++语言中，通常使用字符数组和malloc/new之类内存分配函数实现缓冲区。
- 缓冲区溢出指数据被添加到分配给该缓冲区的内存块之外，是最常见的程序缺陷

什么是缓冲区溢出？

- 栈帧结构的引入为高级语言中实现函数或过程调用提供直接的硬件支持，但由于将函数返回地址这样的重要数据保存在程序员可见的堆栈中，因此也给系统安全带来隐患

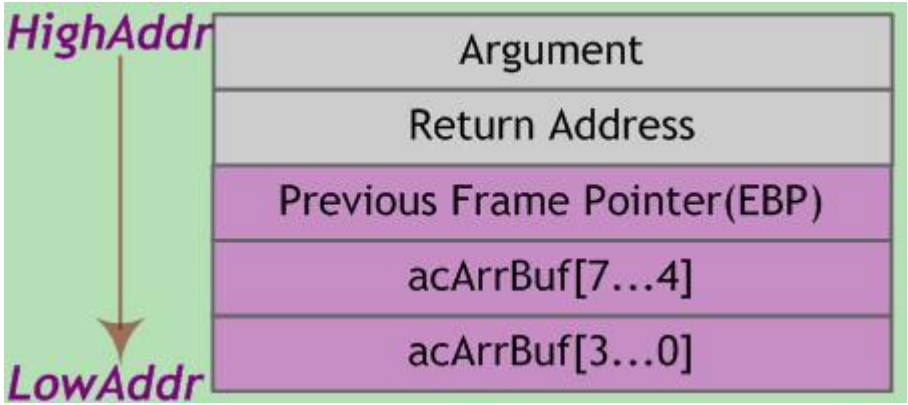
- 若将函数返回地址修改为指向一段精心安排的恶意代码，则可达到危害系统安全的目的。此外，堆栈的正确恢复依赖于压栈的EBP值的正确性，但EBP域邻近局部变量，若编程中有意无意地通过局部变量的地址偏移窜改EBP值，则程序的行为将变得非常危险。

缓冲区溢出攻击

- 由于C/C++语言没有数组越界检查机制，当向局部数组缓冲区里写入的数据超过为其分配的大小时，就会发生缓冲区溢出。

- 攻击者可利用缓冲区溢出来窜改进程运行时栈，从而改变程序正常流向，轻则导致程序崩溃，重则系统特权被窃取。

缓冲区溢出攻击



- 若将长度为16字节的字符串赋给acArrBuf数组，则系统会从acArrBuf[0]开始向高地址填充栈空间，导致覆盖EBP值和函数返回地址

- 若攻击者用一个有意义的地址覆盖返回地址的内容，函数返回时就会去执行该地址处事先安排好的攻击代码
  - 最常见的手段是通过制造缓冲区溢出使程序运行一个用户shell，再通过shell执行其它命令。若该程序有root或suid执行权限，则攻击者就获得一个有root权限的shell，进而可对系统进行任意操作

感谢大家！

南京郵電大學
Nanjing University of Posts and Telecommunications