



南京邮电大学  
Nanjing University of Posts and Telecommunications

# ASLR技术

# 本次课程支撑的毕业要求指标点

- 毕业要求4-3:

针对设计或开发的解决方案，能够基于信息安全领域科学原理对其进行研究，并能够通过理论证明、实验仿真或者系统实现等多种科学方案说明其有效性、合理性，并对解决方案的实施质量进行分析，通过信息综合得到合理有效的结论

# ASLR技术

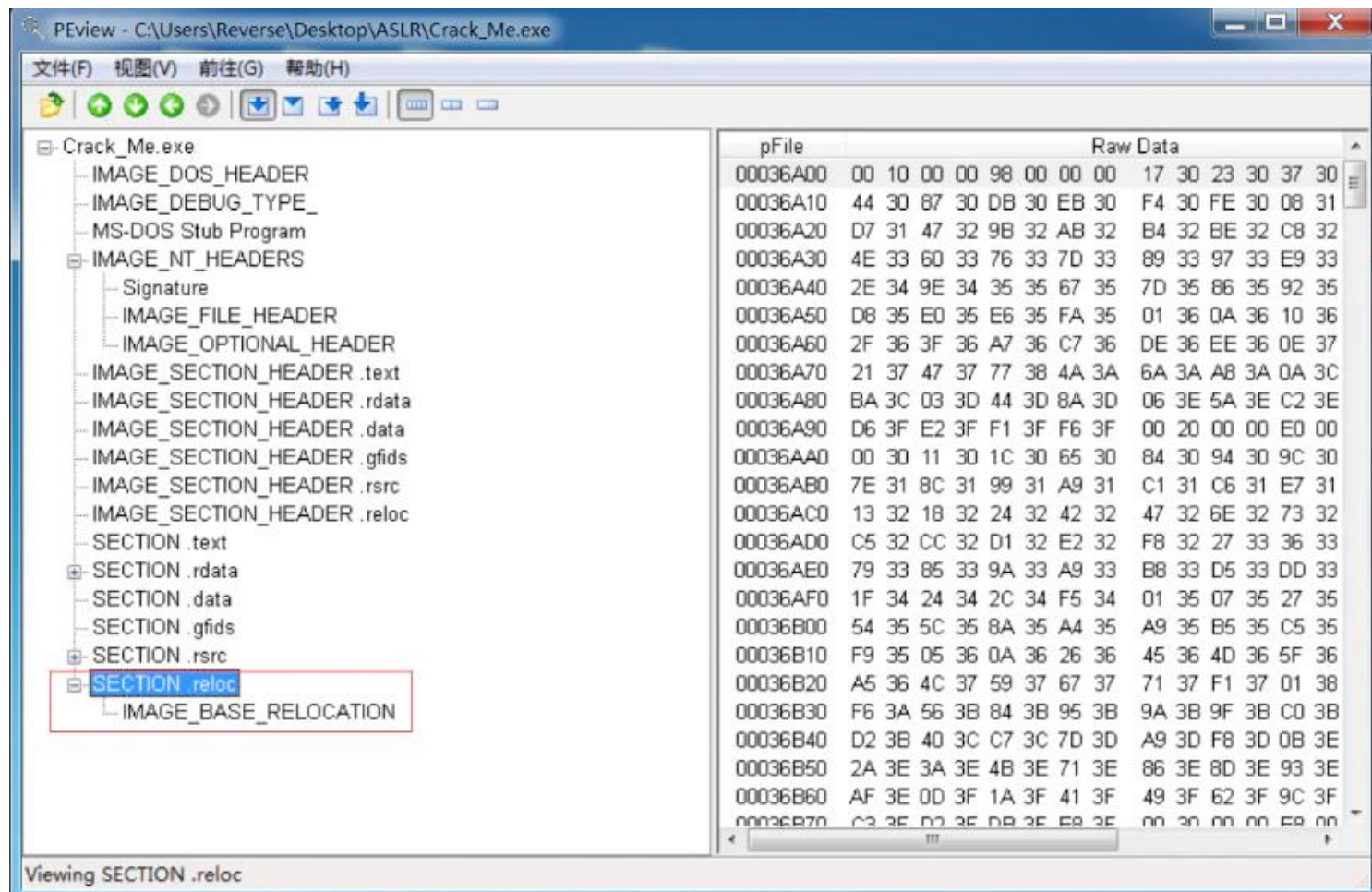
- Address space layout randomization, 地址随机化技术
- 微软从windows vista/windows server 2008 (kernel version 6.0) 开始采用ASLR技术, 主要目的是为了防止缓冲区溢出
- ASLR技术会使PE文件每次加载到内存的起始地址随机变化, 并且进程的栈和堆的起始地址也会随机改变

## ASLR技术

- 该技术需要操作系统和编译工具的双重支持（前提是操作系统的支持，编译工具主要作用是生成支持ASLR的PE格式）
- 若不想使用ASLR功能，以VS为例，可以在VS编译的时候将“配置属性->链接器->高级->随机基址”的值修改为否即可

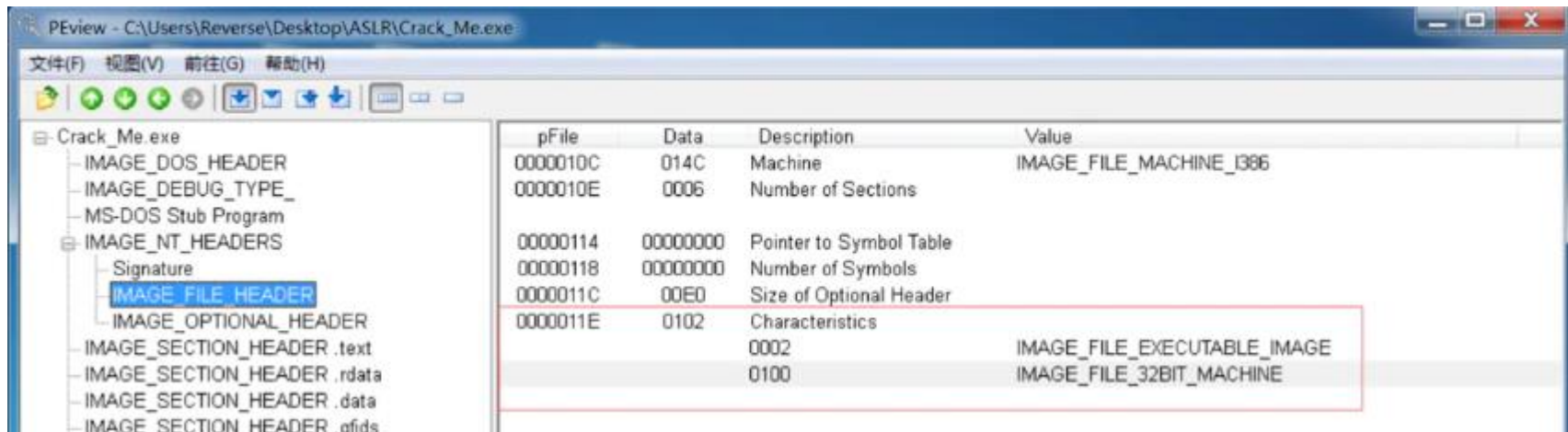
# 开启ASLR的文件

- 多了一个.reloc节



# 开启ASLR的文件

- IMAGE\_FILE\_HEADER\Characteristics不同
- 开启ASLR的程序的Characteristic(0102) =  
IMAGE\_FILE\_EXECUTABLE\_IMAGE(0002) | IMAGE\_FILE\_32BIT\_MACHINE(0100)

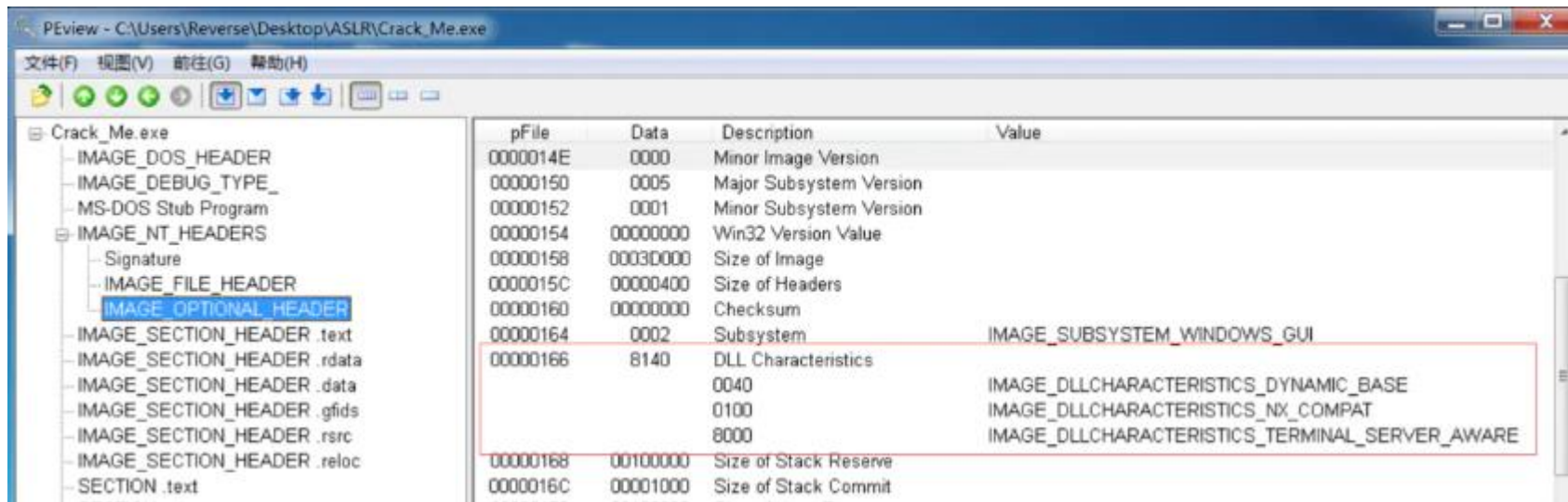


# 开启ASLR的文件

- 未开启ASLR的程序Characteristic(0103) =  
IMAGE\_FILE\_RELOCS\_STRIPPED(0001) |  
IMAGE\_FILE\_EXECUTE\_IMAGE(0002) | IMAGE\_FILE\_32BIT\_MACHINE  
(0100)
- 未开启ASLR的程序多一个 IMAGE\_FILE\_RELOCS\_STRIPPED字段值，该字段值的含义为：已从文件中剥离重定位信息，必须在其首选基地址加载该文件，如果基地址不可用，加载程序将报告错误

# 开启ASLR的文件

- IMAGE\_OPTIONAL\_HEADER\DllCharacteristics不同
- 开启ASLR的程序比未开启ASLR的程序的DllCharacteristics多了一个字段IMAGE\_DLLCHARACTERISTICS\_DYNAMIC\_BASE，意思是该DLL可以在加载时重定向





## 删除ASLR功能

- 有时候，需要分析的程序只能在较高版本的WINDOWS系统上运行，如win7, server2008等
- 然而这些平台都使用了ASLR技术
- 每次OD载入时，其映像基址都是会变化的。在分析过程中有时候需要计算一些地址，基址的变化会带来困扰
- 怎么办？

## 删除ASLR功能

- 从系统方面关闭ASLR功能
  - 添加一个DWORD键值项  
HKLM\System\CurrentControlSet\Control\SESSION  
MANAGER\MEMORY MANAGEMENT\MoveImages, 其值为0; 重启系统即可
  - 打开ASLR就是删除掉MoveImages键值项

## 删除ASLR功能

- 将IMAGE\_OPTIONAL\_HEADER\DllCharacteristic中的IMAGE\_DLLCHARACTERISTICS\_DYNAMIC\_BASE字段值去掉即可：将PE中8140数据改为8100

00000140	00	10	00	00	00	02	00	00	05	00	01	00	00	00	00	.....
00000150	05	00	01	00	00	00	00	00	00	D0	03	00	00	04	00	.....D.....
00000160	00	00	00	00	02	00	40	81	00	00	10	00	00	10	00	.....@.....
00000170	00	00	10	00	00	10	00	00	00	00	00	00	10	00	00	.....
00000180	00	00	00	00	00	00	00	00	AC	E5	01	00	3C	00	00	.....

修改前

0000140	00	10	00	00	00	02	00	00	05	00	01	00	00	00	00	.....
0000150	05	00	01	00	00	00	00	00	00	D0	03	00	00	04	00	.....D.....
0000160	00	00	00	00	02	00	00	81	00	00	10	00	00	10	00	.....
0000170	00	00	10	00	00	10	00	00	00	00	00	00	10	00	00	.....
0000180	00	00	00	00	00	00	00	00	AC	E5	01	00	3C	00	00	.....

修改后

## 删除ASLR功能

- 通过编译工具入手
  - 若不想使用ASLR功能，可以在VS编译的时候将“配置属性->链接器->高级->随机基址”的值修改为否即可



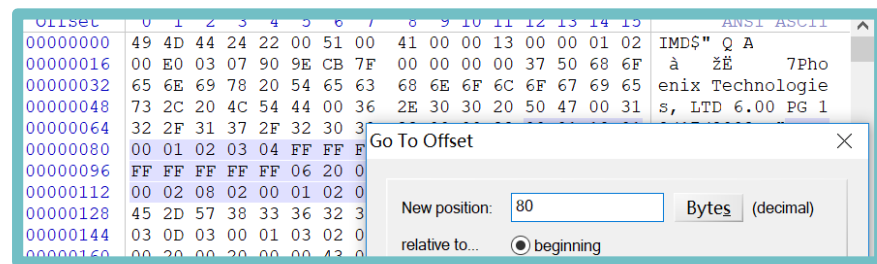
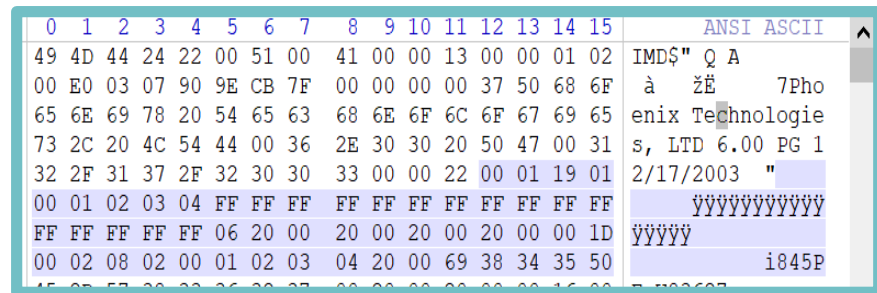
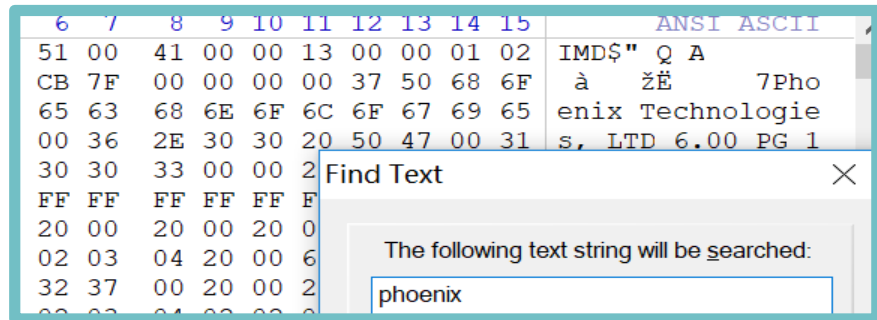
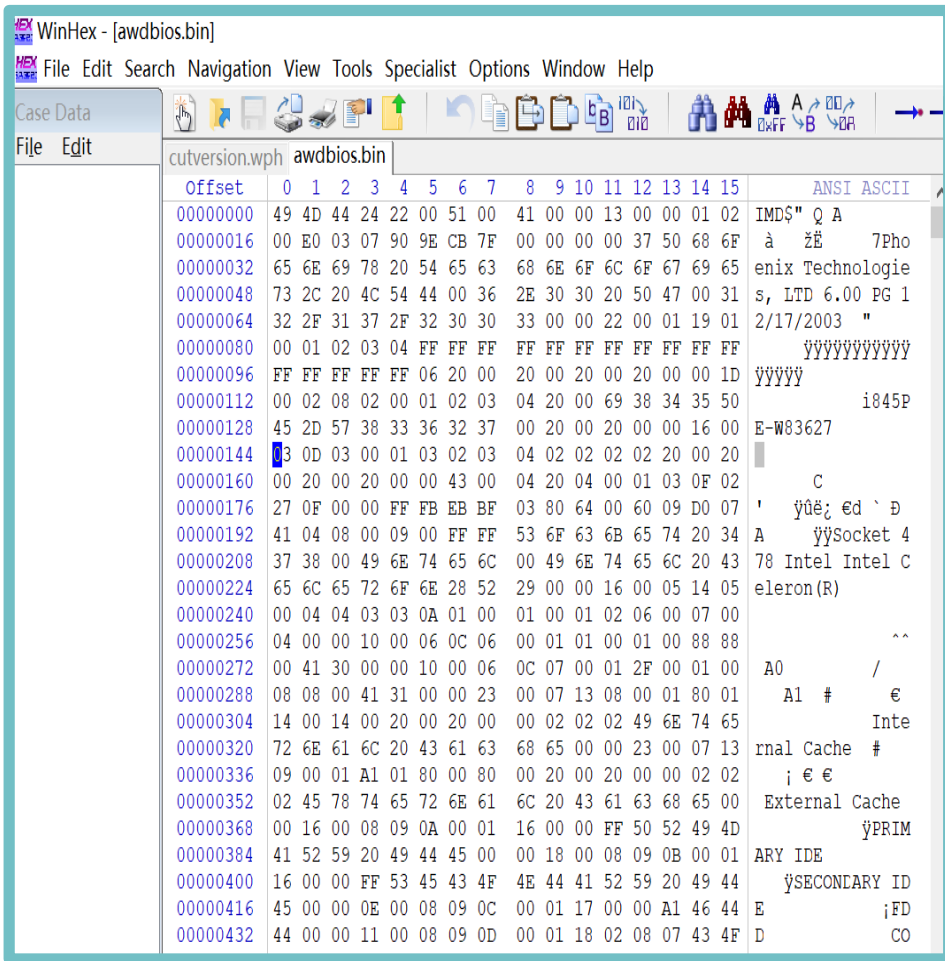
南京邮电大学  
Nanjing University of Posts and Telecommunications



# 逆向分析技术复习

南京邮电大学计算机学院信息安全系

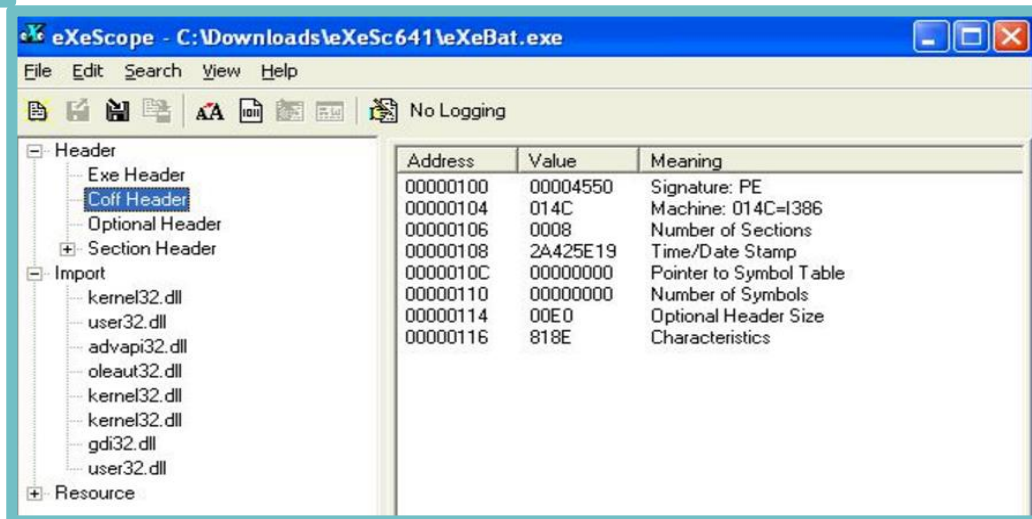
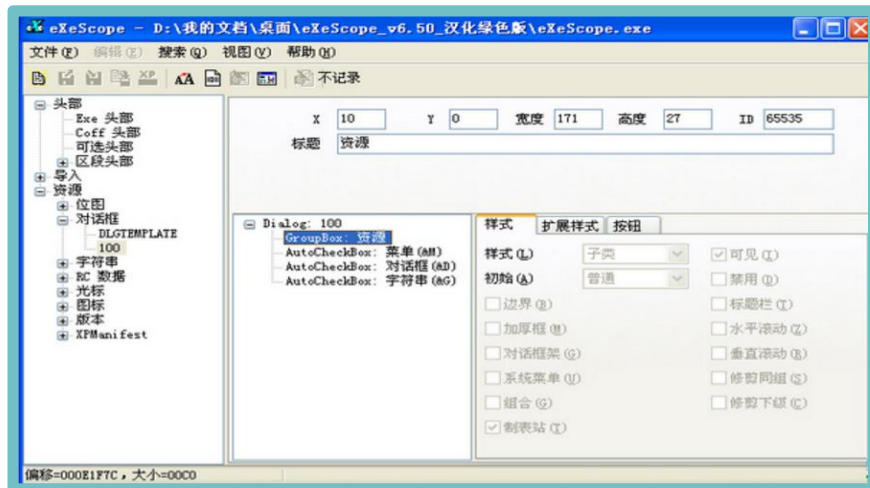
## 相关工具 - 二进制查看 - winhex/ultraedit/...



# 相关工具 - 二进制查看 - exeScope

## 文件结构框架

## 文件结构解析



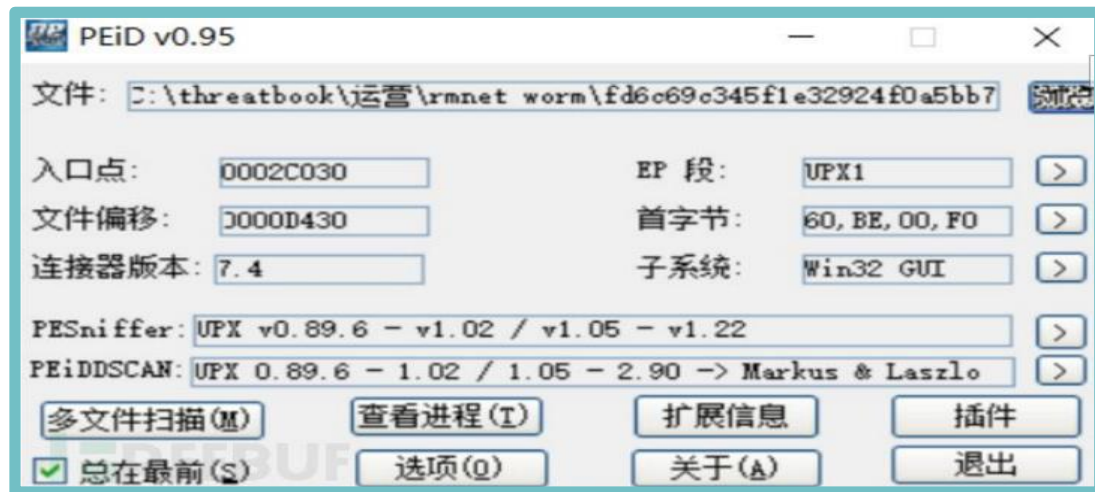


## 相关工具 - 二进制查壳 - PEID/LoadPE

识别zprotect  
(加密壳)



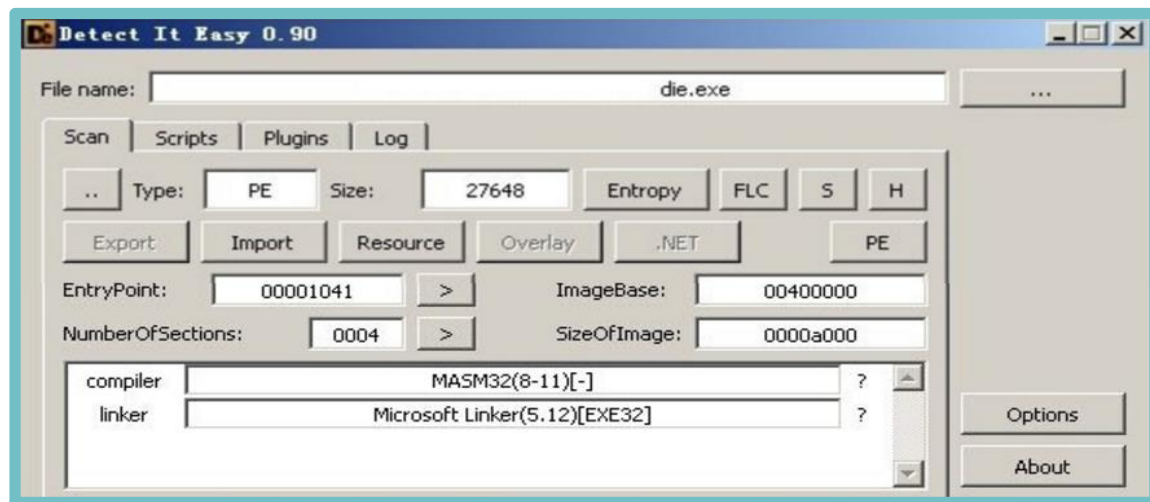
识别UPX  
(压缩壳)



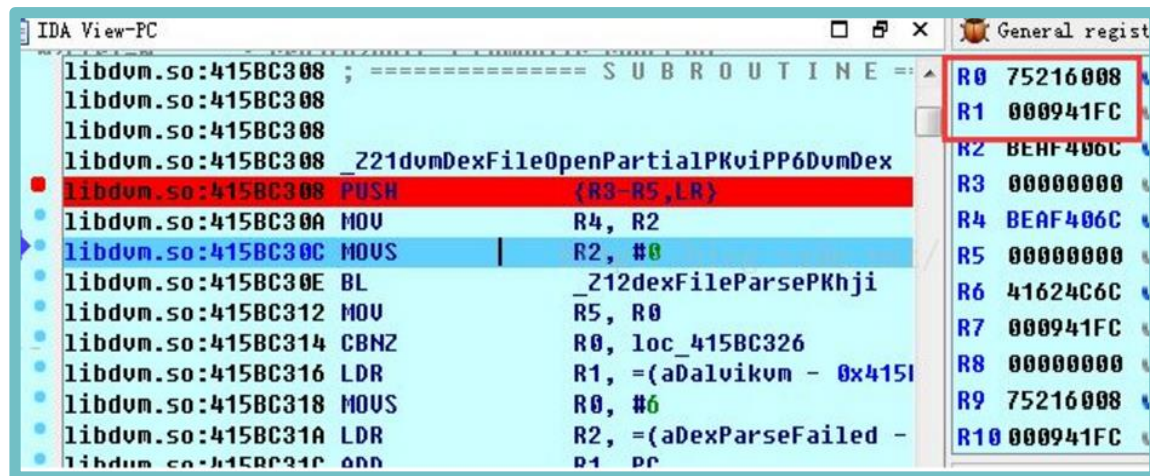


# 相关工具 - 二进制查壳 - DIE/手动分析

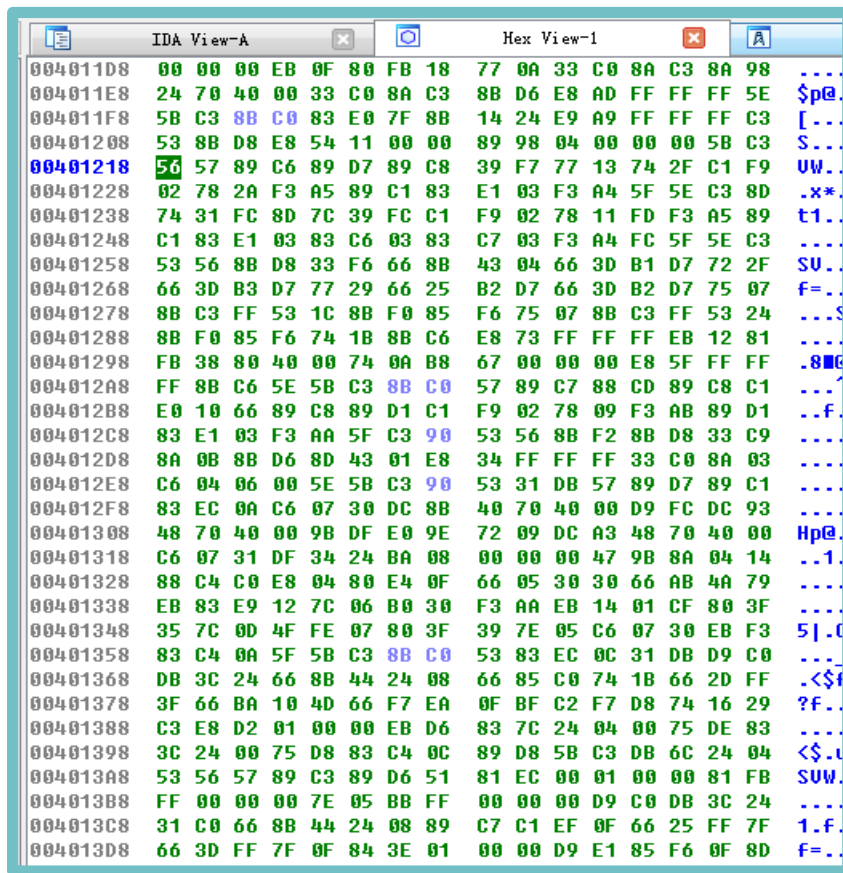
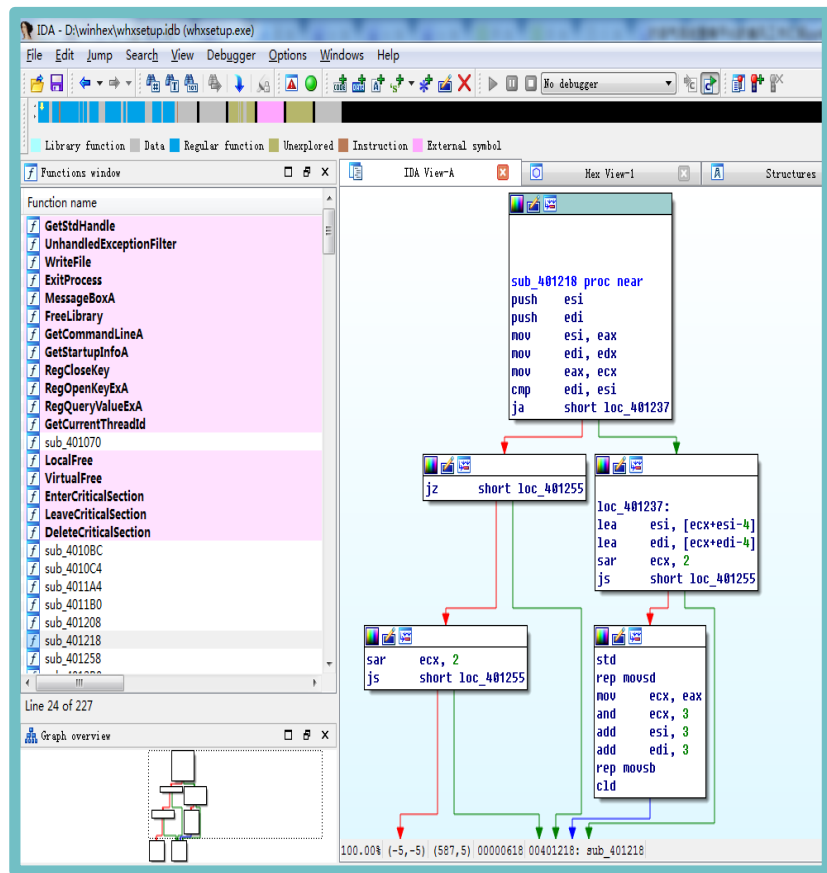
Windows/Linux/Mac  
通用



IDA手动分析

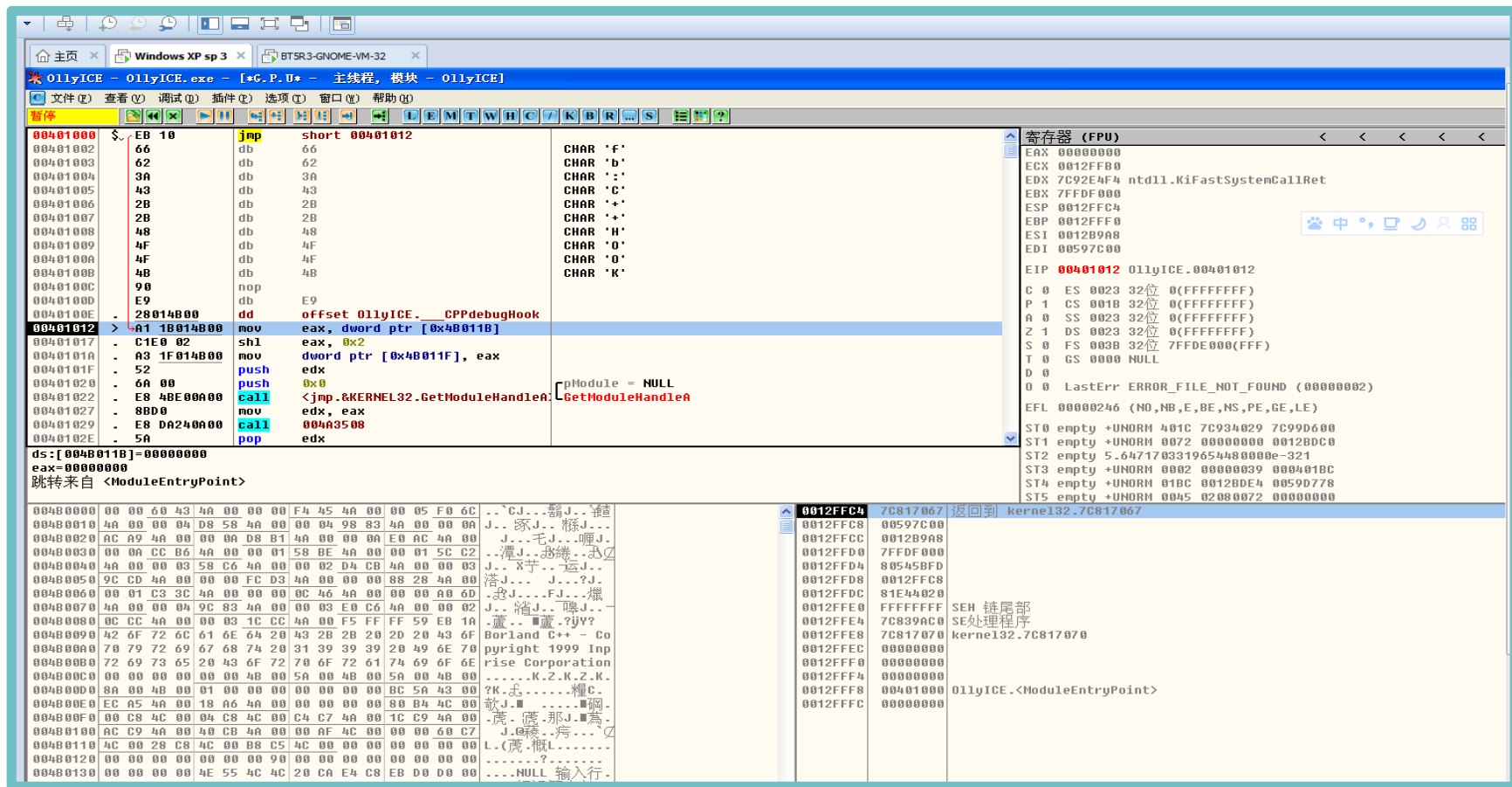


# 静态分析工具 - Windows - IDA - 应用



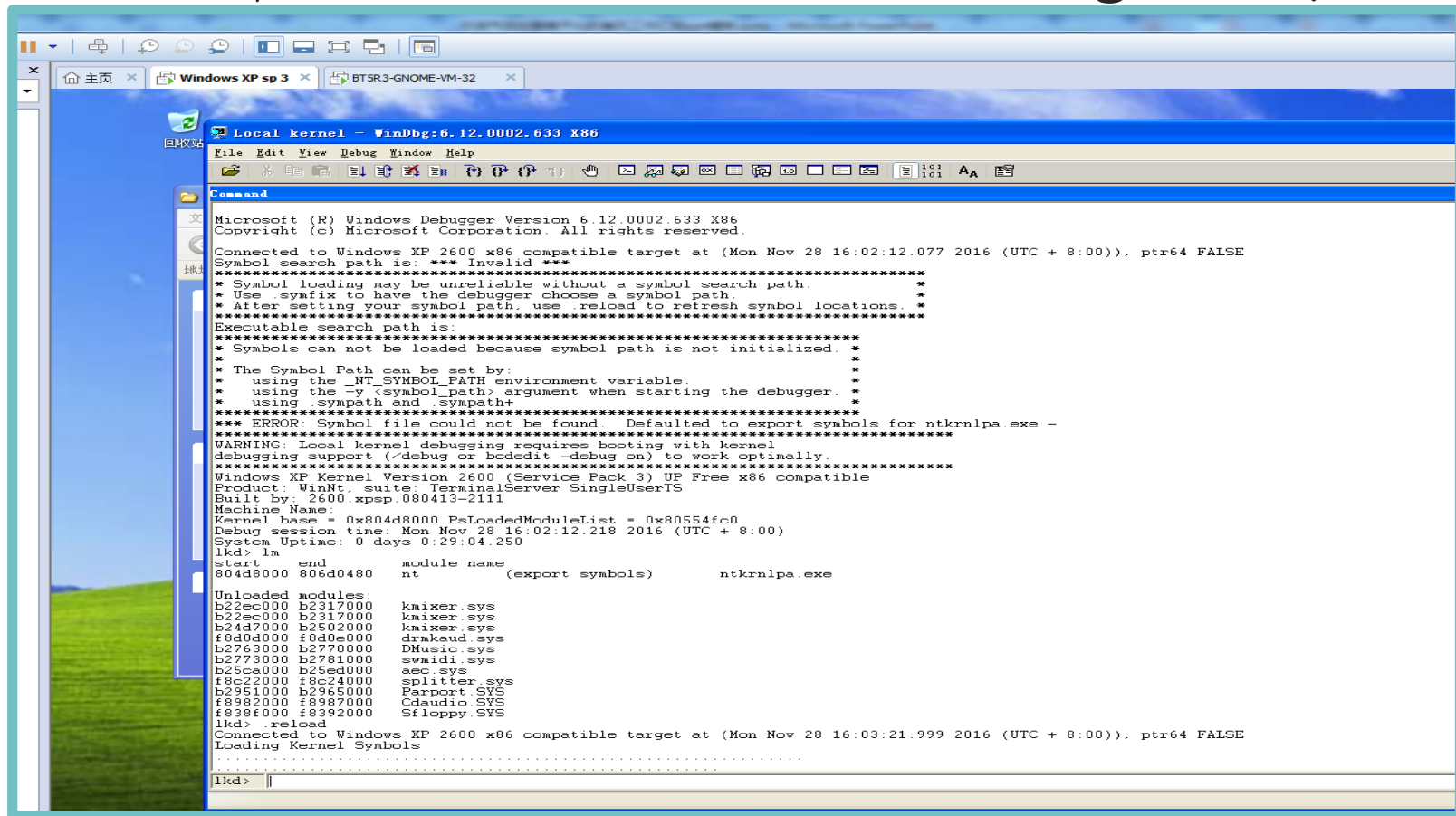
可与Vmware进行联动，调试应用程序；  
可与Qemu联动，调试固件

# 动态分析工具 - windows - OllyDbg - 应用



Windows平台下最常用的动态分析工具

# 动态分析工具 - windows - WinDbg - 内核



除了分析内核，也可以用于分析应用程序

# 动态分析工具 - Linux - gdb - 应用

```
(gdb) c
Continuing.

Breakpoint 2, main () at gdb-sample.c:21
21             printf("n=%d,nGlobalVar = %d /n", n, nGlobalVar);
(gdb) p nGlobalVar
$2 = 88
(gdb) c
Continuing.

Breakpoint 3, tempFunction (a=1, b=2) at gdb-sample.c:7
7             printf("tempFunction is called, a = %d, b = %d /n", a, b);
(gdb) p a
$3 = 1
(gdb) p b
$4 = 2
```



# 动态分析工具 - Linux - kgdb - 内核

## Re-compiling linux kernel

```
linux-kgdb:/home/linux-2.6.32.12-0.7 # ls /boot/ -l
total 37648
-rw-r--r-- 1 root root 1617387 Nov 26 08:51 System.map-2.6.32.12-0.7-default
-rw-r--r-- 1 root root 1617387 Nov 26 08:51 System.map-2.6.32.12-0.7-default.old
-rw-r--r-- 1 root root 512 Nov 24 11:05 backup_mbr
lrwxrwxrwx 1 root root 1 Nov 24 10:58 boot -> .
-rw-r--r-- 1 root root 1236 May 10 2010 boot.readme
-rw-r--r-- 1 root root 107874 May 20 2010 config-2.6.32.12-0.7-default
drwxr-xr-x 2 root root 4096 Nov 26 11:15 grub
lrwxrwxrwx 1 root root 28 Nov 26 13:40 initrd -> initrd-2.6.32.12-0.7-default
-rw-r--r-- 1 root root 13777267 Nov 26 13:40 initrd-2.6.32.12-0.7-default
-rw-r--r-- 1 root root 6572832 Nov 26 09:19 initrd-2.6.32.12-0.7-default.org
-rw-r--r-- 1 root root 435712 Nov 24 11:05 message
-rw-r--r-- 1 root root 189729 May 20 2010 sysexts-2.6.32.12-0.7-default.tar.gz
-rw-r--r-- 1 root root 495291 May 20 2010 syntypes-2.6.32.12-0.7-default.gz
-rw-r--r-- 1 root root 178468 May 20 2010 sgmlvers-2.6.32.12-0.7-default.gz
-rw-r--r-- 1 root root 3774506 May 20 2010 vmlinuz-2.6.32.12-0.7-default.gz
lrwxrwxrwx 1 root root 29 Nov 24 11:02 vmlinuz -> vmlinuz-2.6.32.12-0.7-default
-rw-r--r-- 1 root root 3205728 Nov 26 08:51 vmlinuz-2.6.32.12-0.7-default
-rw-r--r-- 1 root root 3231872 Nov 26 13:42 vmlinuz-2.6.32.12-0.7-default.old
-rw-r--r-- 1 root root 3231872 Nov 26 09:19 vmlinuz-2.6.32.12-0.7-default.org
```

## Open Kgdb options

```
root@keven-ubuntu:/home/keven/kgdb_shared# gdb vmlinux
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/keven/kgdb_shared/vmlinux...done.
(gdb) set remotebaud 115200
(gdb) target remote /dev/pts/0
Remote debugging using /dev/pts/0
kgdb_breakpoint () at kernel/kgdb.c:1718
1718 kernel/kgdb.c: 没有那个文件或目录.
(gdb) target remote /dev/pts/0
A program is being debugged already. Kill it? (y or n) y

Remote debugging using /dev/pts/0
Ignoring packet error, continuing...
```

# Reverse环境搭建 - windows/Linux工具集(1)

## 静态分析

- Winhex/UltraEdit/...
- PEID/LoadPE/DIE/...
- IDA and plugins

## 动态分析

- R3 debugging: Ollydbg/gdb (应用级调试)
- R0 debugging: windbg/kgdb, softice.. (内核级调试)
- VM images: vmware (常用于调试的动态模拟环境) /virtualBox/Qemu

## Reverse环境搭建 - windows/Linux工具集(2)

### 非主流工具

- 符号执行: Angr; Z3 ..
- 污点跟踪: Pin; Valgrind; TraintDroid ..
- 模糊测试: FileFuzz; AFL; Trinity; Peach ..
- Android专用: apktool; SMALI/BAKSMALI; Dex2JAR; JD-GUI;
- 固件专用: Qemu(动态模拟环境); KVM(底层虚拟化环境)



## Pwn环境搭建 - Linux工具集

- gdb: Linux动态调试必备
- gdb-peda: 类似的如gef, gdbinit
- pwntools: 写exp和poc的利器
- Checksec: elf程序安全性及运行平台
- Objdump/readelf: elf的关键信息
- ida pro : 必备静态反编译
- ROPgadget: 强大的rop利用工具
- one\_gadget: 快速寻找libc中的调用exec('bin/sh')的位置
- Pwntools: 必备的库, 类似的如zio
- libc-database: 通过泄露的libc中某个函数地址查出远程系统中libc版本

```
h11p@ubuntu:~/hackme$ checksec petbook
[*] '/home/h11p/hackme/petbook'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
FORTIFY:   Enabled
h11p@ubuntu:~/hackme$
```

# OllyDbg工具的使用 - 基本用法

**F2**: 设置断点，在光标位置按**F2**键即可，再按则会删除断点

**F3**: 打开文件

**F4**: 运行到选定位置。作用就是直接运行到光标所在位置处暂停

**F7**: 单步步入。功能同(**F8**)，遇到 **CALL** 等子程序时会进入其中

**F8**: 单步步过。每按一次执行一条指令，遇到 **CALL** 等子程序不进入

**F9**: 运行。被调试的程序将直接开始运行

**ALT+F9**: 执行到用户代码。可用于从系统领空快速返回到我们调试的程序领空

**ALT+M**: 查看当前程序的加载模块，可用于分析目标程序主文件结构

**CTRL+E**: 编辑数据

**CTRL+F2**: 重新开始调试

**CTRL+F9**: 执行到返回。此命令在执行到一个 **ret** (返回指令)指令时暂停，常用于从系统领空返回到我们调试的程序领空

**SHIFT+F7**: 忽略异常后单步执行

**SHIFT+F8**: 忽略异常后单步步过

# OllyDbg工具的使用 - 基本汇编指令

- MOV** 传送字或字节 如MOV A B,就是将B中的字传给A
- PUSH** 把字压入堆栈
- CALL** 子程序调用指令
- XOR** 异或运算 所谓异或,就是两值不同,则为真,反之,为假
- RET** 子程序返回指令
- CMP** 比较.(两操作数作减法,仅修改标志位,不回送结果)
- JNZ/jNE** OPR--结果不为零转移,测试条件ZF=0
- DEC** 减 1 **INC** 加 1
- JZ(或jE)** OPR--结果为零转移,测试条件ZF=1
- SUB** 减法
- LEA** 装入有效地址 例: LEA DX,string;把偏移地址存到DX.
- MOVSX** 先符号扩展,再传送
- REP** 当CX/ECX≠0时重复
- AND** 与运算
- TEST** 测试.(两操作数作与运算,仅修改标志位,不回送结果)

# OllyDbg工具的使用 - 关键位置

004011AE下一个断点,有调用到GetDlgItemTextA这个函数

004011AA	. 8D4424 4C	LEA EAX,DWORD PTR SS:[ESP+4C]	
004011AE	. 6A 51	PUSH 51	Count = 51 (81.)
004011B0	. 50	PUSH EAX	Buffer
004011B1	. 6A 6E	PUSH 6E	ControlID = 6E (1
004011B3	. 56	PUSH ESI	hWnd
004011B4	. FFD7	CALL EDI	GetDlgItemTextA
004011B6	. 8D8C24 9C000	LEA ECX,DWORD PTR SS:[ESP+9C]	
004011BD	. 6A 65	PUSH 65	Count = 65 (101.)
004011BF	. 51	PUSH ECX	Buffer

开始分析汇编代码的意义,所以,我们在使用这个软件的时候,一定要明白这些代码的含义,此处只是爆破,接下来就是修改代码: 004011F5

004011E4	. 50	PUSH EAX	
004011E5	. E8 56010000	CALL TraceMe.00401340	序列号计算的CALL
004011EA	. 8B3D BC40400	MOV EDI,DWORD PTR DS:[<USER32.GetDlgItem	USER32.GetDlgItem
004011F0	. 83C4 0C	ADD ESP,0C	
004011F3	. 85C0	TEST EAX,EAX	EXA=0,注册失败; EXA=1, 注册成功
004011F5	. 74 37	JE SHORT TraceMe.0040122E	不跳转则成功

# OllyDbg工具的使用 - 爆破

修改反汇编代码段，双击反汇编列后按空格键，键入NOP，汇编

004011DC	. 8D8424 A0000	LEA EAX,DWORD PTR SS:[ESP+A0]
004011E3	. 52	PUSH EDX
004011E4	. 50	PUSH EAX
004011E5	. E8 56010000	CALL TraceMe.00401340
004011EA	. 8B3D BC40400	MOV EDI,DWORD PTR DS:[<&USER32.GetDlgIt
004011F0	. 83C4 0C	ADD ESP,0C
004011F3	. 85C0	TEST EAX,EAX
004011F5	. 90	NOP
004011F6	. 90	NOP
004011F7	. 8D4C24 0C	LEA ECX,DWORD PTR SS:[ESP+C]

汇编于此处: 004011F5

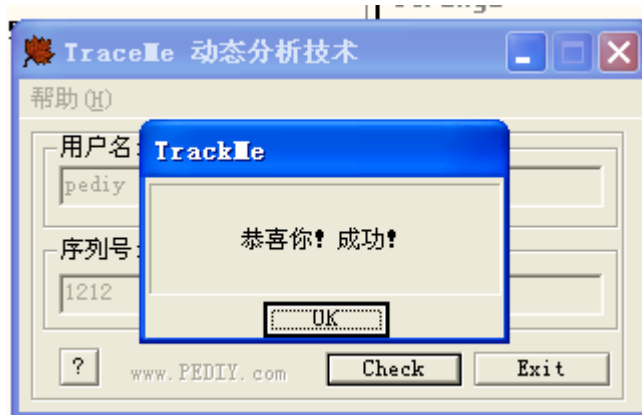
NOP

☒ 使用 NOP 填充

汇编 取消

不跳转则成功

最后 F9运行，你会看到：



# windbg工具的使用 - 符号表

- PDB文件

- 链接器自动生成
- 文件由两个部分构成，私有符号数据 (*private symbol data*) 和公共符号表 (*public symbol table*)

- 私有符号数据 (Private Symbol Data)

- 函数
- 全局变量
- 局部变量
- 用户定义的结构体，类，数据类型
- 源文件的名称和源文件中每个二进制指令的行号

- 公共符号表(Public Symbol Table)

- 静态函数
- 全局变量(extern)

# windbg工具的使用 - 内核调试环境 - windbg+vmware

直观效果



# windbg工具的使用 - 内核调试 - 修改Boot.ini

找到Boot.ini



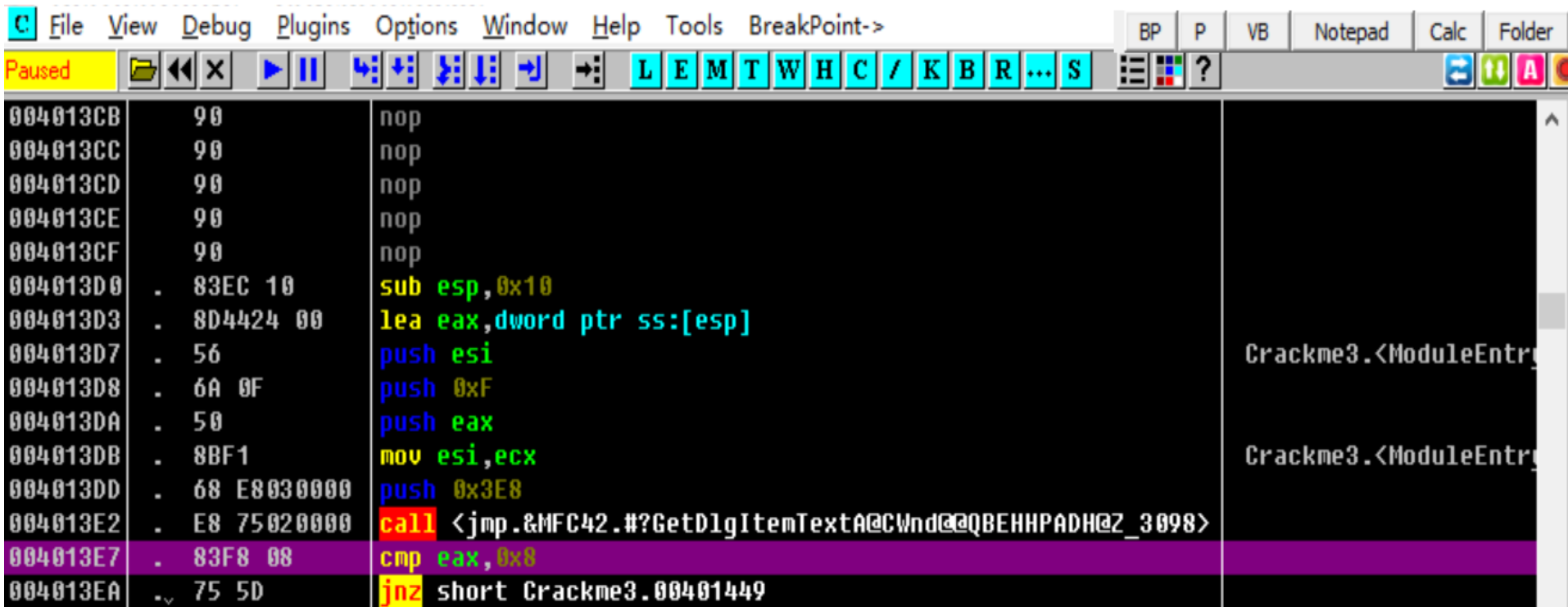
进行调试修改

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
/noexecute=optin /fastdetect /noexecute=alwaysoff
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional Debug"
/fastdetect /debug /debugport=com /baudrate=115200
```



# 面向汇编的逐句解析 - 关键点

KernelMode - Crackme3.exe - [\*G.P.U\* - main thread, module Crackme3]



```
004013CB  90      nop
004013CC  90      nop
004013CD  90      nop
004013CE  90      nop
004013CF  90      nop
004013D0  . 83EC 10  sub esp,0x10
004013D3  . 8D4424 00 lea eax,dword ptr ss:[esp]
004013D7  . 56      push esi
004013D8  . 6A 0F   push 0xF
004013DA  . 50      push eax
004013DB  . 8BF1   mov esi,ecx
004013DD  . 68 E8030000 push 0x3E8
004013E2  . E8 75020000 call <jmp.&MFC42.??GetDlgItemTextA@CWnd@@QBEPADH@Z_3098>
004013E7  . 83F8 08  cmp eax,0x8
004013EA  . 75 5D   jnz short Crackme3.00401449
```

# 面向汇编的逐句解析 - 长度判断

KernelMode - Crackme3.exe - [\*G.P.U\* - main thread, module Crackme3]

```
004013CB  90      nop
004013CC  90      nop
004013CD  90      nop
004013CE  90      nop
004013CF  90      nop
004013D0  83EC 10  sub esp,0x10
004013D3  8D4424 00 lea eax,dword ptr ss:[esp]
004013D7  56      push esi
004013D8  6A 0F   push 0xF
004013DA  50      push eax
004013DB  8BF1    mov esi,ecx
004013DD  68 E8030000 push 0x3E8
004013E2  E8 75020000 call <jmp.&MFC42.#!?GetDlgItemTextA@C
004013E7  83F8 08  cmp eax,0x8
004013EA  75 5D   jnz short Crackme3.00401449
004013EC  807C24 09 2D cmp byte ptr ss:[esp+0x9],0x2D
004013F1  75 56   jnz short Crackme3.00401449
004013F3  0FBEC24 04 movsx ecx,byte ptr ss:[esp+0x4]
004013F8  D1E1    shl ecx,1
004013FA  83F9 64  cmp ecx,0x64
004013FD  75 4A   jnz short Crackme3.00401449
004013FF  8A4424 0B mov al,byte ptr ss:[esp+0xB]
00401403  84C0    test al,al
```

注册码长度等于8, 不相等就跳出去, 注册失败

eax=00000006

Project PolyPhemous-NTS-Crackme3

Serial

Serial

123123

Check

Close

Cyclops / REAL

# 面向汇编的逐句解析 - 字符判断

\* KernelMode - Crackme3.exe - [\*G.P.U\* - main thread, module Crackme3]

Address	Disassembly	Comment
004013D8	8BF1	mov esi,ecx
004013DD	68 E8030000	push 0x3E8
004013E2	E8 75020000	call <jmp.&MFC42.##?GetDlgItemTextf>
004013E7	83F8 08	cmp eax,0x8
004013EA	74 5D	je short Crackme3.00401449
004013EC	807C24 09 2D	cmp byte ptr ss:[esp+0x9],0x2D
004013F1	74 56	je short Crackme3.00401449
004013F3	0FBEC424 04	movsx ecx,byte ptr ss:[esp+0x4]
004013F8	D1E1	shl ecx,1
004013FA	83F9 64	cmp ecx,0x64
004013FD	74 4A	je short Crackme3.00401449
004013FF	8A4424 0B	mov al,byte ptr ss:[esp+0x8]
00401403	84C0	test al,al
00401405	75 42	jnz short Crackme3.00401449
00401407	807C24 08 2B	cmp byte ptr ss:[esp+0x8],0x2B
0040140C	74 3B	je short Crackme3.00401449
0040140E	0FBEE424 05	movsx edx,byte ptr ss:[esp+0x5]
00401413	83C2 0A	add edx,0xA
00401416	83FA 44	cmp edx,0x44
00401419	75 2E	jnz short Crackme3.00401449
0040141B	0FBEE424 07	movsx eax,byte ptr ss:[esp+0x7]
00401420	83E8 2E	sub eax,0x2E
00401423	75 24	jnz short Crackme3.00401449

edx=00000032

注册码长度等于8，不相等就跳出去，注册失败

字符“-”所以注册码第6位为-

注册码第1位扩展到ECX  
左移一位  
字符d，把d右移一位则为32H，说明注册码第1位是数字2

注册码第8位放入AL  
第8位是否为0  
第8位只要不为0即可  
第5位为2B即十进制43，就是+号

注册码第2位零扩展到EDX  
与000AH相加  
加完之后等于44H，即第2位为44-A=3A，也就是:号

# 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

# 花指令

- 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，缺少了也能正常运行
- 加花指令后，IDA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
- 花指令有可能利用各种指令：jmp, call, ret的一些堆栈技巧，位置运算等

# 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
- 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
- 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

## 寻找程序入口点 (OEP)

- 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 OEP 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

# 反调试技术

## 反调试分类：

1. 调试器检测：各种方法查看调试器是否存在
2. 识别调试器：识别是否在调试中
3. 干扰调试器：令调试失败

# 反调试技术

1. 调试器检测：Windows API，手动检测数据结构，系统痕迹检测
2. 识别调试器：检测软件/硬件断点，时钟检测，父进程判断
3. 干扰调试器：TLS回调，利用中断，陷阱标志位

# 调试器检测 -- Windows API

- **1.2 CheckRemoteDebuggerPresent**

CheckRemoteDebuggerPresent同IsDebuggerPresent几乎一致。它不仅可以探测系统其他进程是否被调试，通过传递自身进程句柄还可以探测自身是否被调试。

```
1  BOOL CheckDebug()  
2  {  
3      BOOL ret;  
4      CheckRemoteDebuggerPresent(GetCurrentProcess(), &ret);  
5      return ret;  
6  }
```

# 调试器检测 -- Windows API

## • 1.3 NtQueryInformationProcess

第二个参数是一个枚举类型，其中与反调试有关的成员有：

ProcessDebugPort(0x7)、ProcessDebugObjectHandle(0x1E)

和ProcessDebugFlags(0x1F)。若将该参数置为ProcessDebugPort，如果进程正在被调试，则返回调试端口，否则返回0

注意，这里的NtQueryInformationProcess是通过函数名引出，不能直接调用。

```
BOOL CheckDebug()
{
    int debugPort = 0;
    HMODULE hModule = LoadLibrary("Ntdll.dll");
    NtQueryInformationProcessPtr NtQueryInformationProcess = (NtQueryInformationProcessPtr)GetProcAddress(hModule, "NtQueryInformationProcess");
    NtQueryInformationProcess(GetCurrentProcess(), 0x7, &debugPort, sizeof(debugPort), NULL);
    return debugPort != 0;
}
```

# 调试器检测 – 手动检测数据结构

## • 2.1 BeingDebugged属性

Windows操作系统维护着每个正在运行的进程的PEB结构，它包含与这个进程相关的所有用户态参数。这些参数包括进程环境数据，环境数据包括环境变量、加载的模块列表、内存地址，以及调试器状态

FS指向当前的TEB结构，偏移0x30处是ProcessEnvironmentBlock域，指向PEB结构体

```
typedef struct _PEB {  
    BYTE Reserved1[2];  
    BYTE BeingDebugged;  
    BYTE Reserved2[1];  
    PVOID Reserved3[2];  
    PPEB_LDR_DATA Ldr;  
};
```

```
1  BOOL CheckDebug()  
2  {  
3      int result = 0;  
4      __asm  
5      {  
6          mov eax, fs:[30h]  
7          mov al, BYTE PTR [eax + 2]  
8          mov result, al  
9      }  
10     return result != 0;  
11 }
```

# 调试器检测 – 手动检测数据结构

## • 2.2 ProcessHeap属性

ProcessHeap位于PEB结构的0x18处。

第一个堆头部有一个属性字段，它告诉内核这个堆是否在调试器中

创建。这些属性叫作ForceFlags。在Windows 7或更高版本的系统中，对于32位的应用程序来说ForceFlags属性位于堆头部偏移量0x44处。低版本的系统中，偏移量为0x10。

```
BOOL CheckDebug()
{
    int result = 0;
    DWORD dwVersion = GetVersion();
    DWORD dwWindowsMajorVersion = (DWORD)(LOBYTE(LOWORD(dwVersion)));
    //for xp
    if (dwWindowsMajorVersion == 5)
    {
        __asm
        {
            mov eax, fs:[30h]
            mov eax, [eax + 18h]
            mov eax, [eax + 10h]
            mov result, eax
        }
    }
    else
    {
        __asm
        {
            mov eax, fs:[30h]
            mov eax, [eax + 18h]
            mov eax, [eax + 44h]
            mov result, eax
        }
    }
    return result != 0;
}
```



# 调试器检测 – 手动检测数据结构

## • 2.3 NTGlobalFlag

由于调试器中启动进程与正常模式下启动进程有些不同，所以它们创建内存堆的方式也不同。系统使用PEB结构偏移量0x68处的一个未公开位置，来决定如何创建堆结构。如果这个位置的值为0x70，我们就知道进程正运行在调试器中。

```
BOOL CheckDebug()
{
    int result = 0;
    __asm
    {
        mov eax, fs:[30h]
        mov eax, [eax + 68h]
        and eax, 0x70
        mov result, eax
    }
    return result != 0;
}
```

# 调试器检测 – 系统痕迹检测

## • 3.1 查找调试器引用的注册表项

SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug(32位系统)

SOFTWARE\Wow6432Node\Microsoft\WindowsNT\CurrentVersion\AeDebug(64位系统)

该注册表项指定当应用程序发生错误时，触发哪一个调试器。默认情况下，它被设置为**Dr.Watson**。如果该注册表的键值被修改为**OllyDbg**（或其他非默认值），则恶意代码可确定它正在被调试。

```
}  
char tmp[256];  
DWORD len = 256;  
DWORD type;  
ret = RegQueryValueExA(hkey, key, NULL, &type, (LPBYTE)tmp, &len);  
if (strstr(tmp, "OllyIce")!=NULL || strstr(tmp, "OllyDBG")!=NULL || strstr(tmp, "WinDbg")!=NULL || strstr(tmp, "x64dbg")!=NULL || strstr(tmp, "Immunity")!=NULL)  
{  
    return TRUE;  
}  
else  
{  
    return FALSE;  
}
```

# 调试器检测 – 系统痕迹检测

## • 3.2 查找窗体信息

FindWindow函数检索处理顶级窗口的类名和窗口名称匹配指定的字符串。

```
BOOL CheckDebug()
{
    if (FindWindowA("OLLYDBG", NULL)!=NULL || FindWindowA("WinDbgFrameClass", NULL)!=NULL || FindWindowA("Qwidget", NULL)!=NULL)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

```
BOOL CheckDebug()
{
    char fore_window[1024];
    GetWindowTextA(GetForegroundWindow(), fore_window, 1023);
    if (strstr(fore_window, "WinDbg")!=NULL || strstr(fore_window, "x64_dbg")!=NULL || strstr(fore_window, "OlllyICE")!=NULL || strstr(fore_window, "OlllyDBG")!=NULL || strstr(fore_window, "Immunity")!=NULL)
    {
        return TRUE;
    }
}
```

# 识别调试器－检测断点

## • 4.1 检测软件断点

- 调试器设置断点的基本机制是用软件中断指令INT 3临时替换运行程序中的一条指令，然后当程序运行到这条指令时，调用调试异常处理例程。
- INT 3指令的机器码是0xCC，因此无论何时，使用调试器设置一个断点，它都会插入一个0xCC来修改代码。
- 常用的一种反调试技术是在它的代码中查找机器码0xCC，来扫描调试器对它代码的INT 3修改。

# 识别调试器 — 检测断点

```
BOOL CheckDebug()
{
    PIMAGE_DOS_HEADER pDosHeader;
    PIMAGE_NT_HEADERS32 pNtHeaders;
    PIMAGE_SECTION_HEADER pSectionHeader;
    DWORD dwBaseImage = (DWORD)GetModuleHandle(NULL);
    pDosHeader = (PIMAGE_DOS_HEADER)dwBaseImage;
    pNtHeaders = (PIMAGE_NT_HEADERS32)((DWORD)pDosHeader + pDosHeader->e_lfanew);
    pSectionHeader = (PIMAGE_SECTION_HEADER)((DWORD)pNtHeaders + sizeof(pNtHeaders->Signature) + sizeof(IMAGE_FILE_HEADER) +
        (WORD)pNtHeaders->FileHeader.SizeOfOptionalHeader);
    DWORD dwAddr = pSectionHeader->VirtualAddress + dwBaseImage;
    DWORD dwCodeSize = pSectionHeader->SizeOfRawData;
    BOOL Found = FALSE;
    __asm
    {
        cld
        mov     edi,dwAddr
        mov     ecx,dwCodeSize
        mov     al,0CCH
        repne   scasb
        jnz     NotFound
        mov     Found,1
    }
    NotFound:
    }
    return Found;
}
```

# 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;
    }
}
```



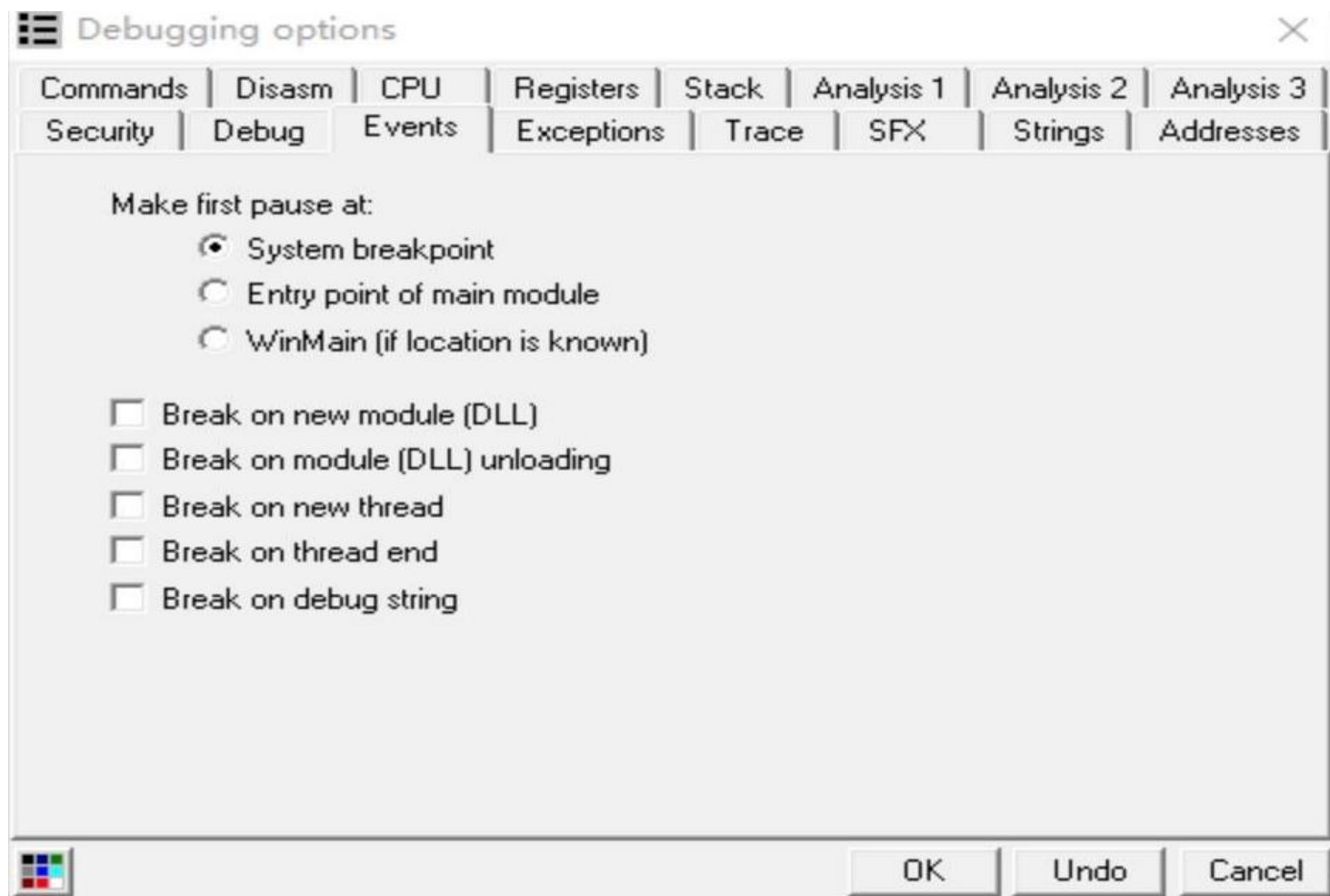
# 识别调试器 – 父进程判断

## • 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



# 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。

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

# 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);
}
```

# 祝大家考试顺利！



南京邮电大学  
Nanjing University of Posts and Telecommunications