



# 恶意软件分析

## 第5章：动态分析高级技术

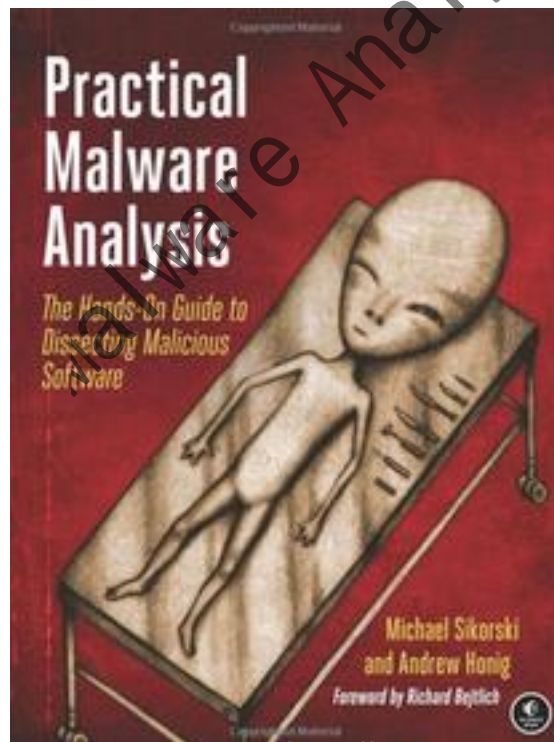


# 主要内容

- 5.1 动态调试
- 5.2 OllyDbg
- 5.3 使用WinDbg调试内核



## 5.1: 动态调试



Malware Analysis(<http://scs.ucas.ac.cn>)



# 反汇编器与调试器

- 反汇编器如IDA Pro，显示程序在开始执行前的状态
- 调试器显示
  - 每一个内存位置
  - 寄存器
  - 每个函数的参数
- 在程序执行过程中的任何时候
  - 都能改变它们



# 两个调试器

- Ollydbg
  - 最流行的恶意代码分析工具
  - 只用于用户模式的调试
  - IDA Pro 有个内置的调试器，但它不像Ollydbg那样强大易用
- Windbg
  - 支持内核模式的调试

# 源代码级与汇编级的调试器



- 源代码级调试器
  - 通常内置于开发平台
  - 能够设置断点
  - 能够单步调试程序，一次执行一行
- 汇编级调试器
  - 操作汇编代码而不是源代码
  - 恶意软件分析师通常不得不使用它们，因为他们没有源代码



# 内核模式与用户模式调试



# 用户模式调试

- 调试器与被调试的代码运行在同一个系统中
- 调试单个可执行文件
- 操作系统会将它与其他可执行程序隔离



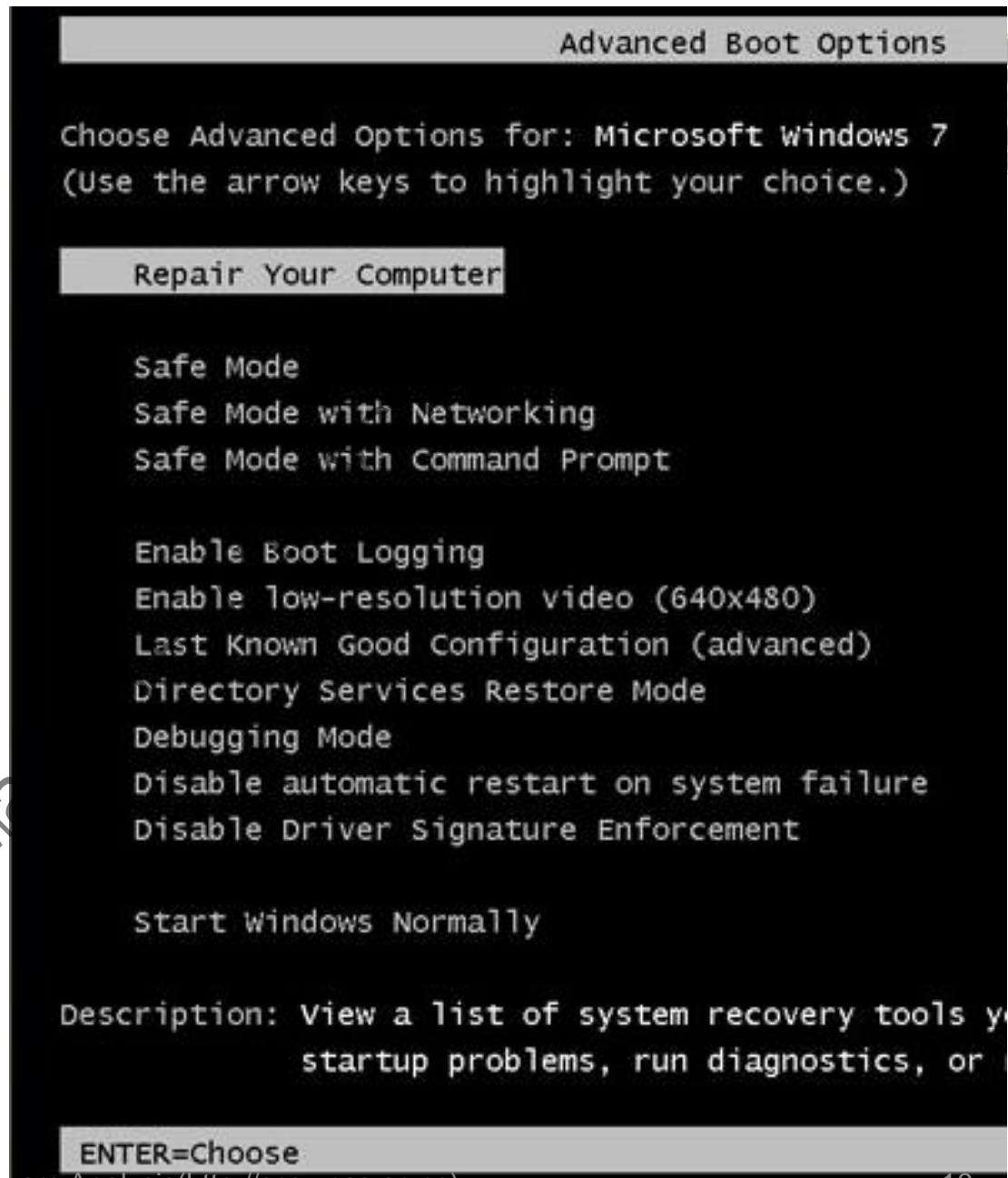


# 内核模式调试

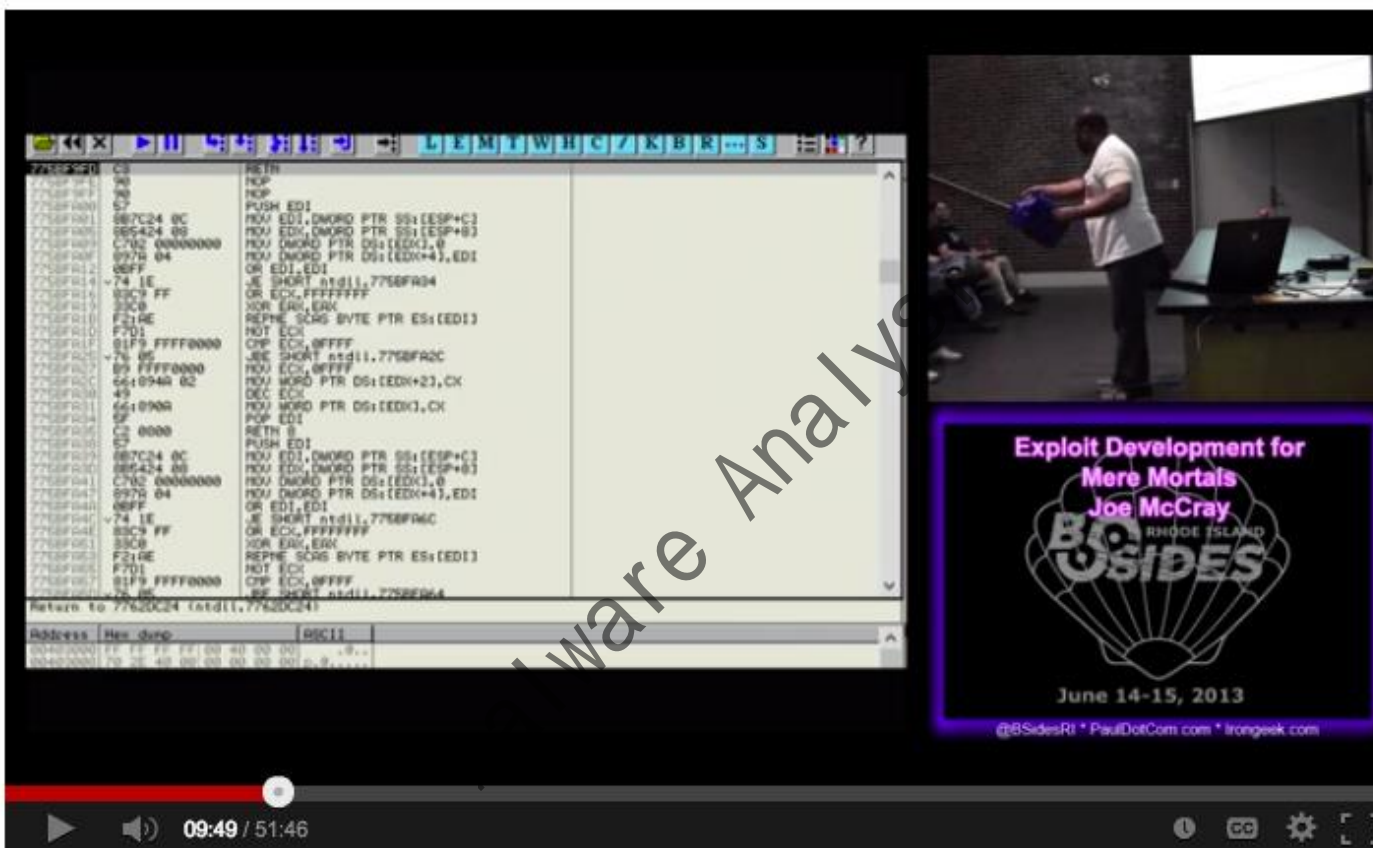
- 需要两个系统，因为每个系统只有一个内核
- 如果内核处于断点处，系统就停止了
- 一个系统运行被调试代码
- 另一个系统运行调试器
- 操作系统配置为开启内核调试功能
- 两个系统必须连通

# Windows 7 高级启动 选项

- 启动期间按F8
- “Debugging Mode”



# 011yDbg介绍



BsidesRI 2013 1 4 Exploit Development for Mere Mortals Joe Mc...



# 使用调试器

Malware Analysis



# 两种调试程序的方法

- 利用调试器启动程序
  - 程序在其入口点运行之前 立即停止运行
- 附加调试器到一个已经运行的程序上
  - 程序的所有线程被暂停
  - 是调试被恶意代码感染的进程的好方法



# 单步调试

- 简单但慢
- 不要陷入细节

Malware Analysis



# 事例

- 这段代码使用XOR  
解码字符串

## *Example 9-1. Stepping through code*

```
mov     edi, DWORD_00406904
mov     ecx, 0x0d
LOC_040106B2
xor     [edi], 0x9C
inc     edi
loopw   LOC_040106B2
...
DWORD:00406904:  F8FDF3D01
```

## *Example 9-2. Single-stepping through a section of code to see how it changes memory*

```
D0F3FDF8 D0F5FEEE FDEEE5DD 9C (.....)
4CF3FDF8 D0F5FEEE FDEEE5DD 9C (L.....)
4C6FFDF8 D0F5FEEE FDEEE5DD 9C (Lo.....)
4C6F61F8 D0F5FEEE FDEEE5DD 9C (Loa.....)
. . . SNIP . . .
4C6F6164 4C696272 61727941 00 (LoadLibraryA.)
```

# 单步跳过 (Stepping-over) 与 单步跳入 (Stepping-Into)



- 单步执行一条指令
- Step-over 指令
  - 完成调用并返回, 中间没有暂停
  - 减少你需要分析的代码数量
  - 可能会错过重要的功能, 尤其是如果函数没有返回时
- Step-into 函数调用
  - 进入函数并停在它的第一条 命令处





# 用断点暂停执行

- 一个程序在断点处暂停运行称作“中断”
- 事例
  - 不会告诉你调用了哪个函数
  - 调用处设置断点，看看eax中是有什么

## *Example 9-3. Call to EAX*

```
00401008  mov     ecx, [ebp+arg_0]
0040100B  mov     eax, [edx]
0040100D  call    eax
```

- 这段代码构建一个文件名，然后创建该文件
- 在 `CreateFileW` 处设置断点并查看堆栈中的文件名

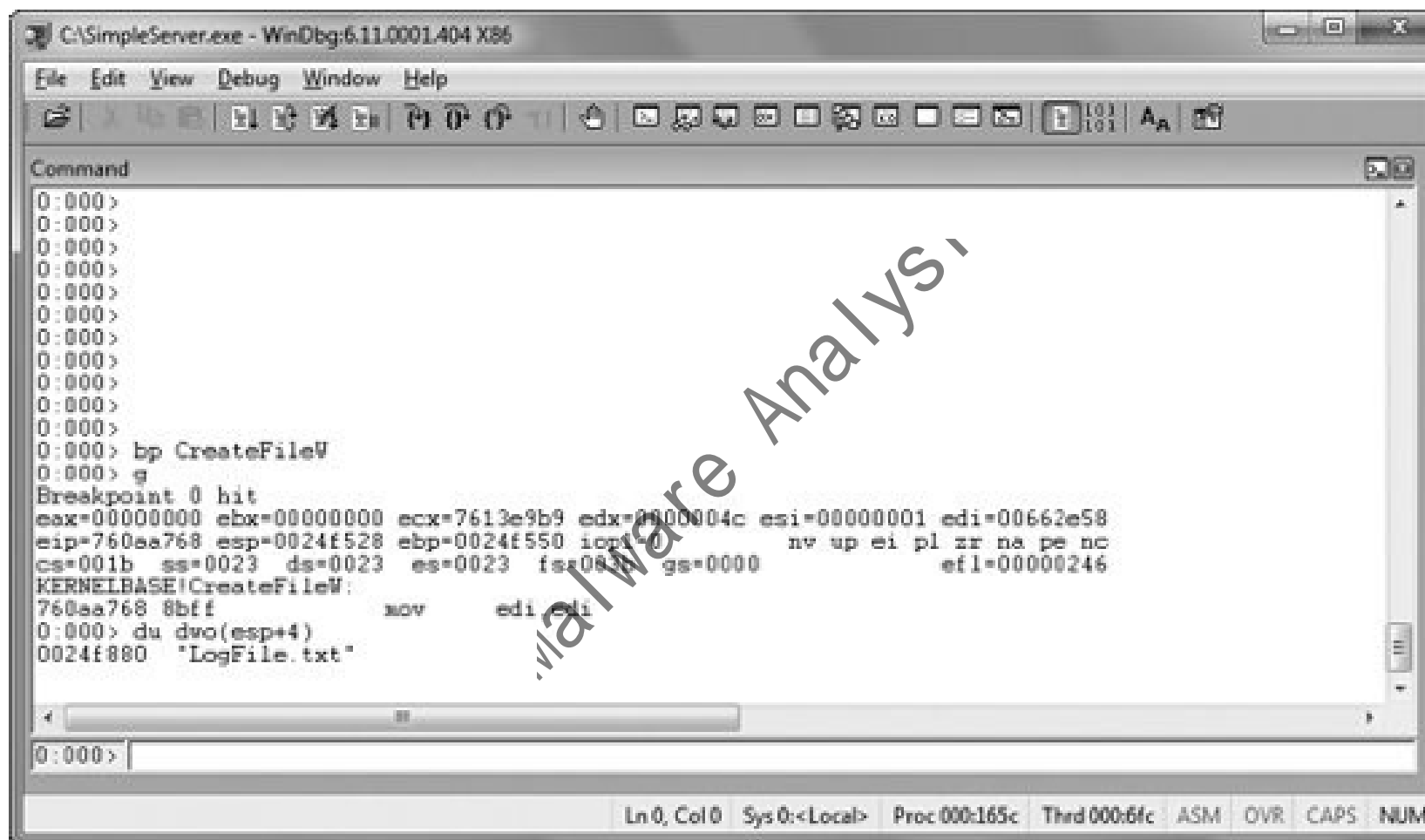
*Example 9-4. Using a debugger to determine a filename*

```

0040100B  xor     eax, esp
0040100D  mov     [esp+0D0h+var_4], eax
00401014  mov     eax, edx
00401016  mov     [esp+0D0h+NumberOfBytesWritten], 0
0040101D  add     eax, 0FFFFFFFh
00401020  mov     cx, [eax+2]
00401024  add     eax, 2
00401027  test    cx, cx
0040102A  jnz     short loc_401020
0040102C  mov     ecx, dword ptr ds:a_txt ; ".txt"
00401032  push    0 ; hTemplateFile
00401034  push    0 ; dwFlagsAndAttributes
00401036  push    2 ; dwCreationDisposition
00401038  mov     [eax], ecx
0040103A  mov     ecx, dword ptr ds:a_txt+4
00401040  push    0 ; lpSecurityAttributes
00401042  push    0 ; dwShareMode
00401044  mov     [eax+4], ecx
00401047  mov     cx, word ptr ds:a_txt+8
0040104E  push    0 ; dwDesiredAccess
00401050  push    edx ; lpFileName
00401051  mov     [eax+8], cx
00401055  call    CreateFileW ; CreateFileW(x,x,x,x,x,x,x,x)

```

# WinDbg



*Figure 9-1. Using a breakpoint to see the parameters to a function call. We set a breakpoint on `CreateFileV` and then examine the first parameter of the stack.*



# 加密的数据

- 假设恶意代码发送加密的网络数据
- 数据加密之前设置一个断点的并查看它

Malware Analysis



*Example 9-5. Using a breakpoint to view data before the program encrypts it*

```
004010D0  sub     esp, 0CCh
004010D6  mov     eax, dword_403000
004010DB  xor     eax, esp
004010DD  mov     [esp+0CCh+var_4], eax
004010E4  lea     eax, [esp+0CCh+buf]
004010E7  call    GetData
004010EC  lea     eax, [esp+0CCh+buf]
004010EF  1call   EncryptData
004010F4  mov     ecx, s
004010FA  push    0           ; flags
004010FC  push    0C8h        ; len
00401101  lea     eax, [esp+0D4h+buf]
00401105  push    eax         ; buf
00401106  push    ecx         ; s
00401107  call    ds:Send
```



# 011yDbg

The screenshot shows the OllyDbg interface with the assembly window displaying the following code:

```
010410D0 $ 81EC CC000000 SUB ESP,0CC
010410D6 . A1 00300401 MOV EAX,DWORD PTR DS:[_security_cookie]
010410DB . 33C4 XOR EAX,ESP
010410DD . 898424 C80000 MOV DWORD PTR SS:[ESP+C8],EAX
010410E4 . 8D0424 LEA EAX,DWORD PTR SS:[ESP]
010410E7 . E8 A4FFFFFF CALL SimpleSe.GetData
010410EC . 8D0424 LEA EAX,DWORD PTR SS:[ESP]
010410EF . E8 BCFFFFFF CALL SimpleSe.EncryptData
010410F4 . 8B0D 70330401 MOV ECX,DWORD PTR DS:[s]
010410FA . 6A 00 PUSH 0
010410FC . 68 C8000000 PUSH 0C8
01041101 . 8D4424 08 LEA EAX,DWORD PTR SS:[ESP+8]
01041105 . 50 PUSH EAX
01041106 . 51 PUSH ECX
01041107 . 55 15 F4000401 CALL SimpleSe.GetData
0104110B . 55 15 F4000401 CALL SimpleSe.GetData
0104110E =SimpleSe.EncryptData
```

Below the assembly window, a hex dump is shown:

Address	Hex dump	ASCII
0012F9C8	53 65 63 72 65 74 20 40 65 73 73 61 67 65 2E 00	Secret Message..
0012F9D8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0012F9E8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0012F9F8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

The status bar at the bottom indicates a breakpoint at SimpleSe.010410EF and the program is paused.

Figure 9-2. Viewing program data prior to the encryption function call



# 断点类型

- 软件执行断点
- 硬件执行断点
- 条件断点

Malware Analysis



# 软件执行断点

- 大多数调试器的默认设置
- 调试器通过调用0xCC重写指令的首个字节来实现软件断点
  - 指令INT 3
  - 一个设计用于调试器的中断
  - 执行断点时，操作系统生成一个异常并将控制转到调试器





# 断点处的内存内容

- 这是push指令处的断点
- 调试器显示 0x55，但实际上是 0xCC

*Table 9-1. Disassembly and Memory Dump of a Function with a Breakpoint Set*

Disassembly view				Memory dump			
00401130	55	1	push	ebp	00401130	2	CC 8B EC 83
00401131	8B EC		mov	ebp, esp	00401134		E4 F8 81 EC
00401133	83 E4 F8		and	esp, 0FFFFFFF8h	00401138		A4 03 00 00
00401136	81 EC A4 03 00 00		sub	esp, 3A4h	0040113C		A1 00 30 40
0040113C	A1 00 30 40 00		mov	eax, dword_403000	00401140		00

# 软件断点执行失败的情况



- 在代码执行过程中，如果0xCC字节被改变，断点将不会发生
- 如果其他代码读到的内存包含断点，它读到代替原始字节的0xCC
- 验证完整性的代码会注意到这种差异



# 硬件执行断点

- 使用4个硬件调试寄存器
  - DR0 ~ DR3 存储 断点的地址
  - DR7存储控制信息
- 寄存器中存储断点地址
- 能够在访问或执行时触发中断
  - 能够设置读、写或访问时触发中断
- 没有改变代码字节



# 硬件执行断点

- 运行代码可以更改DR寄存器，干扰调试器
- DR7通用探测标志位（General Detect flag）
  - 任何将会改变调试寄存器内容的mov指令的执行都会触发中断
  - 但并不检测其他指令



# 条件断点

- 仅条件满足时才触发中断
  - 如：在函数 `GetProcAddress` 上设置断点
  - 只想在传入的参数为 `RegSetValue` 时触发中断
- 实现与软件断点相同
  - 调试器总是接收中断
  - 如果条件不满足，它会自动继续执行而不通知用户



# 条件断点

- 条件断点比普通指令花费更长的时间
- 在频繁访问的指令设置条件断点会减慢程序执行
- 有时，程序运行会变得非常慢以至于它不能结束运行



异常

Malware Analysis



# 异常

- 被调试器用来获取运行程序控制权
- 断点产生异常
- 以下操作也会产生异常
  - 无效的内存访问
  - 被0除
  - 其他情况





# 首次和二次异常处理

- 附加调试器时会产生异常
  - 程序停止执行
  - 调试器第一次获取控制权
  - 调试器可以自己处理异常也可以将异常转给被调试的应用程序处理
  - 如果被转给程序，程序异常处理程序会接收它



# 二次异常

- 如果应用程序没有处理异常
- 调试器获得第二次处理它的机会
  - 这意味着如果程序没有附加调试器就会崩溃
- 在恶意代码分析中，首次异常经常被忽略
- 二次异常不能被忽略
  - 它们通常表示恶意代码并不想在当前环境中运行



# 常见异常

- INT 3 (软件断点)
- 调试器中单步调试作为一个异常来实现
  - 如果寄存器中的陷阱标志 ( trap flag) 被设置
  - 处理器执行一条指令并产生一个异常
- 内存访问冲突 (Memory-access violation) 异常
  - 代码试图访问一个无权访问的内存位置, 因为内存地址无效或者因为访问了受保护而无权访问的内存位置



# 常见异常

- 违反特权规则
  - 在非特权模式试图执行特权指令
  - 换句话说，试图在用户模式执行内核模式指令
  - 或者，试图从Ring 3执行Ring 0 指令



# 异常列表

The following chart lists the exceptions that can be generated by the Intel 80286, 80386, 80486, and Pentium processors:

Exception (dec/hex)	Description
0 00h	Divide error: Occurs during a DIV or an IDIV instruction when the divisor is zero or a quotient overflow occurs.
1 01h	Single-step/debug exception: Occurs for any of a number of conditions: <ul style="list-style-type: none"><li>- Instruction address breakpoint fault</li><li>- Data address breakpoint trap</li><li>- General detect fault</li><li>- Single-step trap</li><li>- Task-switch breakpoint trap</li></ul>
2 02h	Nonmaskable interrupt: Occurs because of a nonmaskable hardware interrupt.
3 03h	Breakpoint: Occurs when the processor encounters an INT 3 instruction.



# 使用调试器修改可执行文件



# 跳过函数调用

- 改变控制标志、指令指针或者代码本身
- 避免函数被调用，在其被调用处设置断点并改变指令指针到函数之后的指令
  - 可能会导致程序不能正常运行或崩溃



# 测试一个函数

- 可以直接运行函数，不用等主代码调用它
  - 不得不设置函数的参数
  - 这将破坏程序堆栈
  - 在函数执行完毕后，程序将不会正确运行





# 修改可执行程序与实践



# 真实病毒

- 操作取决于电脑语言设置
  - 简体中文
    - 卸载自己并不对计算机产生任何危害
  - 英语
    - 弹出窗口显示 "Your luck's no good"
  - 日语或者印度尼西亚语
    - 使用随机数据重写硬盘

# 在1处设置断点来改变返回值

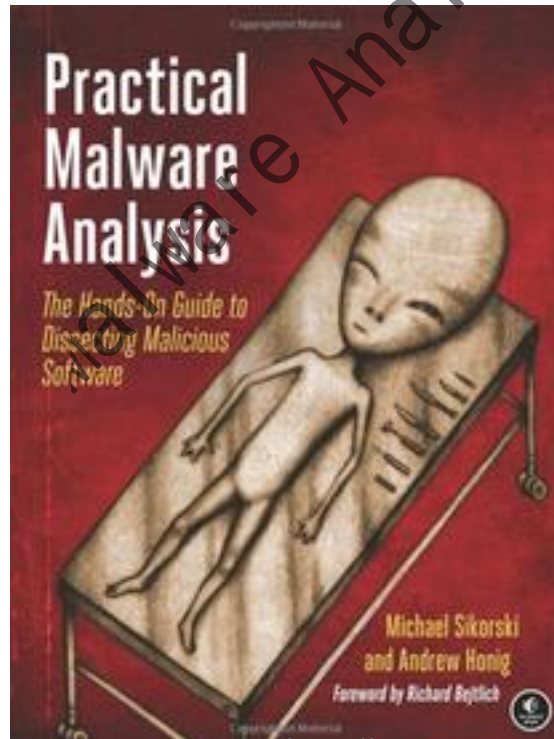


*Example 9-6. Assembly for differentiating between language settings*

```
00411349  call    GetSystemDefaultLCID
0041134F  1mov     [ebp+var_4], eax
00411352  cmp     [ebp+var_4], 409h           409 = English
00411359  jnz     short loc_411360
0041135B  call    sub_411037
00411360  cmp     [ebp+var_4], 411h           411 = Japanese
00411367  jz      short loc_411372
00411369  cmp     [ebp+var_4], 421h           421 = Indonesian
00411370  jnz     short loc_411377
00411372  call    sub_41100F
00411377  cmp     [ebp+var_4], 0C04h          C04 = Chinese
0041137E  jnz     short loc_411385
00411380  call    sub_41100A
```



## 5.2: 011yDbg



# 历史



- OllyDbg是十多年前开发的
- 首先用在破解软件和漏洞利用开发
- OllyDbg 1.1 源代码被Immunity并更名为Immunity Debugger
- 这两个产品很相似



# 加载恶意代码



# 调试恶意代码的方法

- 将EXE或者DLL直接加载到011yDbg
- 如果恶意代码已经执行，可以使用附加功能附加到正在运行的进程



# 打开一个可执行文件

- File->Open
- 如果需要的话添加命令行参数
- 如果能够确定软件入口点位置（如：WinMain）OllyDbg 将停在该处
- 否则它将停在PE头部提供的入口点处
  - 选择Debugging Options配置启动项



# 附加调试器到一个运行程序



- File-> Attach
- OllyDbg 暂停这个程序和它的所有线程
  - 如果程序正在执行DLL时被OllyDbg暂停了，在整个代码段中设置一个访问断点，从而到达感兴趣的代码



# 重新加载一个文件

- Ctrl+F2重新加载当前可执行文件
- F2 设置一个断点

Malware Analysis



# OllyDbg 界面



OlllyDbg - Lab09-01.exe - [CPU - main thread, module Lab09-01]

File View Debug Trace Options Windows Help

Registers (FPU)

EAX 76F51142 kernel32.76F51142  
ECX 00000000  
EDX 00403896 Lab09-01.<ModuleEntryPoint>  
EBX 77FD5000  
ESP 0012FF8C  
EBP 0012FF94  
ESI 00000000  
EDI 00000000  
EIP 00403896 Lab09-01.<ModuleEntryPoint>

Registers

Disassembler

Highlight: next instruction to be executed

Stack

Memory dump

Stack

Entry point of main module

Paused



# 修改数据

- 反汇编窗口
  - 按空格键
- 寄存器或堆栈窗口
  - 右击->修改 ( modify)
- 内存转储窗口
  - 右击->二进制 ( Binary) ->编辑 ( Edit)
  - Ctrl+G 跳转到指定内存位置
  - 右击一个另一边的内存地址点击 “Follow in dump”



# 内存映射

View Memory Map

Memory map									
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as	
00010000	00010000			Stack of main thr	Map	Rw	Rw	\Device\HarddiskVolume1\Windows\System32\locale.nls	
00020000	00010000				Map	Rw	Rw		
00120000	00001000				Priv	Rw	Guar		
0012E000	00002000				Priv	Rw	Rw		
00130000	00004000				Map	R	R		
00140000	00001000				Priv	Rw	Rw		
00150000	00067000				Map	R	R		
001C0000	00001000				Priv	Rw	Rw		
001D0000	00001000				Priv	Rw	Rw		
00240000	00003000				Priv	Rw	Rw		
002A0000	00008000				Priv	Rw	Rw		
00400000	00001000	Lab09-01		PE header	Img	R	RWE	Cop	
00401000	0000A000	Lab09-01	.text	Code	Img	R E	RWE	Cop	
0040B000	00001000	Lab09-01	.rdata	Imports	Img	R	RWE	Cop	
0040C000	00005000	Lab09-01	.data	Data	Img	Rw	Cop	RWE	Cop
00420000	00005000				Map	R	R		
004E0000	00003000			GDI handles	Map	R	R		
004F0000	00101000				Map	R	R		
00600000	00008000				Map	R	R		
75C60000	00001000	KERNELBA		PE header	Img	R	RWE	Cop	
75C61000	00044000	KERNELBA			Img	R E	RWE	Cop	
75CA5000	00002000	KERNELBA			Img	Rw	RWE	Cop	
75CA7000	00004000	KERNELBA			Img	R	RWE	Cop	
75EB0000	00001000	NSI		PE header	Img	R	RWE	Cop	
75EB1000	00002000	NSI			Img	R E	RWE	Cop	
75EB3000	00001000	NSI			Img	Rw	RWE	Cop	
75EB4000	00002000	NSI			Img	R	RWE	Cop	
75EC0000	00001000	SHELL32		PE header	Img	R	RWE	Cop	
75EC1000	0003C000	SHELL32			Img	R E	RWE	Cop	
76289000	00007000	SHELL32			Img	Rw	Cop	RWE	Cop
76290000	00079000	SHELL32			Img	R	RWE	Cop	
76B10000	00001000	USER32		PE header	Img	R	RWE	Cop	
76B11000	00068000	USER32			Img	R E	RWE	Cop	
76B79000	00001000	USER32			Img	Rw	RWE	Cop	
76B7A000	0005F000	USER32			Img	R	RWE	Cop	
76BE0000	00001000	sechost		PE header	Img	R	RWE	Cop	
76BE1000	00013000	sechost			Img	R E	RWE	Cop	
76BF4000	00003000	sechost			Img	Rw	Cop	RWE	Cop
76BF7000	00002000	sechost			Img	R	RWE	Cop	

- EXE 和 DLLs 被标注
- 双击任意行显示内存转储
- 右击->View in Disassembler



# 基地址重定位

- 基地址重定向发生在一个模块没有加载到其预定地址的时候
- PE 文件有个预定的基地址
  - 映像基地址在PE 文件头里
  - 通常在这个地址加载该文件
  - 大部分可执行文件都设计为在 0x00400000处加载
- 支持地址空间布局随机化 (ASLR) 的 EXE文件经常会被重定位





# DLL 基地址重定位

- DLL的重定位更普遍
  - 因为一个应用程序可能导入多个DLL
  - Windows 自带的DLL各自有不同的基地址，以避免重定位
  - 第三方的DLL经常使用同一个预定基地址



# 绝对地址与相对地址

*Example 10-1. Assembly code that requires relocation*

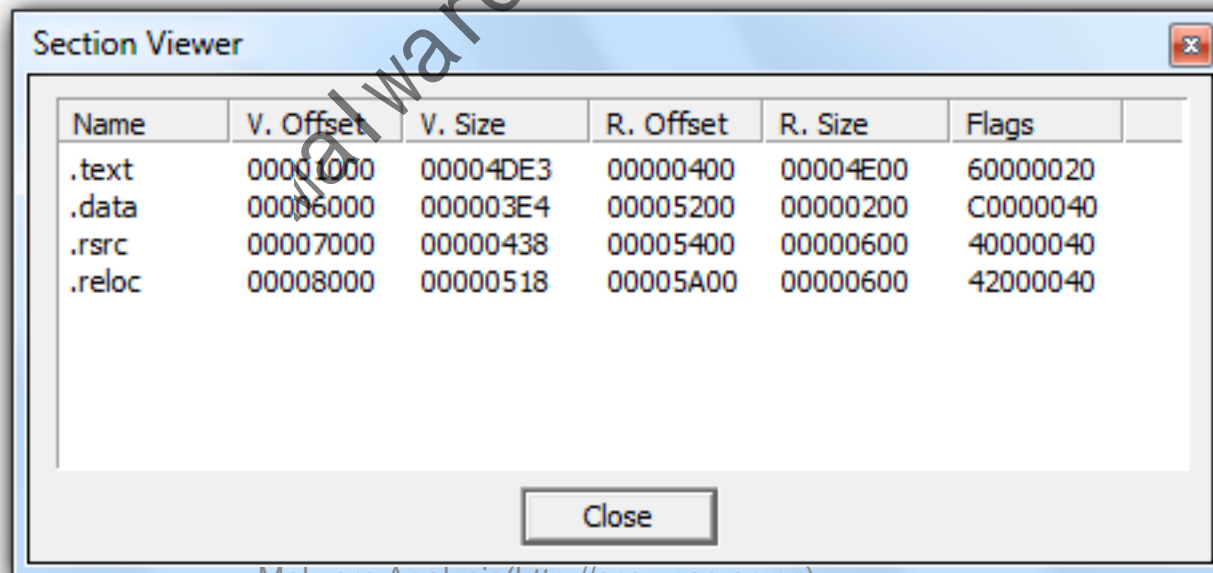
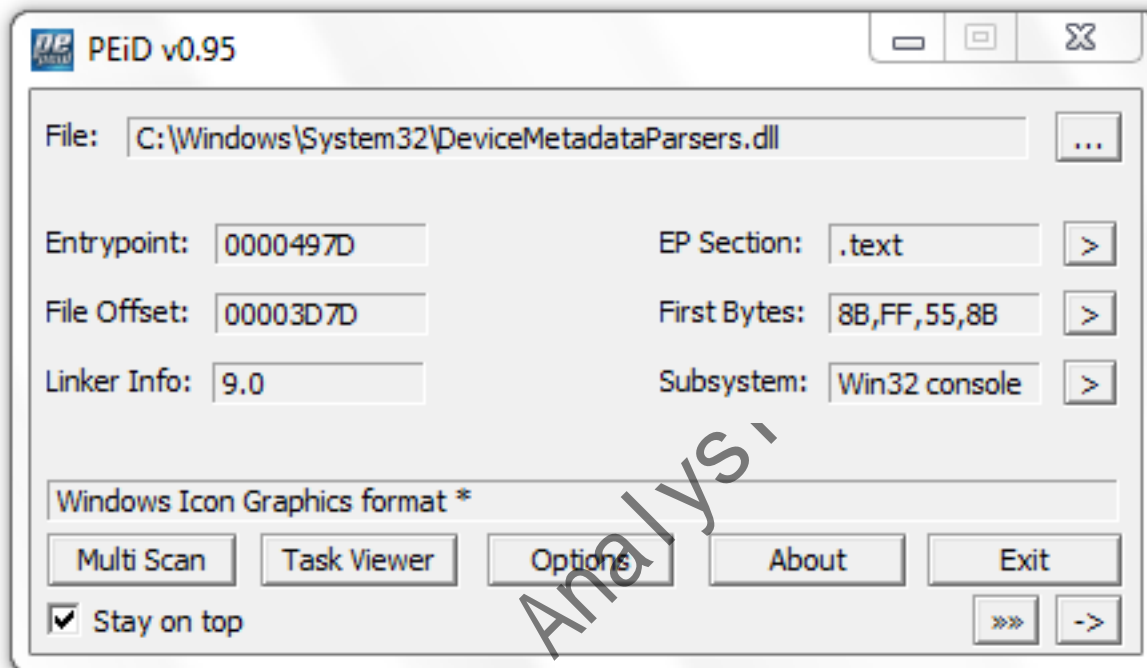
```
00401203      mov eax, [ebp+var_8]
00401206      cmp [ebp+var_4], 0
0040120a      jnz loc_0040120
0040120c      1mov eax, dword_40CF60
```

- 如果发生重定向，前3个指令能够正常工作，因为他们使用相对地址
- 最后一条指令使用绝对地址，如果代码被重定向，它将不能正常工作



# 修订位置

- 多数DLL在PE头文件的 .reloc 节有个修订位置列表
  - 当代码重定位时这些指令必须做相应的改变
- DLL在EXE载入后以任意顺序加载
- 如果存在基地址重定位，你无法预测DLL会被加载到内存中的那个位置





# DLL 基地址重定位

- DLL能够将他们的 `.reloc` 删除
  - 这样的 DLL 不能被重定位
  - 必须加载到预定基地址
- 重定位DLL会影响性能
  - 增加加载时间
  - 因此优秀的程序员在编译DLL时会指定非默认地址



# 011y 内存映射显示DLL 重定位的事例

- DLL-A 和 DLL-B 预定地址都是 0x100000000

00340000	00001000	DLL-B		PE header	Imag	R	RWE
00341000	00009000	DLL-B	.text	code	Imag	R	RWE
0034A000	00002000	DLL-B	.rdata	imports,exp	Imag	R	RWE
0034C000	00003000	DLL-B	.data	data	Imag	R	RWE
0034F000	00001000	DLL-B	.rsrc	resources	Imag	R	RWE
00350000	00001000	DLL-B	.reloc	relocations	Imag	R	RWE
00400000	00001000	EXE-1		PE header	Imag	R	RWE
00401000	00010000	EXE-1	.textbss	code	Imag	R	RWE
00411000	00004000	EXE-1	.text	SFX	Imag	R	RWE
00415000	00002000	EXE-1	.rdata		Imag	R	RWE
00417000	00001000	EXE-1	.data	data	Imag	R	RWE
00418000	00001000	EXE-1	.idata	imports	Imag	R	RWE
00419000	00001000	EXE-1	.rsrc	resources	Imag	R	RWE
10000000	00001000	DLL-A		PE header	Imag	R	RWE
10001000	00009000	DLL-A	.text	code	Imag	R	RWE
1000A000	00002000	DLL-A	.rdata	imports,exp	Imag	R	RWE
1000C000	00003000	DLL-A	.data	data	Imag	R	RWE
1000F000	00001000	DLL-A	.rsrc	resources	Imag	R	RWE
10010000	00001000	DLL-A	.reloc	relocations	Imag	R	RWE

*Figure 10-5. DLL-B is relocated into a different memory address from its requested location*



# IDA Pro

- IDA Pro不是附加到真的运行的进程
- 它不知道基地址重定向
- 如果你同时使用OllyDbg 和 IDA 你会得到不同的结果
  - 为了避免这种情况，使用IDA Pro的“Manual Load” 选项
  - 手动指定虚拟基地址



# 查看线程和堆栈

- View-> Threads
- 右击一个线程 选择“Open in CPU”、“Kill thread”等

T Threads									
Ord	Ident	Window's title	Last error	Entry	TIB	Suspend	Priority	User time	System time
Main	00000F34	Cisco Packet Trac	ERROR_SUCCESS (000		7FFDF000	0.	Normal	1.1544 s	0.2964 s
2.	00000488		ERROR_SUCCESS (000	7029345E	7FFDE000	0.	Normal	0.0000 s	0.0000 s
3.	000007C4		ERROR_SUCCESS (000	76E6EB16	7FFDD000	0.	Normal	0.0000 s	0.0000 s
4.	00000414		ERROR_SUCCESS (000	76E6D34E	7FFDC000	0.	Normal	0.0000 s	0.0000 s
5.	00000A80		ERROR_SUCCESS (000	76E6D34E	7FFDB000	0.	Normal	0.0000 s	0.0000 s
6.	0000093C		ERROR_SUCCESS (000	768DC89D	7FFDA000	0.	Normal	0.0000 s	0.0000 s
7.	000008C8		ERROR_SUCCESS (000	749E6F14	7FFD9000	0.	High	0.0000 s	0.0000 s





# 每个线程都有自己的栈

- 内存映射中可以看出

M Memory map							
Address	Size	Owner	Section	Contains	Type	Access	Initial
05050000	00800000				Priv	RW	RW
05850000	00A80000				Priv	RW	RW
06820000	003FC000				Map	R	R
06D1D000	00002000			Stack of thread 2. (00000488)	Priv	RW	Guar
06D1F000	00001000				Priv	RW	RW
06E1D000	00002000			Stack of thread 3. (000007C4)	Priv	RW	Guar
06E1F000	00001000				Priv	RW	RW
06F10000	00880000				Priv	RW	RW
07AD0000	006B5000				Priv	RW	RW
0828D000	00002000			Stack of thread 4. (00000414)	Priv	RW	Guar
0828F000	00001000				Priv	RW	RW
0838D000	00002000			Stack of thread 5. (00000A80)	Priv	RW	Guar
0838F000	00001000				Priv	RW	RW
0848C000	00002000			Stack of thread 6. (0000093C)	Priv	RW	Guar
0848E000	00002000				Priv	RW	RW
0858D000	00002000			Stack of thread 7. (000008C8)	Priv	RW	Guar
0858F000	00001000				Priv	RW	RW
08630000	00019000				Priv	RW	RW
08670000	0021F000				Map	RW	RW
088B0000	01C57000				Priv	RW	RW
08C10000	001F4000				Priv	RW	RW



# 执行代码



*Table 10-1. OllyDbg Code-Execution Options*

Function	Menu	Hotkey	Button
Run/Play	Debug ► Run	F9	
Pause	Debug ► Pause	F12	
Run to selection	Breakpoint ► Run to Selection	F4	
Run until return	Debug ► Execute till Return	CTRL-F9	
Run until user code	Debug ► Execute till User Code	ALT-F9	
Single-step/step-into	Debug ► Step Into	F7	
Step-over	Debug ► Step Over	F8	



# Run与Pause选项

- 运行一个程序，然后在你想要暂停的地方单击暂停
- 但这是草率的，可能会让你定在不关心的地方，比如停在库代码中
- 设置断点要比暂停好得多



# Run 与 Run to Selection选项

- Run用来在触及断点后恢复
- Run to Selection直到选择的指令执行之前一直执行
  - 如果选择的指令从未执行，则被调试程序将会一直运行下去

# Execute till Return选项



- 在当前函数返回前暂停
- 如果你想结束当前函数并暂停，该选项很有用
- 但是如果这个函数从不返回，被调试程序会一直执行下去



# Execute till User Code选项

- 若调试时你迷失在库代码中，给选项很有用
- 被调试程序将继续运行，直到它到达编译的恶意代码
  - 通常是 `.text` 节



# 单步调试代码

- F7 - 单步进入 (Single-step又叫step-into)
- F8 - 单步跳过 (Step-over)
  - 单步跳过意味着执行所有代码, 但是你看不用看执行过程
- 一些恶意软件被设计用于愚弄你, 通过调用例程并永远不返回, 所以单步跳过会错过最重要的部分





# 断点

Malware Analysis



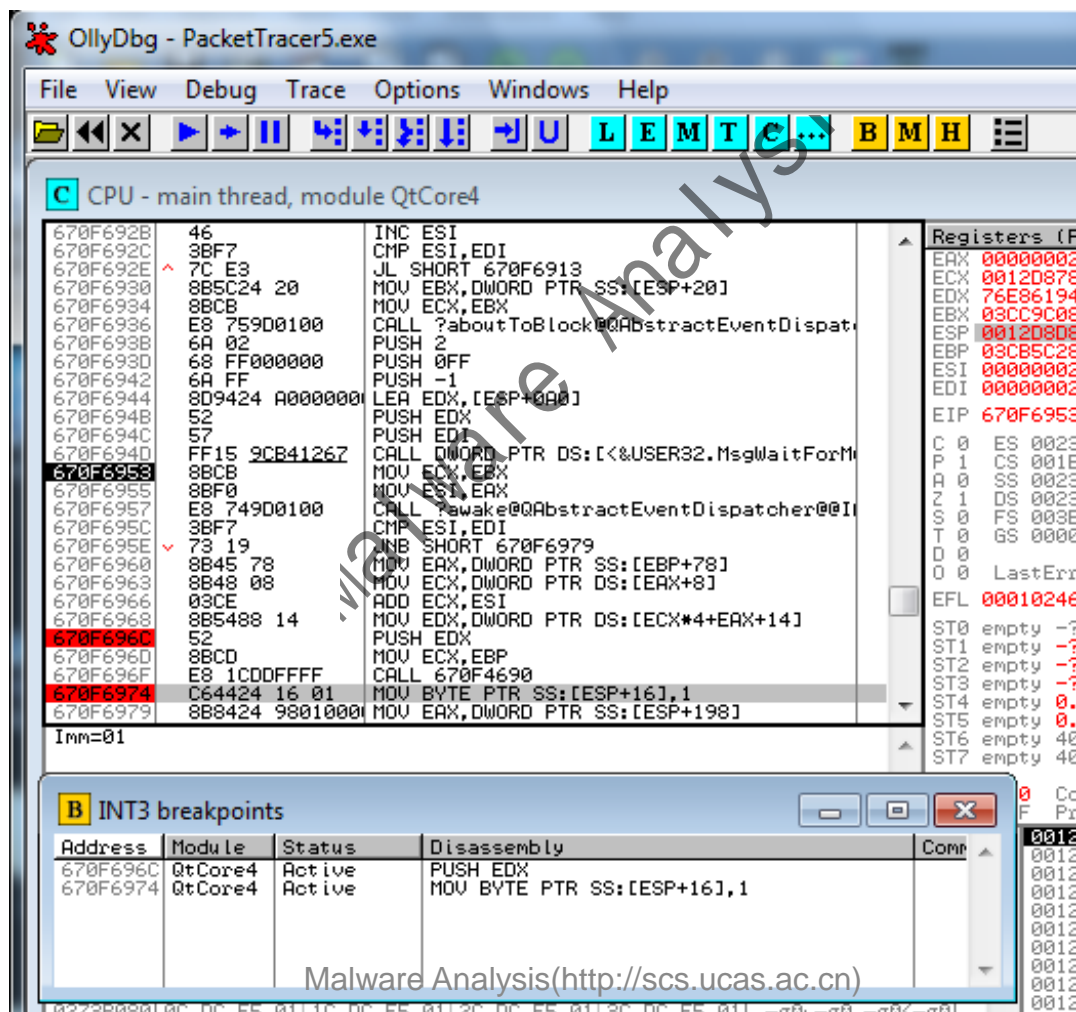
# 断点类型

- 软件断点
  - 硬件断点
  - 条件断点
  - 内存断点
- Malware Analysis
- F2 - 添加或删除断点



# 查看活跃断点

- View->Breakpoints或者点击工具栏上的B图标





*Table 10-2. OllyDbg Breakpoint Options*

Function	Right-click menu selection	Hotkey
Software breakpoint	Breakpoint ► Toggle	F2
Conditional breakpoint	Breakpoint ► Conditional	SHIFT-F2
Hardware breakpoint	Breakpoint ► Hardware, on Execution	
Memory breakpoint on access (read, write, or execute)	Breakpoint ► Memory, on Access	F2 (select memory)
Memory breakpoint on write	Breakpoint ► Memory, on Write	



# 保存断点

- 当你关闭OllyDbg时，它会保存你设置的断点
- 如果你再次打开同一个文件，断点仍然可用



# 软件断点

- 对于调试字符串解码函数很有用
- 恶意代码作者经常混淆字符串
  - 在每个字符串被使用之前，用一个叫做字符串解码器的函数解码

*Example 10-2. A string decoding breakpoint*

```
push offset "4NNpTNHLKIXoPm7iBhUAjvRKNaUVBlr"  
call String_Decoder
```

```
...
```

```
push offset "ugKLdNlLT6emldCeZi72mUjieuBqdfZ"  
call String_Decoder
```

```
...
```



# 字符串解码器

- 将一个断点设置在解码器例程结束位置
- 堆栈中字符串变得易读  
每次你按OllyDbg中的Play，程序将执行并在字符串被解码使用时暂停
- 这个方法只会揭示它们所使用字符串



# 条件断点

- 仅条件为真时暂停
- 如： Poison Ivy 后门
  - Poison Ivy 分配内存用来存储它从命令与控制服务 (C&C) 接收到的shellcode
  - 大多数内存分配是为了其他目的，对调试没有任何意义
  - 在Kernel32.dll中的VirtualAlloc 函数设置条件断点





# 普通断点

- 在VirtualAlloc函数开始处设置标准断点
- 当命中断点时，栈的情况如下，显示5项内容：
  - 返回地址
  - 4个参数(Address, Size, AllocationType, Protect)

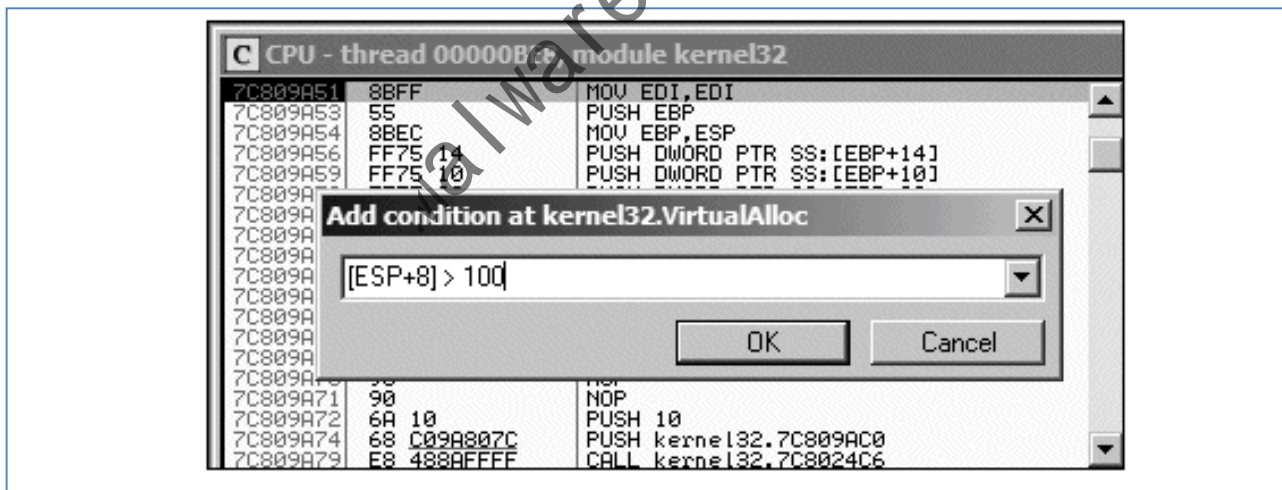
00C3FDB0	0095007C	CALL to VirtualAlloc from 00950079
00C3FDB4	00000000	Address = NULL
00C3FDB8	00000029	Size = 29 (41.)
00C3FDBC	00001000	AllocationType = MEM_COMMIT
00C3FDC0	00000040	Protect = PAGE_EXECUTE_READWRITE

Figure 10-7. Stack window at the start of VirtualAlloc



# 条件断点

- 1、右击反汇编面板窗口中VirtualAlloc函数的第一条指令，选择Breakpoint->Conditional。然后会弹出一个对话框，要求你输入条件表达式
- 2、设置条件表达式并点击OK。在本例中使用[ESP+8]>100
- 3、单击Play按钮并等待断点命中





# 硬件断点

- 不改变代码、堆栈或任何目标资源
- 不会降低代码执行速度
- 但一次只能设置4个
- 点击Breakpoint->Hardware, on Execution
- 可以在“Debugging Options”中设置011yDbg默认使用硬件断点



# 内存断点

- 代码在访问指定内存位置时中断执行
- OllyDbg 支持软件内存断点和硬件内存断点
- 能够在读、写或者任何访问时中断执行
- 右击内存位置点击 Breakpoint->"Memory, on Access"



# 内存断点

- 一次只能设置一个内存断点
- OllyDbg 通过改变内存块的属性来实现软件内存断点
- 这种技术并不完全可靠，同时带来相当大的开销
- 应有节制的使用内存断点



# DLL什么时候被使用？

- 1、打开内存映射面板窗口，右击需要跟踪的DLL的.text段（.text包含DLL的可执行代码）。
- 2、选择Set Memory Breakpoint on Access
- 3、按F9键或者单击Play按钮恢复程序运行

当应用程序运行到DLL的.text段代码时，会中断执行



# 加载DLL



# loaddll.exe

- DLL不能直接被执行
- OllyDbg 使用一个名为 loaddll.exe 虚拟程序来加载它
- 一旦DLL被加载，在DLL入口点（DLLMain）处中断
- 按下Play按钮运行DLLMain 并初始化DLL



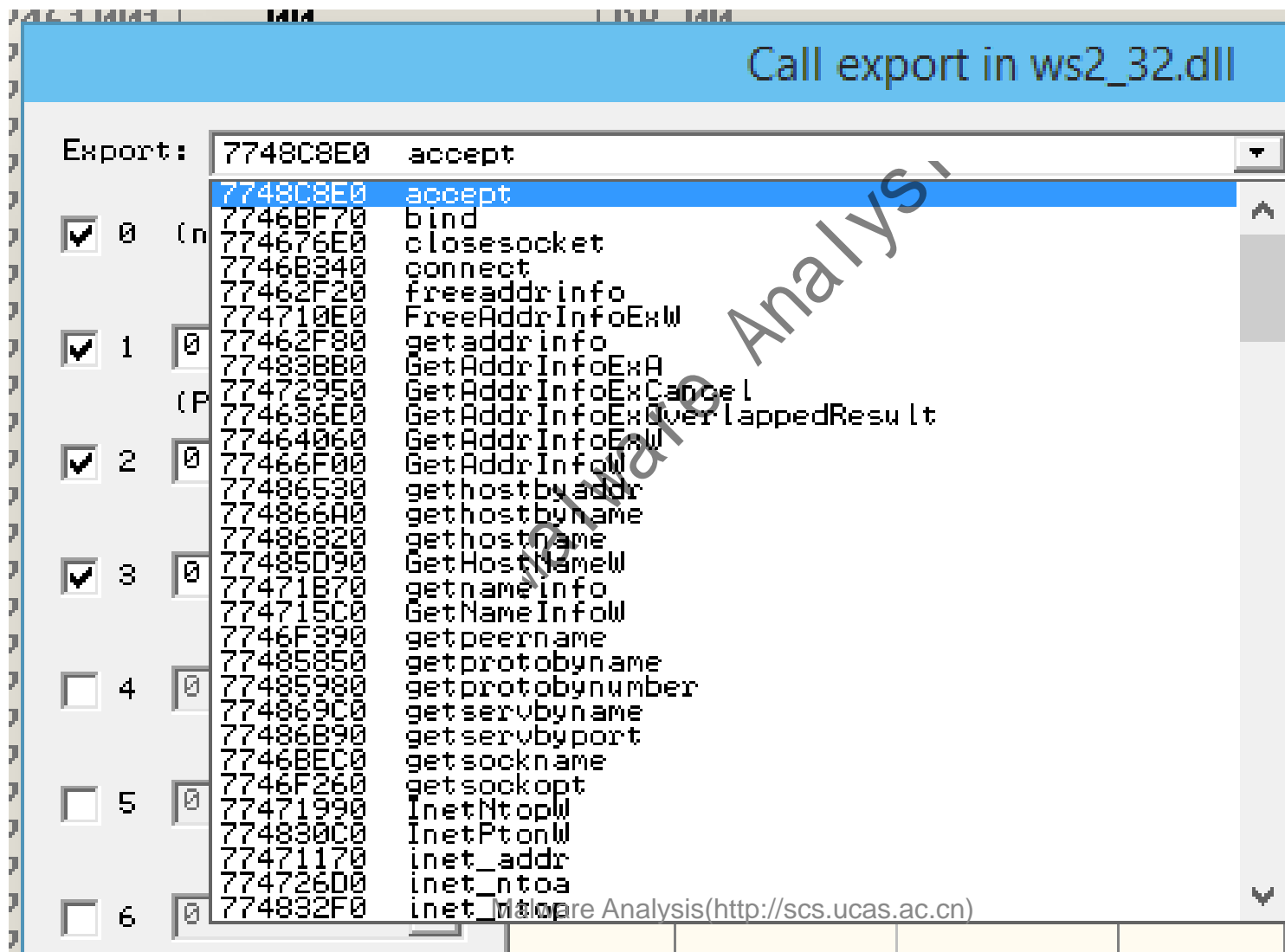


# Demo

- 使用OllDbg 1.10不是2.00 或 2.01
- 使用Win XP 或者 Win 7
- 在OllDbg中打开  
c:\windows\system32\ws2\_32.dll
- Debug-> Call DLL Export - 失败
- 重新加载DLL , 点击一次Run按钮
- Debug-> Call DLL Export - 现在可以了



# 调用ws2\_32.dll中的导出函数





# ntohl

- 将一个32位数字从网络字节序转化为主机字节序
- 点击参数1，输入7f000001
  - 数字127.0.0.1的网络字节序
- 点击 “Follow in Disassembler” 按钮查看代码
- 点击 “Call” 按钮运行函数 to run the function
- 返回结果在EAX中

# Call export in ws2\_32.dll

Export: 754C2F77 ntohl

☒ 0 (no arguments)

Follow in Disassembler

Pure function (no side effects)  
Number of arguments: 1.  
Valid stack frame

☒ 1 7f000001  
(Pushed last)

☐ 2 0

☐ 3 0

☐ 4 0

☐ 5 0

☐ 6 0

☐ 7 0

☐ 8 0

☐ 9 0

☐ 10 0  
(Pushed first)

Value of registers:  
Before call After call

EAX 0 0100007F

ECX 0 00007F00

EDX 0 0000007F

EBX 0 00000000

ESI 0 00000000

EDI 0 00000000

Done

☐ Hide on call

Call

☐ Pause after call

Close



# 跟踪

Malware Analysis



# 跟踪

- 一种强大的调试技术
- 记录详细的运行信息
- 跟踪的类型
  - 标准回溯跟踪
  - 堆栈调用跟踪
  - 运行跟踪



# 标准回溯跟踪

- 在反汇编面板窗口执行Step Into 和 Step Over操作
- OllyDbg会记录这种操作
- 使用减号键（-）查看以前的指令
  - 单不能查看之前的寄存器值
- 使用加号键（+）查看下一条指令
  - 若你使用Step Over但不能反回后再决定使用step into



# 堆栈调用跟踪

- 查看一个给定函数的执行路径
- 点击 View->Call Stack
- 显示到达当前位置的调用顺序



**C** CPU - thread 00000F20, module CFGMGR32

```

75C95FF7  6A 18      PUSH 18
75C95FF9  68 7060C975  PUSH CFGMGR32.75C96070
75C95FFE  E8 29B2FFFF  CALL CFGMGR32.75C9122C
75C96003  33FF      XOR EDI,EDI
75C96005  897D E4     MOV DWORD PTR SS:[EBP-1C],EDI
75C96008  897D E0     MOV DWORD PTR SS:[EBP-20],EDI
    
```

Registers (FPU)

```

EAX 000000C0
ECX 001B1170
EDX 2E3FA105
EBX 001B49E8
ESP 0177F704
    
```

**K** Call stack of thread 00000F20

Address	Stack	Procedure / arguments	Called from	Frame
0177F704	77AC43FC	Includes ntdll.KiFastSystemCallRet	ntdll.77AC43FA	0177F704
0177F708	75F50346	ntdll.ZwAlpcConnectPort	RPCRT4.75F50340	0177F708
0177F7D4	75F4F51E	RPCRT4.75F501D0	RPCRT4.75F4F519	0177F7D4
0177F804	75F4F3FE	RPCRT4.75F4F418	RPCRT4.75F4F3F9	0177F804
0177F838	75F3846D	RPCRT4.75F4F266	RPCRT4.75F38468	0177F838
0177F888	75F4BC18	Includes RPCRT4.75F3846D	RPCRT4.75F4BC18	0177F888
0177F8AC	75F49D6D	RPCRT4.I_RpcGetBufferWithObject	RPCRT4.75F49D68	0177F8AC
0177F8BC	75F4A041	RPCRT4.I_RpcGetBuffer	RPCRT4.75F4A03C	0177F8BC
0177F8CC	75FA5718	Includes RPCRT4.75F4A041	RPCRT4.75FA5712	0177F8CC
0177FCF0	75C960B6	? <JMP.&RPCRT4.NdrClientCall2>	CFGMGR32.75C960B1	0177FCF0
0177FD08	75C96055	CFGMGR32.75C9609D	CFGMGR32.75C96050	0177FD08
0177FD5C	762E0356	? CFGMGR32.CM_Get_Device_Interface_L	SHELL32.762E0350	0177FD5C
0177FD98	762E02ED	SHELL32.762E0326	SHELL32.762E02E8	0177FD98
0177FDA8	7709B6CF	Includes SHELL32.762E02ED	SHLWAPI.7709B6C0	0177FDA8
0177FDB8	77AAB338	Includes SHLWAPI.7709B6CF	ntdll.77AAB335	0177FDB8

Process terminated, exit code 0

Terminated



# 运行跟踪

- 代码运行时, OllyDbg 保存每条执行的命令和所有对寄存器及标志位所做的修改
- 高亮代码, 右击Run Trace-> Add Selection
- 代码执行后, View-> Run Trace
  - 查看被执行的指令
  - + 、 - 键, 向前、向后



File View Debug Plugins Options Window Help



CPU - main thread, module Lab09-01

```

004038C2 33D2 XOR EDX,EDX
004038C4 8AD4 MOV DL,AH
004038C6 8915 7CEB4000 MOV DWORD PTR DS:[40EB7C1],EDX
004038CC 8BC8 MOV ECX,EAX
004038CE 81E1 FF000000 AND ECX,0FF
004038D4 890D 78EB4000 MOV DWORD PTR DS:[40EB781],ECX
004038DA C1E1 08 SHL ECX,8
004038DD 03CA ADD ECX,EDX
004038DF 890D 74EB4000 MOV DWORD PTR DS:[40EB741],ECX
004038E5 C1E8 10 SHR EAX,10
004038E8 A3 70EB4000 MOV DWORD PTR DS:[40EB701],EAX
004038ED 6A 00 PUSH 0
004038EF E8 612A0000 CALL Lab09-01.00406355
004038F4 59 POP ECX
004038F5 85C0 TEST EAX,EAX
004038F7 75 08 JNZ SHORT Lab09-01.00403901
004038F9 6A 1C PUSH 1C
004038FB E8 9A000000 CALL Lab09-01.0040399A
00403900 59 POP ECX

```

Run trace

Back	Thread	Module	Address	Command	Modified registers
9.	Main	Lab09-01	004038C4	MOV DL,AH	EDX=00000002
8.	Main	Lab09-01	004038C6	MOV DWORD PTR DS:[40EB7C1],EDX	
7.	Main	Lab09-01	004038CC	MOV ECX,EAX	ECX=23F00206
6.	Main	Lab09-01	004038CE	AND ECX,0FF	ECX=00000006
5.	Main	Lab09-01	004038D4	MOV DWORD PTR DS:[40EB781],ECX	
4.	Main	Lab09-01	004038DA	SHL ECX,8	ECX=00000600
3.	Main	Lab09-01	004038DD	ADD ECX,EDX	ECX=00000602
2.	Main	Lab09-01	004038DF	MOV DWORD PTR DS:[40EB741],ECX	
1.	Main	Lab09-01	004038E5	SHR EAX,10	EAX=000023F0
0.		Lab09-01	004038E8	MOV DWORD PTR DS:[40EB701],EAX	



# Trace Into 与 Trace Over 选项

- Buttons below "Options"
- 比 Add Selection 更容易使用
- 若你没有设置断点 OllyDbg 会试图跟踪整个被调试程序，这会需要很长时间和大量内存



# Debug-> Set Condition

- 跟踪直到条件命中
- 为获取 Poison Ivy 的 shellcode 设置的条件，Poison Ivy 将代码放置在低于 0x400000 的动态分配的内存中

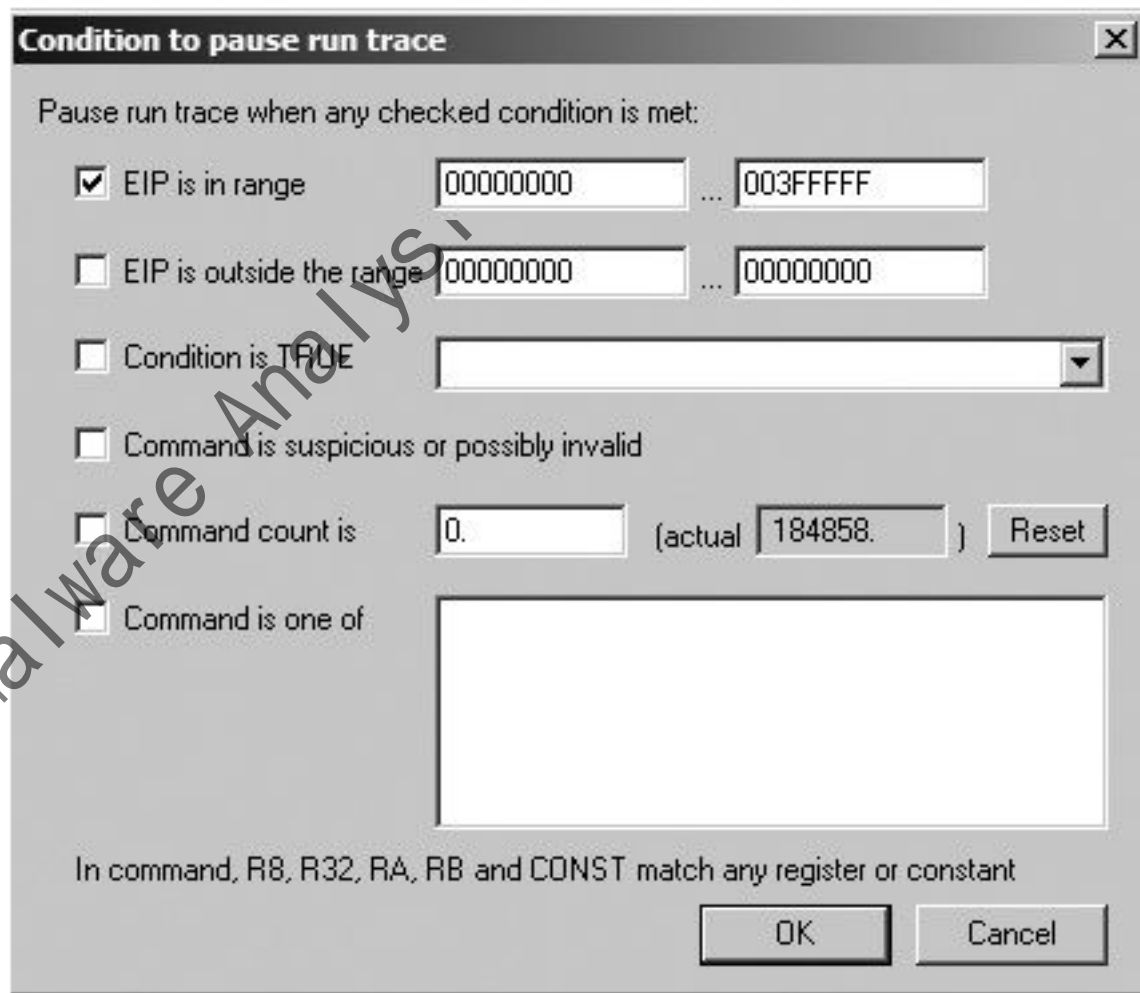


Figure 10-11. Conditional tracing



# 异常处理



# 当产生异常时

- OllyDbg 会暂停被调试程序
- 有这些选项将异常转到被调试的程序：
  - Shift+F7 进入异常
  - Shift+F8 跳过异常
  - Shift+F9 运行异常处理
- 恶意代码分析时，通常忽略所有异常
  - 我们并不想修复代码中的问题



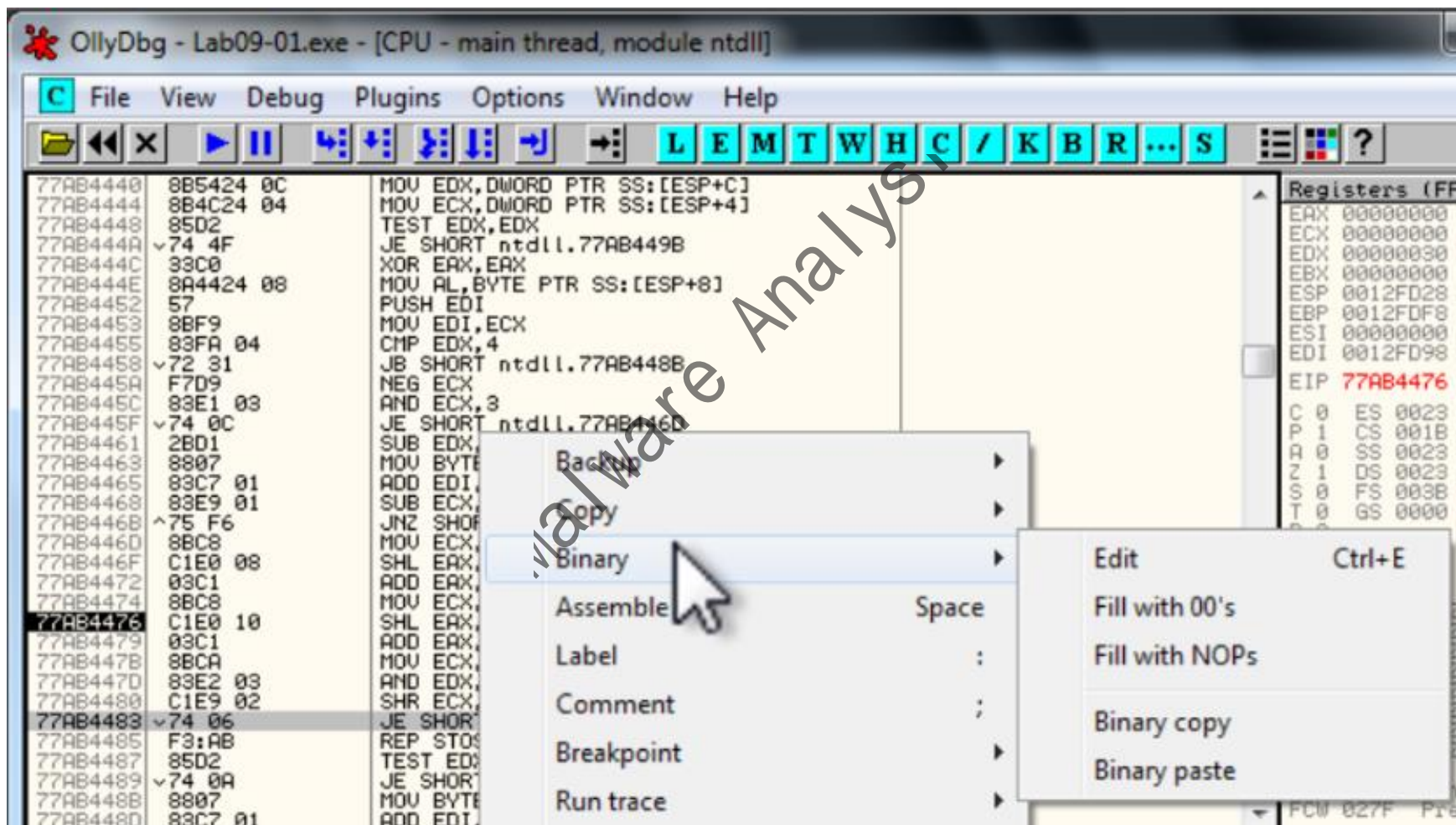
修补

Malware Analysis





# Binary -> Edit





# 填充

- 用00填充
- 用NOP (0x90) 填充
  - 用于跳过一些指令
  - 例如强制执行一个分支



# 保存修补代码

- 修改后，右击反汇编面板
  - 选择Copy To Executable->All Modifications
  - 复制全部
- 在新窗口中右击
  - 选择Save File



# 分析 Shellcode



# 分析Shellcode的简单方法

- 从十六进制编辑器中拷贝到剪切板
- 在内存映射面板窗口，选择类型为“Priv”的内存区域(进程私有内存)
- 双击内存映射面板窗口的某行，显示十六进制转储窗口
  - 找到一区域，该区域有个数百个连续为0的字节
- 右击内存映射中选择的区域设置 Set Access→ Full Access (清除NX 位)



# 分析Shellcode

- 高亮该0值区域, Binary->Binary Paste
- 设置 EIP 指向 shellcode位置
  - 右击第一条指令选择New Origin Here

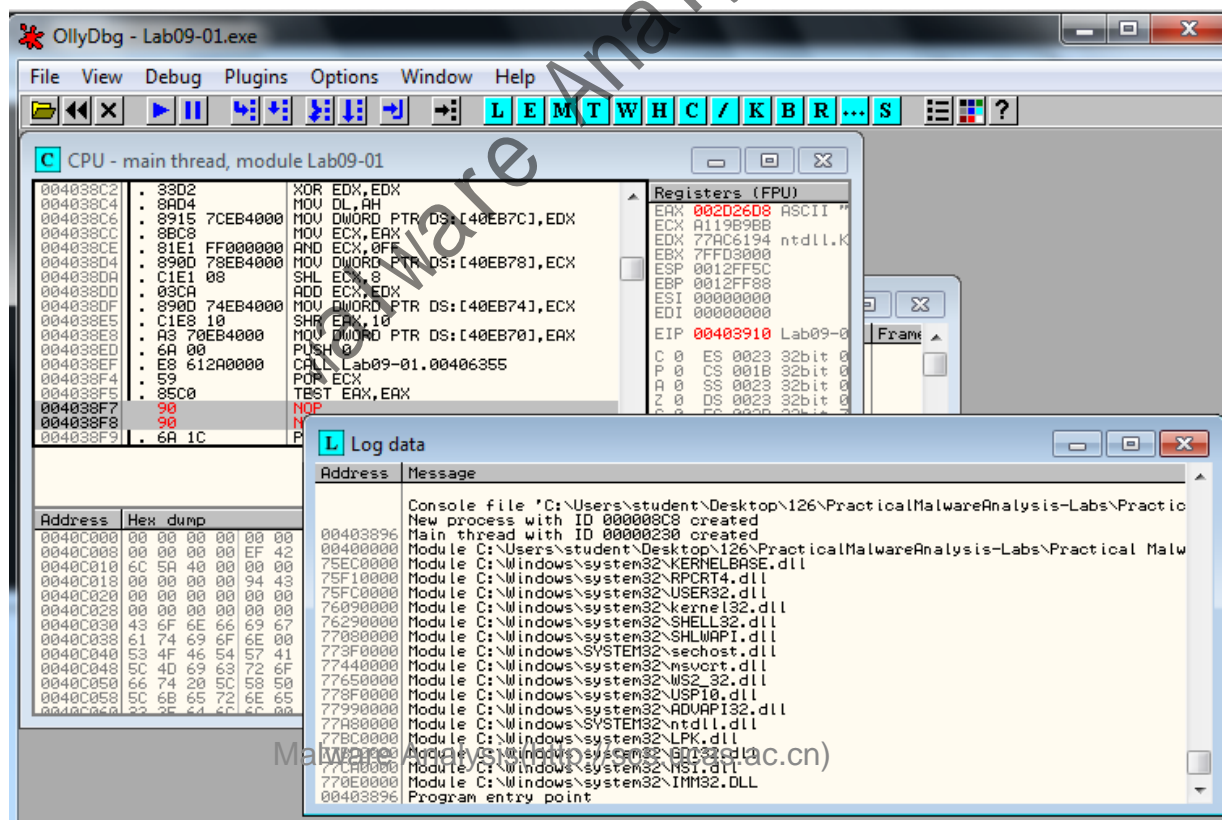


# 帮助功能



# 日志

- View-> Log
  - 显示到达当前位置的步骤

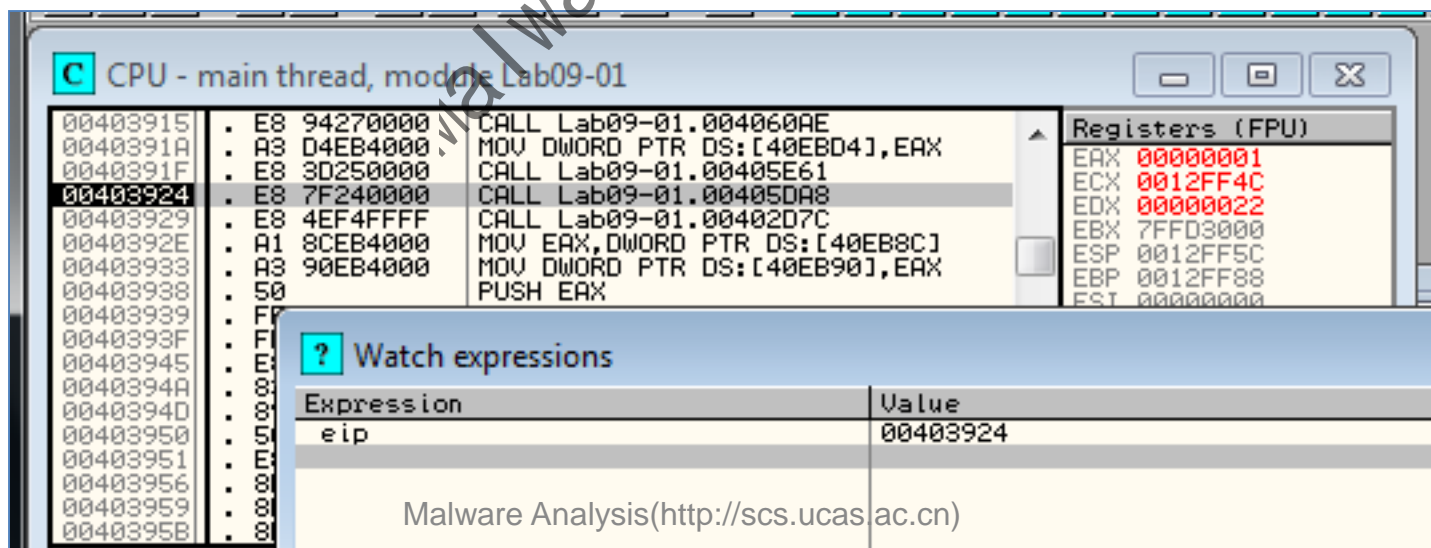






# 监视 (Watches) 窗口

- View->Watches
  - 查看表达式的值
  - 按空格键设置表达式
  - OllyDbg 的 Help->Contents
    - 为表达式的书写提供了详细说明





# 标注

- 为子例程和循环添加标注
  - 右击一个地址，选择Label

Malware Analysis



# 插件

Malware Analysis

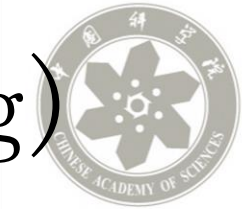


# 推荐插件

- OllyDump
  - 将被调试进程转储成一个PE文件
  - 用于脱壳
- Hide Debugger
  - 隐藏OllyDbg避免被调试器检测到
- Command Line
  - 从命令行控制 OllyDbg
  - 简单像在用WinDbg
- Bookmarks
  - OllyDbg默认包含该插件
  - 将内存位置加到书签中



# 脚本调试

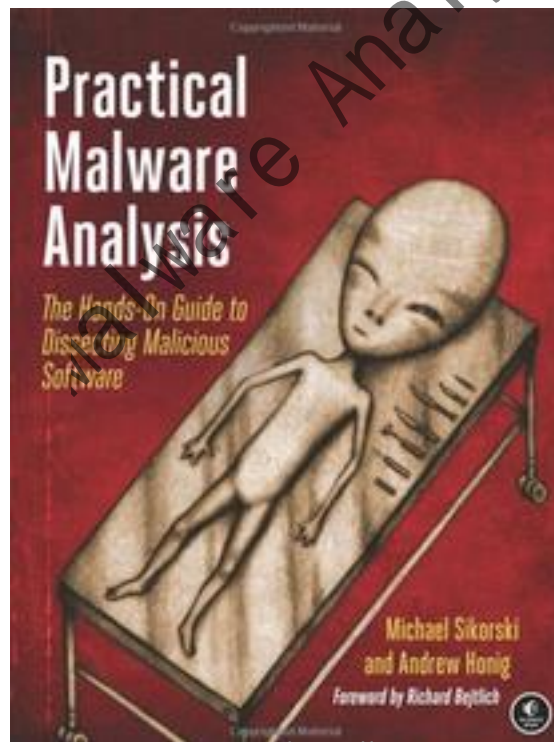


# Immunity Debugger (ImmDbg)

- 不像OllyDbg, ImmDbg 使用python 脚本并有易于使用的 API
- 脚本位于ImmDbg的安装目录下PyCommands的子目录中
- 为ImmDbg创建自定义脚本比较容易



## 5.3: 使用 WinDbg调试内核





# WinDbg 与 OllyDbg

- OllyDbg 最受恶意代码分析者欢迎的用户模式调试器
- WinDbg可用于用户模式和内核模式
- 本章探讨了使用WinDbg调试内核的方法和rookit技术分析





# 驱动和内核代码



# 设备驱动

- Windows设备驱动允许第三方开发商在Windows内核模式下运行代码
- 分析驱动十分困难
  - 它们常驻内存，并负责响应应用程序的请求
- 应用程序不直接访问内核驱动程序
  - 他们访问设备对象，由设备对象发送请求到特定设备



# 设备

- 设备不是物理的硬件组件
- 它们是这些组件的软件表示
- 驱动程序创建和销毁设备，这些设备可以从用户空间访问



# USB 闪存驱动

- 用户插入USB设备
- Windows 创建设备对象 “F:”
- 应用程序现在可以发出请求到 F:
  - 该请求将被发送到USB闪存驱动程序



# 加载驱动

- 驱动必须加载到内核空间
  - 正如dll加载到进程空间
- 当驱动程被首次加载序时，其DriverEntry函数将被调用
  - 正如DLL中的DLLMain



# DriverEntry

- DLL通过函数导出表提供其功能接口
- 驱动必须注册回调函数
  - 当用户态软件组件请求一个服务时，回调函数将被调用
  - DriverEntry执行这种注册
  - Windows创建一个驱动对象，并将其传递到DriverEntry， DriverEntry用回调函数填充该驱动对象
  - 然后DriverEntry 创建一个可以从用户态访问的设备



# 事例： 标准读操作

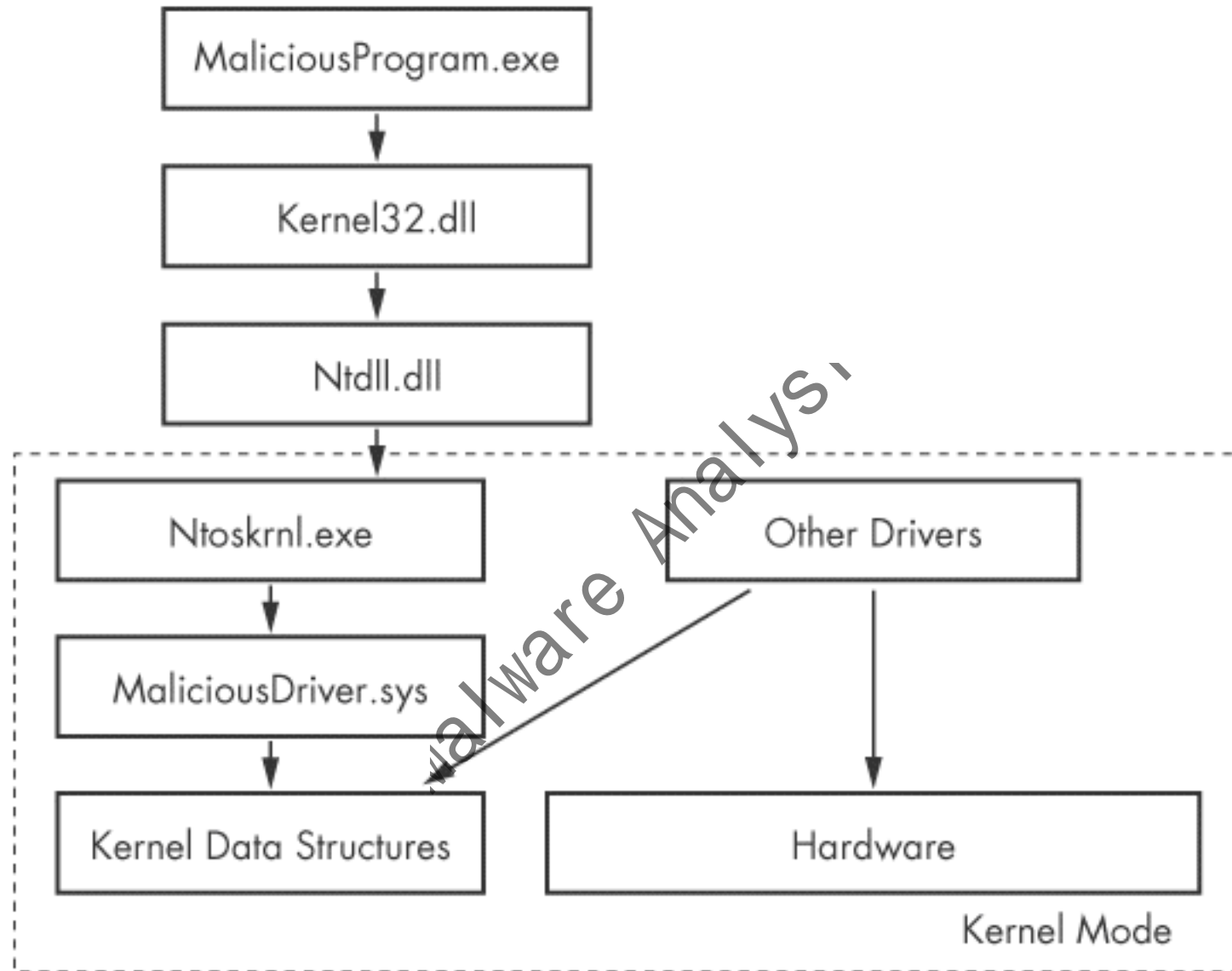
- 标准读请求
  - 用户态应用程序获取一个设备的文件句柄
  - 在该句柄上调用 ReadFile函数
  - 内核处理ReadFile请求
  - 调用驱动的回调函数处理I/O



# 恶意请求

- 从恶意代码发起的最常见的请求是 DeviceIoControl
  - 从用户态模块到内核设备的一种通用请求
  - 用户态程序传递任意长度的缓冲区数据作为输入
  - 接收一个任意长度缓冲区的数据作为输出





*Figure 11-1. How user-mode calls are handled by the kernel*

# Ntoskrnl.dll 和 Hal.dll



- 恶意驱动很少控制硬件
- 它们与Ntoskrnl.dll 和 Hal.dll交互
  - Ntoskrnl.dll 包含操作系统核心功能代码
  - Hal.dll 包含与主要硬件设备交互的代码
- 恶意代码将从它们中的一个或两个文件中导入函数，来操纵内核



# 安装内核调试

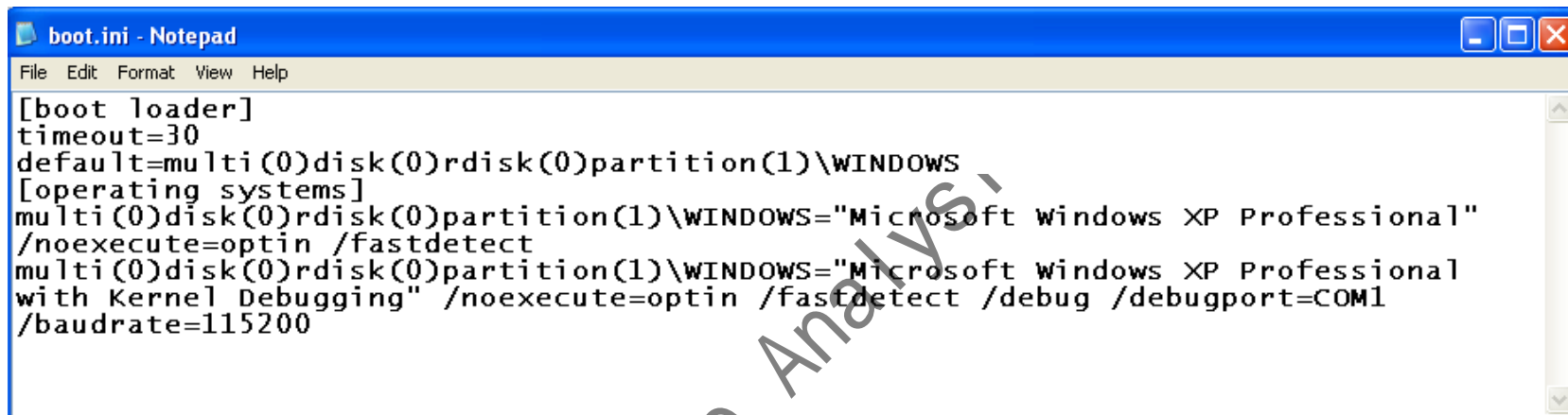


# VMware

- 在虚拟机中开启内核调试
- 在虚拟机与宿主机之间配置一个虚拟化的串口
- 配置宿主机中的 WinDbg



# 编辑C:\Boot.ini

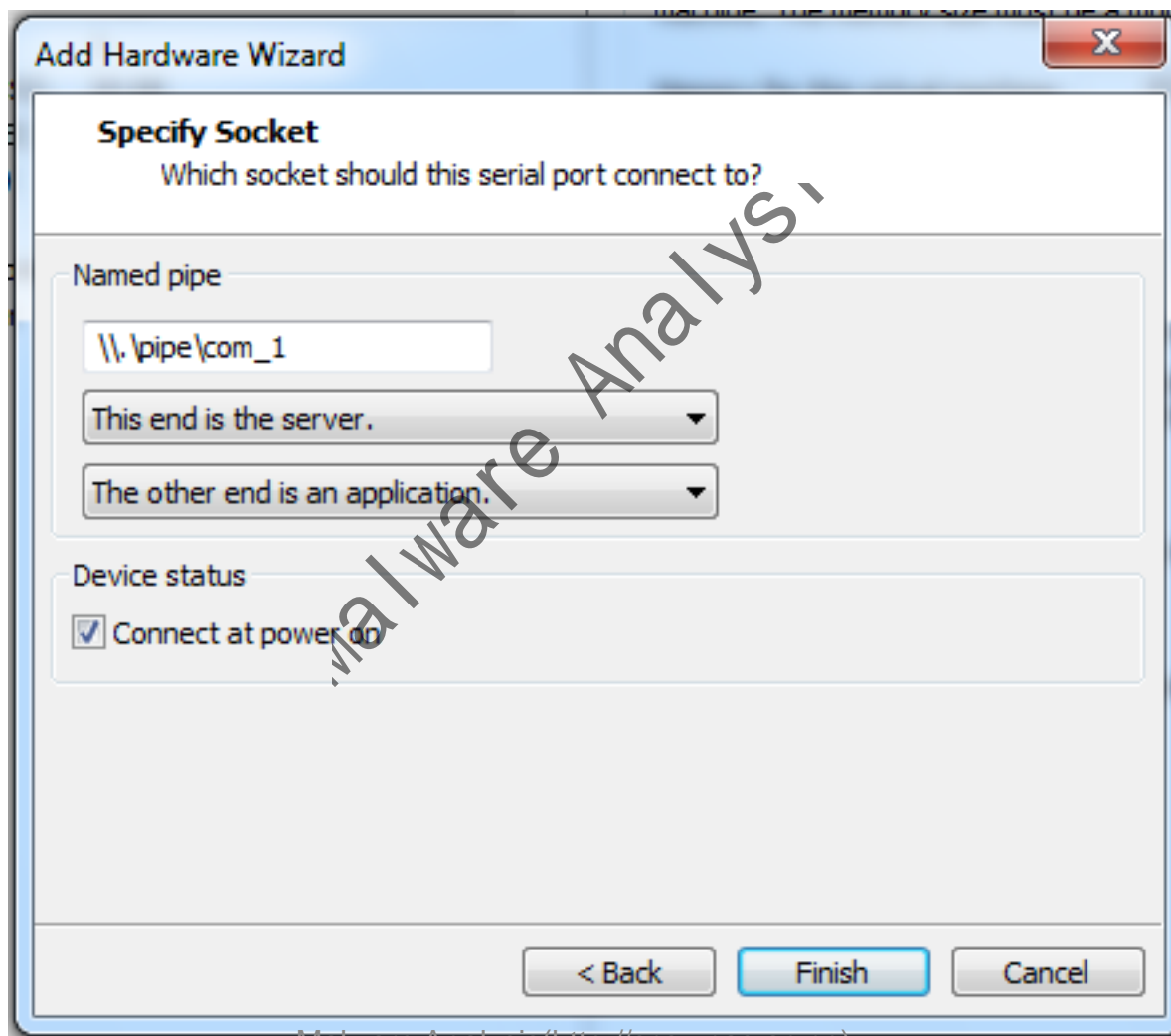


The screenshot shows a Notepad window titled 'boot.ini - Notepad'. The menu bar includes 'File', 'Edit', 'Format', 'View', and 'Help'. The text content of the file is as follows:

```
[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
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional
with Kernel Debugging" /noexecute=optin /fastdetect /debug /debugport=COM1
/baudrate=115200
```

- 第二启动项允许调试
- 这种方式不适用于 Windows 7

# 添加虚拟化串口





# 使用WinDbg

命令行命令



# 从内存中读取

- dx 地址
- x 可以是
  - da 以ASCII文本显示
  - du 以Unicode文本显示
  - dd 以32位双字节显示
- da 0x401020
  - 以ASCII文本显示从地址0x401020开始的字符串





# 编辑内存

- `ex addressToWrite dataToWrite`
- `x` 可以是
  - `ea` 以ASCII文本写入
  - `eu` 以Unicode文本写入
  - `ed` 以32位双字节写入



# 使用算术操作符

- 简单的算术操作符+ - / \*
- dwo 查看一个32位地址指针的值r
- du dwo (esp+4)
  - 以宽字节字符串显示函数的第一个参数



# 设置断点

- bp 设置基本断点
- 当断点命中时，可以指定将被执行的操作
- g 完成操作后继续执行
- bp GetProcAddress "da dwo(esp+8); g"
  - 当GetProcAddress被调用时中断，打印函数第二个参数，然后继续执行
  - 函数的第二个参数是函数名



# 列举模块

- lm
  - 列举加载到进程的所有模块
    - 包括用户模式的EXE和DLL
    - 和内核模式的内核驱动
  - 如同011yDbg中的内存映射



# 微软符号表



# 符号表就是标注

- 符号表使用函数名
  - MmCreateProcessAddressSpace
  - 替代地址
  - 0x8050f1a2



# 搜索符号

- `moduleName!symbolName`
  - 可以用在任意一个需要的地址处
- `moduleName`
  - 表示 .EXE、.DLL 或者 .SYS 文件名 (不含扩展名)
- `symbolName`
  - 与这个地址相关联的名字
- `ntoskrnl.exe` 是个特例，它被命名为 `nt`
  - 例如：`u nt!NtCreateProcess`
    - 反汇编该函数



# 延时断点

- `bu newModule!exportedFunction`
  - 一旦newModule模块加载，在exportedFunction设置断点
- `$iment`
  - 查找模块入口点的函数
- `bu $iment(driverName)`
  - 在驱动代码执行前，驱动入口点处中断





# 使用x查找

- 可以使用通配符查找函数或符号
- x nt!\*CreateProcess\*
  - 显示导出函数和内部函数

```
0:003> x nt!*CreateProcess*
805c736a nt!NtCreateProcessEx = <no type information>
805c7420 nt!NtCreateProcess = <no type information>
805c6a8c nt!PspCreateProcess = <no type information>
804fe144 nt!ZwCreateProcess = <no type information>
804fe158 nt!ZwCreateProcessEx = <no type information>
8055a300 nt!PspCreateProcessNotifyRoutineCount = <no type information>
805c5e0a nt!PsSetCreateProcessNotifyRoutine = <no type information>
8050f1a2 nt!MmCreateProcessAddressSpace = <no type information>
8055a2e0 nt!PspCreateProcessNotifyRoutine = <no type information>
```



# 使用ln列出最接近的符号

- 帮助确认指针指向的函数
- ln address
  - 第一行显示两个相近的匹配
  - 最后一行显示精确匹配

```
0:002> ln 805717aa
kd> ln ntreadfile
1 (805717aa)  nt!NtReadFile    | (80571d38)  nt!NtReadFileScatter
Exact matches:
2      nt!NtReadFile = <no type information>
```



# 使用dt查看结构信息

- 微软符号表包含多个数据结构的类型信息
  - 包括未公开的内部类型
  - 恶意代码经常使用未公开的数据结构
- `dt moduleName!symbolName`
- `dt moduleName!symbolName address`
  - 从地址address显示结构中的数据



## *Example 11-2. Viewing type information for a structure*

```
0:000> dt nt!_DRIVER_OBJECT
```

```
kd> dt nt!_DRIVER_OBJECT
```

```
+0x000 Type           : Int2B
+0x002 Size           : Int2B
+0x004 DeviceObject    : Ptr32 _DEVICE_OBJECT
+0x008 Flags           : Uint4B
1 +0x00c DriverStart    : Ptr32 Void
+0x010 DriverSize      : Uint4B
+0x014 DriverSection   : Ptr32 Void
+0x018 DriverExtension : Ptr32 _DRIVER_EXTENSION
+0x01c DriverName      : _UNICODE_STRING
+0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
+0x028 FastIoDispatch  : Ptr32 _FAST_IO_DISPATCH
+0x02c DriverInit      : Ptr32      long
+0x030 DriverStartIo   : Ptr32      void
+0x034 DriverUnload    : Ptr32      void
+0x038 MajorFunction   : [28] Ptr32      long
```



### *Example 11-3. Overlaying data onto a structure*

```
kd> dt nt!_DRIVER_OBJECT 828b2648
+0x000 Type           : 4
+0x002 Size           : 168
+0x004 DeviceObject    : 0x828b0a30 _DEVICE_OBJECT
+0x008 Flags           : 0x12
+0x00c DriverStart     : 0xf7adb000
+0x010 DriverSize      : 0x1080
+0x014 DriverSection   : 0x82ad8d78
+0x018 DriverExtension : 0x828b26f0 _DRIVER_EXTENSION
+0x01c DriverName      : _UNICODE_STRING "\Driver\Beep"
+0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING
"\REGISTRY\MACHINE\
HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch  : (null)
+0x02c DriverInit      : 0xf7adb66c long Beep!DriverEntry+0
+0x030 DriverStartIo   : 0xf7adb51a void Beep!BeepStartIo+0
+0x034 DriverUnload    : 0xf7adb620 void Beep!BeepUnload+0
+0x038 MajorFunction   : [28] 0xf7adb46a long Beep!BeepOpen+0
```



# 初始化函数

- 当驱动加载时，DriverInit函数被首先调用
- 恶意代码有时会将全部恶意载荷放到该函数中



# 配置 Windows 符号表

- 如果调试机一直连接互联网，可以配置 WinDbg 在需要时自动从微软下载符号表
- 符号表被缓存在本地
- File-> Symbol File Path
  - SRC\*c:\websymbols\*http://msdl.microsoft.com/download/symbols

# 手动下载符号表



← → ↻ 🏠 msdn.microsoft.com/en-us/windows/hardware/gg463028.aspx 4\* 🔍 ☆ 🗲 ⚙ 🖨 📄 📧 ☰

## Download Windows Symbol Packages

The easiest way to get Windows symbols is to use the [Microsoft Symbol Server](#). The symbol server makes symbols available to your debugging tools as needed. After a symbol file is downloaded from the symbol server it is cached on the local computer for quick access.

If you prefer to download the entire set of symbols for Windows 8.1 Preview, Windows Server 2012 R2 Preview, Windows 8, Windows Server 2012, Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, or Windows 2000, then you can download a symbol package and install it on your computer.





# 内核调试实践



# 内核模式和用户模式函数

- 研究一个从内核空间写文件的程序
  - 内核模式程序不能调用用户模式程序如：  
CreateFile 和 WriteFile
  - 必须使用NtCreateFile 和NtWriteFile



# 用户空间的代码

## *Example 11-4. Creating a service to load a kernel driver*

```
04001B3D  push    esi                ; lpPassword
04001B3E  push    esi                ; lpServiceStartName
04001B3F  push    esi                ; lpDependencies
04001B40  push    esi                ; lpdwTagId
04001B41  push    esi                ; lpLoadOrderGroup
04001B42  push    [ebp+lpBinaryPathName] ; lpBinaryPathName
04001B45  push    1                  ; dwErrorControl
04001B47  push    3                  ; dwStartType
04001B49  push    01                 ; dwServiceType
04001B4B  push    0F01FFh            ; dwDesiredAccess
04001B50  push    [ebp+lpDisplayName] ; lpDisplayName
04001B53  push    [ebp+lpDisplayName] ; lpServiceName
04001B56  push    [ebp+hSCManager] ; hSCManager
04001B59  call    ds:__imp__CreateServiceA@52
```

- 使用CreateService函数创建一个服务
- dwServiceType 是 0x01 （内核驱动）



# 用户空间的代码

*Example 11-5. Obtaining a handle to a device object*

```
04001893      xor     eax, eax
04001895      push    eax                ; hTemplateFile
04001896      push    80h               ; dwFlagsAndAttributes
0400189B      push    2                 ; dwCreationDisposition
0400189D      push    eax                ; lpSecurityAttributes
0400189E      push    eax                ; dwShareMode
0400189F      push    ebx                ; dwDesiredAccess
040018A0      push    edi                ; lpFileName
040018A1      call    esi ; CreateFileA
```

- 没有显示 edi 被设置为
  - \\.\FileWriter\Device



# 用户空间的代码

- 一旦恶意代码拥有设备句柄，它会使用1处的DeviceIoControl函数向内核驱动发送数据

Once the malware has a handle to the device, it uses the `DeviceIoControl` function at **1** to send data to the driver as shown in **Example 11-6**.

*Example 11-6. Using `DeviceIoControl` to communicate from user space to kernel space*

```
04001910  push    0                ; lpOverlapped
04001912  sub     eax, ecx
04001914  lea     ecx, [ebp+BytesReturned]
0400191A  push    ecx              ; lpBytesReturned
0400191B  push    64h              ; nOutBufferSize
0400191D  push    edi              ; lpOutBuffer
0400191E  inc     eax
0400191F  push    eax              ; nInBufferSize
04001920  push    esi              ; lpInBuffer
04001921  push    9C402408h        ; dwIoControlCode
04001926  push    [ebp+hObject]    ; hDevice
0400192C  call    ds:DeviceIoControl1
```



# 内核模式的代码

- 设置开启内核模式调试器详细信息输出模式
  - 可以看到加载的每一个内核模块
  - 内核模块加载和卸载并不频繁
    - 任何加载模块都值得怀疑
    - 虚拟机中Kmixer.sys除外
- 下面的例子中，从内核调试窗口看到

In the following example, we see that the *FileWriter.sys* driver has been loaded in the kernel debugging window. Likely, this is the malicious driver.

```
ModLoad: f7b0d000 f7b0e780  FileWriter.sys
```



# 内核模式的代码

- !drvobj 命令显示驱动对象

*Example 11-7. Viewing a driver object for a loaded driver*

```
kd> !drvobj FileWriter
Driver object (f827e3698) is for:
Loading symbols for f7b0d000 FileWriter.sys -> FileWriter.sys
*** ERROR: Module load completed but symbols could not be loaded for
FileWriter.sys
\Driver\FileWriter
Driver Extension List: (id , addr)

Device Object list:
826eb030
```



# 内核模式的代码

- dt 命令显示数据结构

*Example 11-8. Viewing a device object in the kernel*

```
kd>dt nt!_DRIVER_OBJECT 0x827e3698
```

```
nt!_DRIVER_OBJECT
```

```
+0x000 Type           : 4
+0x002 Size           : 168
+0x004 DeviceObject   : 0x826eb030 _DEVICE_OBJECT
+0x008 Flags           : 0x12
+0x00c DriverStart     : 0xf7b0d000
+0x010 DriverSize      : 0x1780
+0x014 DriverSection   : 0x828006a8
+0x018 DriverExtension : 0x827e3740 _DRIVER_EXTENSION
+0x01c DriverName      : _UNICODE_STRING "\Driver\FileWriter"
+0x024 HardwareDatabase : 0x8066ecd8 _UNICODE_STRING
"\REGISTRY\MACHINE\
                                HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch  : (null)
+0x02c DriverInit      : 0xf7b0dfcd      long +0
+0x030 DriverStartIo   : (null)
+0x034 DriverUnload    : 0xf7b0da2a      void +0
+0x038 MajorFunction   : [28] 0xf7b0da06      long +0
```





# 内核模式的文件名

- 跟踪这个函数，它最后创建如下名称的文件
  - \DosDevices\C:\secretfile.txt
- 这是一个完全合格的对象名
  - 识别根设备，通常使用 \DosDevices



# 查找驱动对象

- 应用程序与设备交互，而不是驱动
- 查看用户态应用程序，识别感兴趣的设备对象
- 使用用户模式设备对象查找内核模式驱动对象
- 使用 `!devobj` 找到更多关于驱动对象的信息
- 使用 `!devhandles` 查找使用该驱动的应用程序



# Rootkit

Malware Analysis



# Rootkit 基础

- Rootkits 通过修改操作系统内部函数，来隐藏自己
  - 隐藏运行程序的进程、网络连接以及其他资源
  - 反病毒产品、管理员和安全分析员难于发现它们的恶意行为
- 大部分rootkit修改内核
- 最流行的方式：
  - 系统服务描述表（SSDT）挂钩技术



# 系统服务描述表 (SSDT)

- 微软内部使用
  - 查找进入内核的函数调用
  - 通常不被第三方应用程序或驱动访问
- 用户态访问内核代码只有三种方式
  - SYSCALL
  - SYSENTER
  - INT 0x2E



# SYSENTER

- 在XP及以后的 Windows版本中使用
- 函数代码存在 EAX 寄存器中



# ntdll.dll中的一个事例

*Example 11-11. Code for NtCreateFile function*

```
7C90D682 1mov     eax, 25h          ; NtCreateFile
7C90D687  mov     edx, 7FFE0300h
7C90D68C  call    dword ptr [edx]
7C90D68E  retn    2Ch
```

The call to `dword ptr [edx]` will go to the following instructions:

```
7c90eb8b 8bd4  mov     edx, esp
7c90eb8d 0f34  sysenter
```

- EAX 设置为0x25
- 栈指针保存到 EDX中
- SYSENTER 被调用



# SSDT 表入口

*Example 11-12. Several entries of the SSDT table showing NtCreateFile*

```
SSDT[0x22] = 805b28bc (NtCreateDirectoryObject)
SSDT[0x23] = 80603be0 (NtCreateEvent)
SSDT[0x24] = 8060be48 (NtCreateEventPair)
SSDT[0x25] = 8056d3ca (NtCreateFile)
SSDT[0x26] = 8056bc5c (NtCreateIoCompletion)
SSDT[0x27] = 805ca3ca (NtCreateJobObject)
```

- Rootkit 改变SSDT中的值因此调用的预期函数会被替换成 rootkit的 代码
- 0x25 被修改为恶意驱动驱动的函数
- 但挂钩NtCreateFile 并不能隐藏文件





# Rootkit 分析实践

- 检测SSDT挂钩技术的最直接的方式
  - 查看SSDT
  - 查找不合理的值
  - 在当前情况下ntoskrnl.exe 起始地址为804d7000 ， 结束地址为806cd580
  - ntoskrnl.exe 是核心！
- `!m m nt`
  - 列出匹配“nt”的模块（内核模块）
  - 显示SSDT表



# SSDT 表

*Example 11-13. A sample SSDT table with one entry overwritten by a rootkit*

```
kd> lm m nt
```

```
...
```

```
8050122c 805c9928 805c98d8 8060aea6 805aa334
8050123c 8060a4be 8059cbbc 805a4786 805eb406
8050124c 804feed0 8060b5c4 8056ae64 805343f2
8050125c 80603b90 805b09c0 805e9624 80618a56
8050126c 805edb86 80598e34 80618caa 805986e6
8050127c 805401f0 80636c9c 805b28bc 80603be0
8050128c 8060be48 1f7ad94a4 8056bc5c 805ca3ca
8050129c 805ca102 80618e86 8056d4d8 8060c240
805012ac 8056d404 8059fba6 80599202 805c5f8e
```

- 标记入口点，识别挂钩
- 通过查看一个干净的系统SSDT，来识别它



# 查找恶意驱动

- `lm`
  - 列举打开模块
  - 在内核中，它们都是驱动

*Example 11-14. Using the `lm` command to find which driver contains a particular address*

```
kd>lm
```

```
...
```

```
f7ac7000 f7ac8580 intelide (deferred)
```

```
f7ac9000 f7aca700 dmload (deferred)
```

```
f7ad9000 f7ada680 Rootkit (deferred)
```

```
f7aed000 f7aee280 vmmouse (deferred)
```

```
...
```



### *Example 11-16. Listing of the rootkit hook function*

```
000104A4  mov     edi, edi
000104A6  push    ebp
000104A7  mov     ebp, esp
000104A9  push    [ebp+arg_8]
000104AC  call    1sub_10486
000104B1  test    eax, eax
000104B3  jz      short loc_104BB
000104B5  pop     ebp
000104B6  jmp     NtCreateFile
000104BB  -----
000104BB                ; CODE XREF: sub_104A4+F j
000104BB  mov     eax, 0C0000034h
000104C0  pop     ebp
000104C1  retn    2Ch
```

The hook function jumps to the original `NtCreateFile` function for some requests and returns to `0xC0000034` for others. The value `0xC0000034` corresponds to `STATUS_OBJECT_NAME_NOT_FOUND`. The call at **1** contains



# 中断

- 中断允许硬件触发软件事件
- 驱动调用IoConnectInterrupt为中断注册处理程序
- 指定中断服务例程 (ISR)
  - 当触发中断时，该例程会被调用
- 中断描述表 (IDT)
  - 存储ISR 信息
  - !idt 命令显示IDT

### *Example 11-17. A sample IDT*

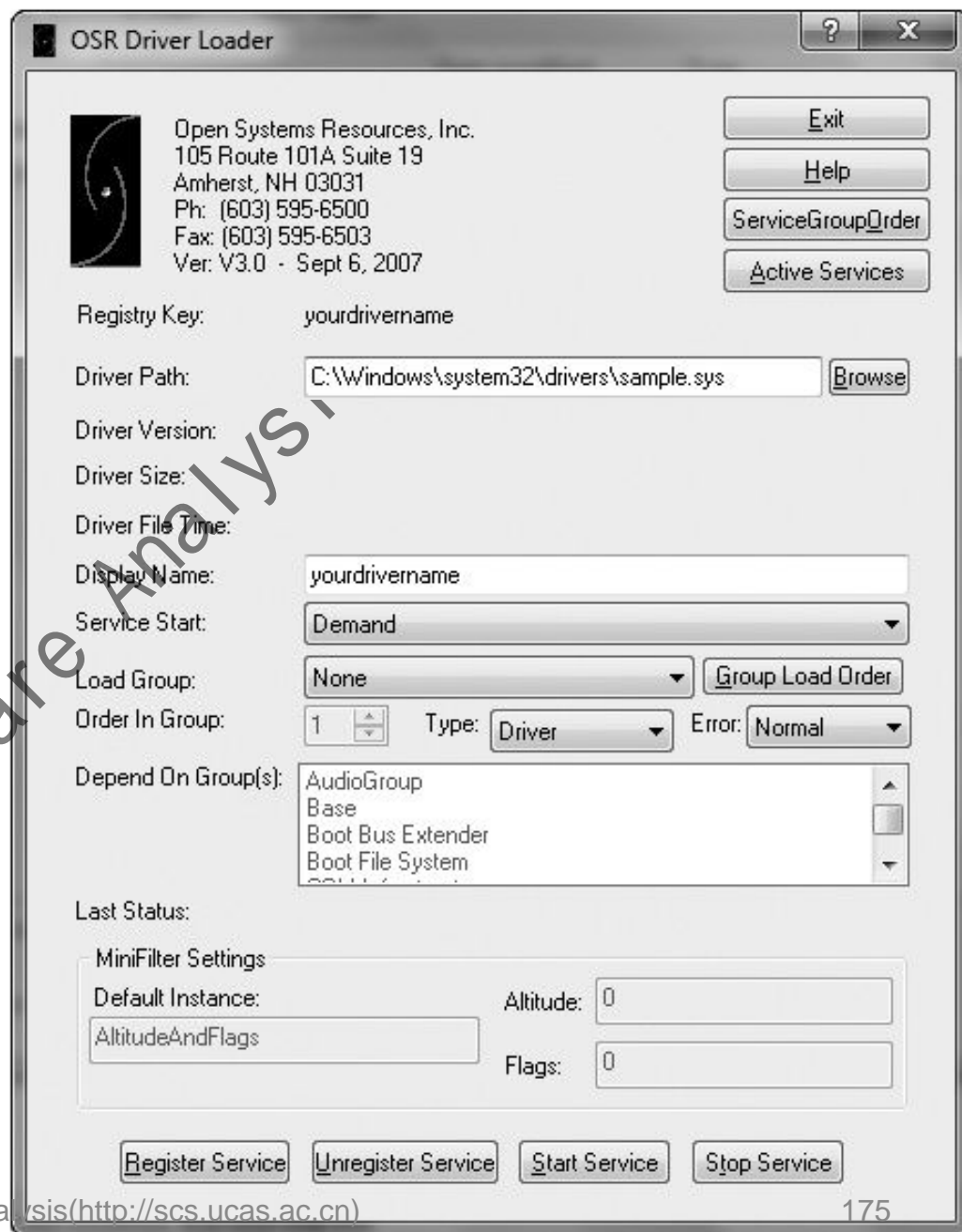
kd> !idt

```
37: 806cf728 hal!PicSpuriousService37
3d: 806d0b70 hal!HalpApcInterrupt
41: 806d09cc hal!HalpDispatchInterrupt
50: 806cf800 hal!HalpApicRebootService
62: 8298b7e4 atapi!IdePortInterrupt (KINTERRUPT 8298b7a8)
63: 826ef044 NDIS!ndisMIsr (KINTERRUPT 826ef008)
73: 826b9044 portcls!CKsShellRequestor::`vector deleting destructor'+0x26
    (KINTERRUPT 826b9008)
    USBPORT!USBPORT_InterruptService (KINTERRUPT 826df008)
82: 82970dd4 atapi!IdePortInterrupt (KINTERRUPT 82970d98)
83: 829e8044 SCSI!ScsiPortInterrupt (KINTERRUPT 829e8008)
93: 826c315c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 826c3120)
a3: 826c2044 i8042prt!I8042MouseInterruptService (KINTERRUPT 826c2008)
b1: 829e5434 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 829e53f8)
b2: 826f115c serial!SerialCISrSw (KINTERRUPT 826f1120)
c1: 806cf984 hal!HalpBroadcastCallService
d1: 806ced34 hal!HalpClockInterrupt
e1: 806cff0c hal!HalpIpiHandler
e3: 806cfc70 hal!HalpLocalApicErrorService
fd: 806d0464 hal!HalpProfileInterrupt
fe: 806d0604 hal!HalpPerfInterrupt
```

Interrupts going to unnamed, unsigned, or suspicious drivers could indicate a rootkit or other malicious software.

# 加载驱动

- 如果你想加载一个驱动来测试，你可以下载 OSR Driver Loader 工具



# Windows Vista、Windows 7和 x64 版本的内核问题



- 使用BCDedit 代替 boot.ini
- x64 版本从 XP开始有PatchGuard
  - 防止第三方代码修改内核
  - 包括内核代码自身、SSDT、 IDT等
  - 能够感染调试过程，因为调试器在插入断点时会修改代码
- 这是64位内核调试工具





# 驱动签名

- 从64位版的Vista开始强制执行驱动签名
- 只加载数字签名的驱动
- 有效防护！
- 64位系统上的内核恶意代码实际上还不存在
  - 可以通过在BCDEdit中指定nointegritychecks关闭驱动签名功能