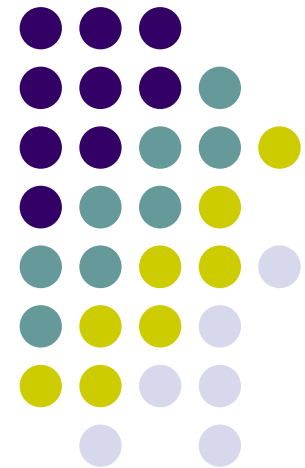


ROOTKIT





Rootkit

看不到的一定不存在吗？

- 恶意程序通常会在系统留下痕迹。
 - 文件
 - 进程
 - 端口号
 - 注册表启动键值
 -

● 看不到就一定不存在吗？

- RootKit
- IceSword





ROOTKIT

- 什么是 Rootkit [此处只讨论基于Windows平台的]
 - Rootkit与普通木马后门以及病毒的区别
- Rootkit宗旨：隐蔽
 - 通信隐蔽、自启动项隐藏、文件隐藏、进程/模块隐藏、
 - 注册表隐藏、服务隐藏、端口隐藏 etc.
- 研究内核级后门 Rootkit 技术的必要性
 - 事物两面性；信息战、情报战





操作系统原理

- 内核（Kernel）
- 外壳（Shell）
- 运行级别（Ring）
 - 内核运行于Ring 0级别
 - 外壳拥有Ring 3级别





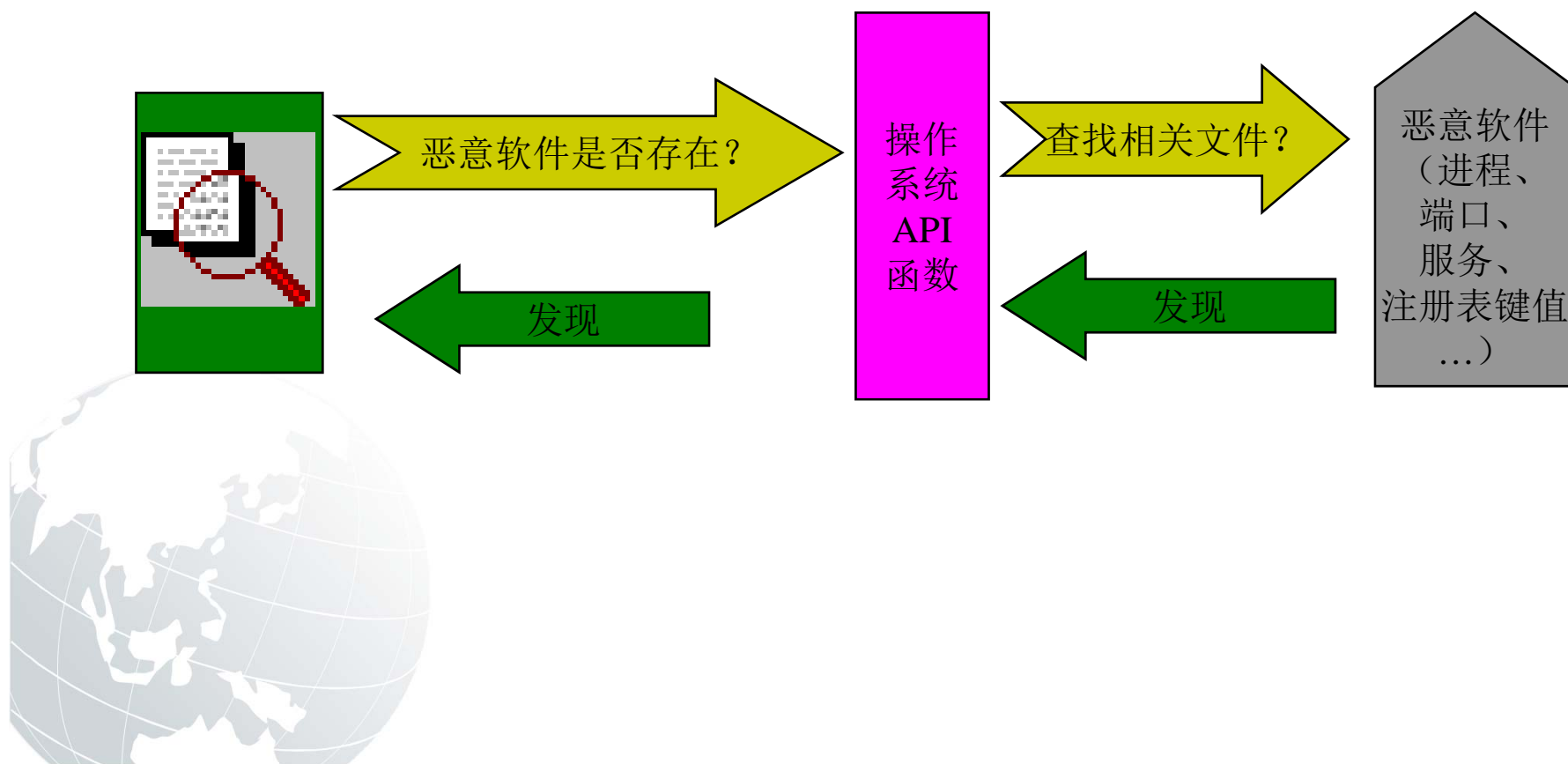
WINDOWS

- 用户态API（RING3）
 - Win32 API和POSIX接口API
- Native API（内核）
 - Windows NT架构系统中真正工作的API
- POSIX标准（可移植操作系统接口，Portable Operating System Interface）
- 子系统”（Sub System）
 - win32

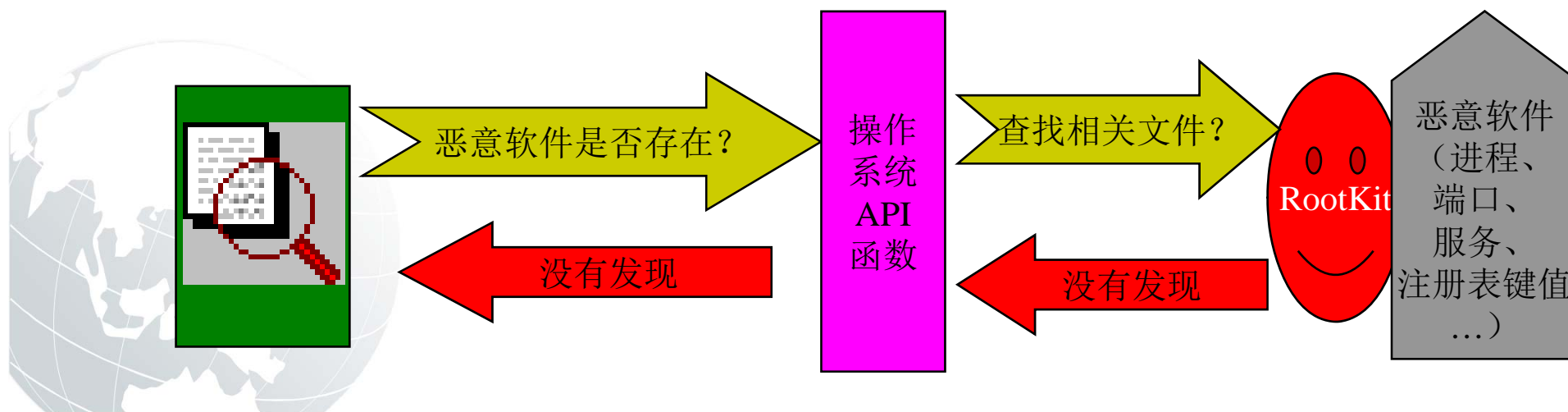
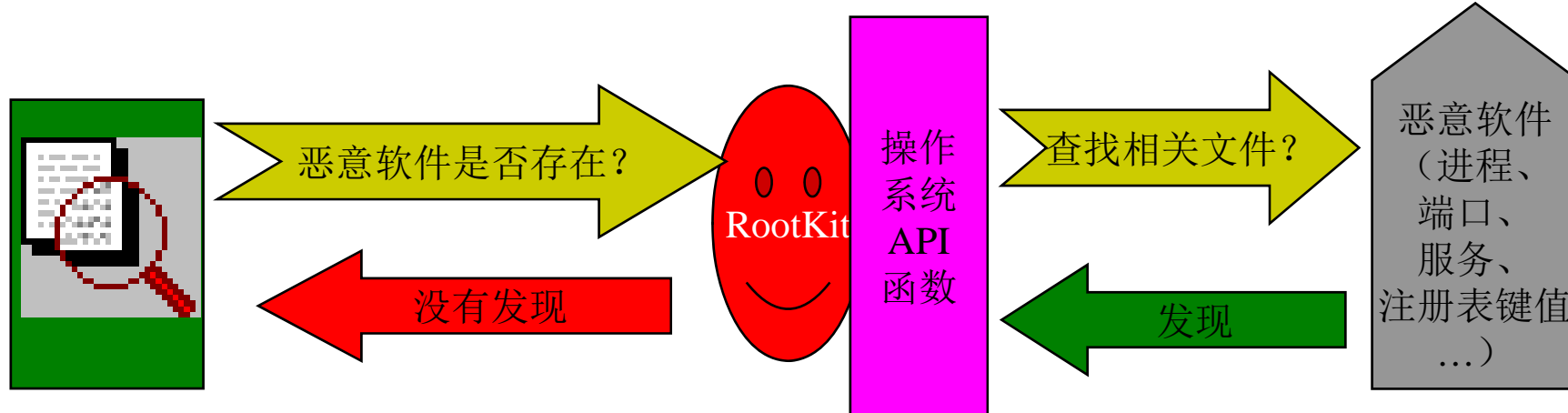




正常的系统查询过程



RootKit入侵之后的系统查询过程

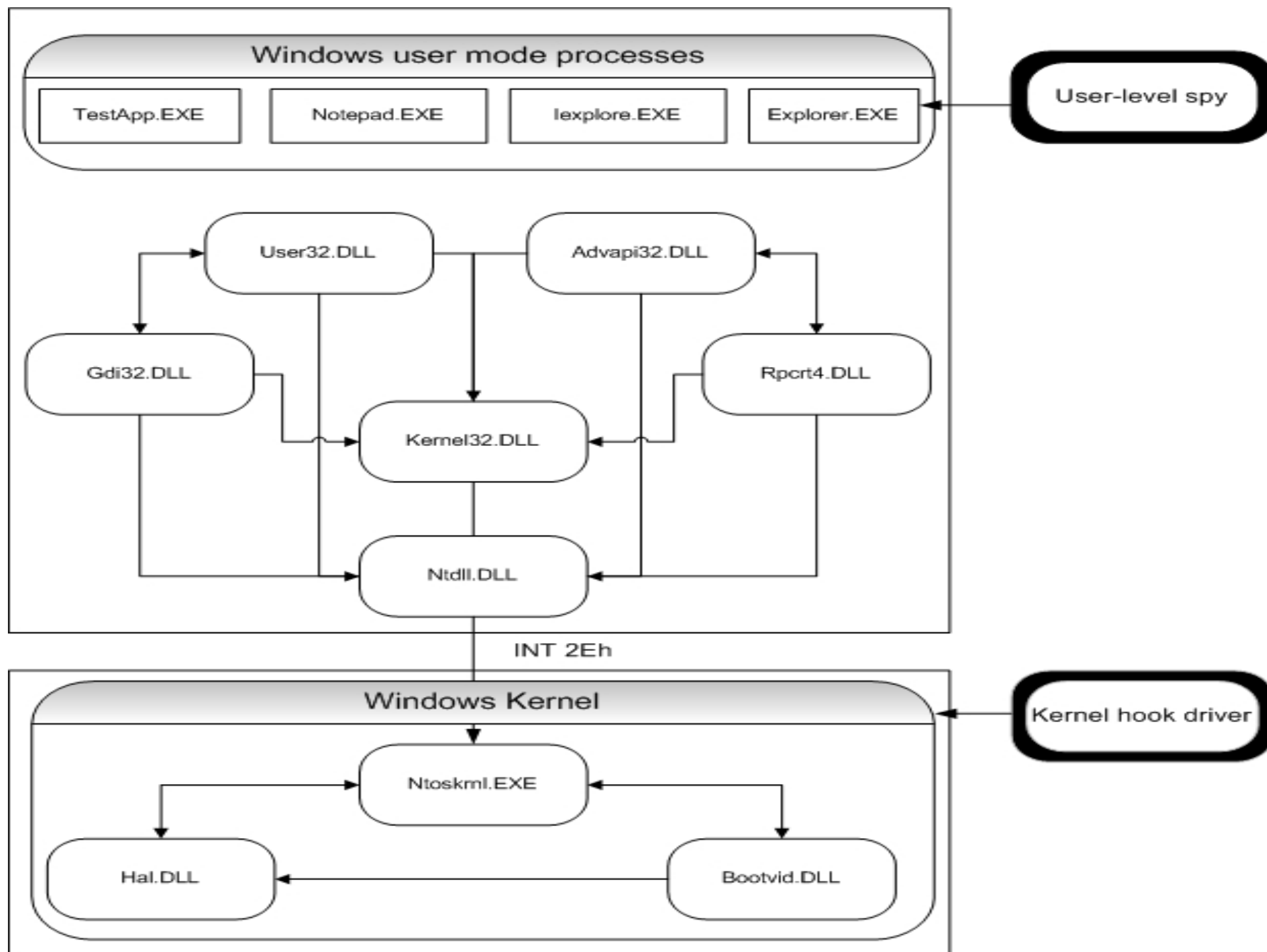


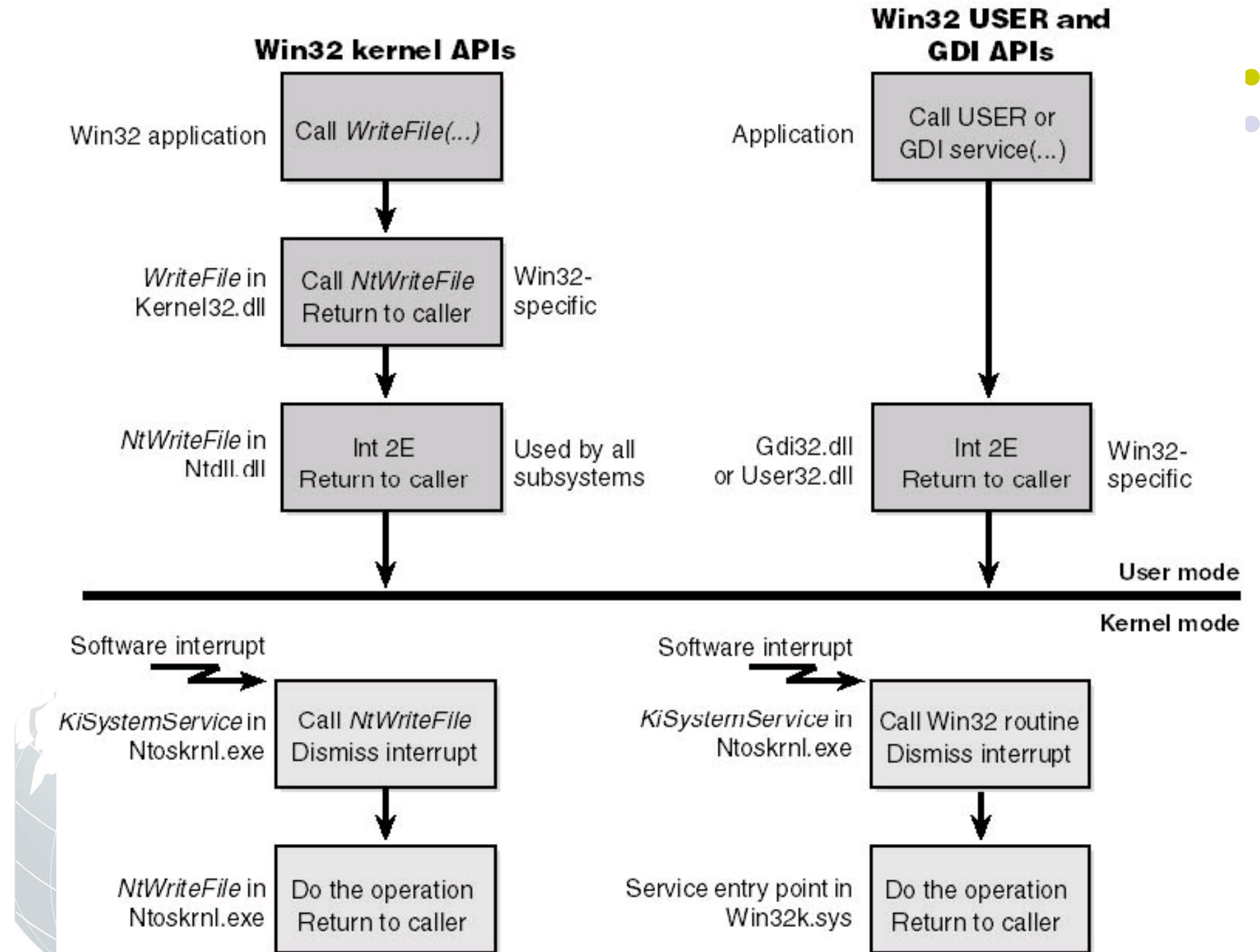


按照内核模式分类

- 用户级Rootkit
 - 不深入系统内核，通常在用户层进行相关操作。
- 内核级Rootkit
 - 深入系统内核，改变系统内核数据结构，控制内核本身。









Rootkit技术发展

- 1. Ring3 (用户态) -> Ring0 (核心态)
- 2. MEP (Modify Execution Path) -> DKOM (Direct Kernel Object Manipulation)
- 3. 越来越深入系统底层,挖掘未公开系统内部数据结构
- 4. 非纯技术性的各种新思路..





Windows用户模式Rootkit

- 像UNIX上一样，修改关键性操作系统软件以使攻击者获得访问权并隐藏在计算机中。
- 用户模式RootKit控制操作系统的可执行程序，而不是内核。



Windows下用户模式RootKit不盛行



- 原因如下：
 - 应用程序级后门迅速增加。
 - 很多Windows RootKit直接聚焦于控制内核
 - Windows文件保护（WFP）阻碍可执行程序的替换。
 - Windows源码不公开。
 - 缺乏详细文档，对Windows的内部工作原理不够了解。

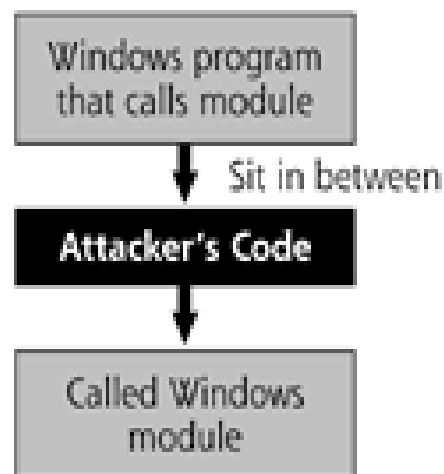


Windows RootKit的三种方法



- 使用现有接口在现有Windows函数之间注入恶意代码。
 - FakeGINA
Ctrl+Alt+Del → winlogon.exe → fakegina.dll → msgina.dll
 - 方法：添加注册表键值
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon 下添加GinaDLL变量名，类型为[REG_SZ]即可。
- 关闭Windows文件保护机制，然后覆盖硬盘上的文件。
 - 解决WFP, SFC (System File Checker)
 - 方法一，sfc /scannow
 - 方法二，gpedit.msc中修改计算机配置 → 管理模板 → 系统 → windows文件保护，设置文件保护，修改为已禁用。
 - 方法三，HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon 修改SFCDisable=dword:ffffff9d
- 利用DLL注入和API挂钩操纵正在内存中运行的进程。





Technique 1:

Use existing interfaces to insert malicious code between existing Windows functions.
Example: FakeGINA



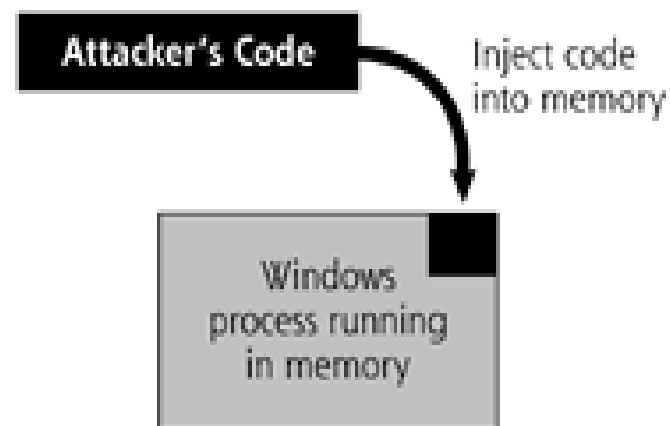
嵌入代码



Technique 2:

Disable Windows File Protection feature and overwrite files on the hard drive.
Example: Code Red II Worm

覆盖代码



Technique 3:

Utilize DLL injection and API hooking to manipulate running processes in memory
Example:
AFX Windows RootKit

DLL注入和APIhook

防御Windows 用户模式 RootKit



- 强化和修补系统，使得攻击者不能获得管理员和系统权限。
 - Win2K Pro Gold Template
 - CIS: Scoring tool
- 使用文件完整性检验工具
 - 如Fcheck, Tripwire商业版...
- 安装防病毒软件
- 安装防火墙
- 如果发现Rootkit已进入系统，最好重建系统，并小心应用补丁程序。





内核模式RootKit

- 内核模式RootKit: 修改现有的操作系统软件（内核本身），从而使攻击者获得一台计算机的访问权并潜伏在其中。
- 比用户模式RootKit更彻底、更高效。



大多数内核模式Rootkit采用以下手段



- 文件和目录隐藏
- 进程、服务、注册表隐藏
- 网络端口隐藏
- 混合模式隐藏（隐藏网络接口混合状态）
- 执行改变方向
- 设备截取和控制
 - 如底层键盘截获





Rootkit的技术思路

- 改变函数的执行路径，从而引入/执行攻击者的代码，如修改IAT, SSDT, in-line 函数 hooking。
- 增加过滤层驱动。（kernel）
- 直接修改物理内存。（Direct Kernel object manipulation, DKOM）





检测Rootkit的技术思路

- 基于签名的检测：keywords
- 启发式或行为的检测：
VICE/Patchfinder(inject code)
- 交叉检测（Cross view based detection）:RootKit
revealer/Klister/Blacklight/GhostBuster
- 完整性检测：System virginity
Verifier/Tripware.





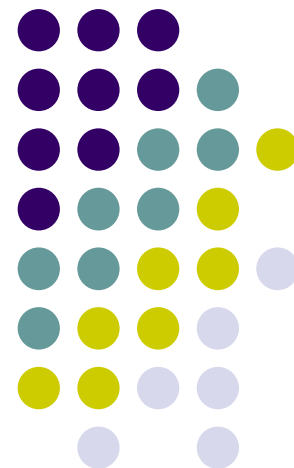
加强Windows内核防护

- 防御
 - 定期加强配置，打补丁
 - IPS（Intrusion Prevention Systems）
- 检测
 - 防病毒软件
 - 文件完整性检测工具
 - RootKit检测工具（IceSword， RootkitRevealer.zip）

- (*show*)



RootKit技术篇

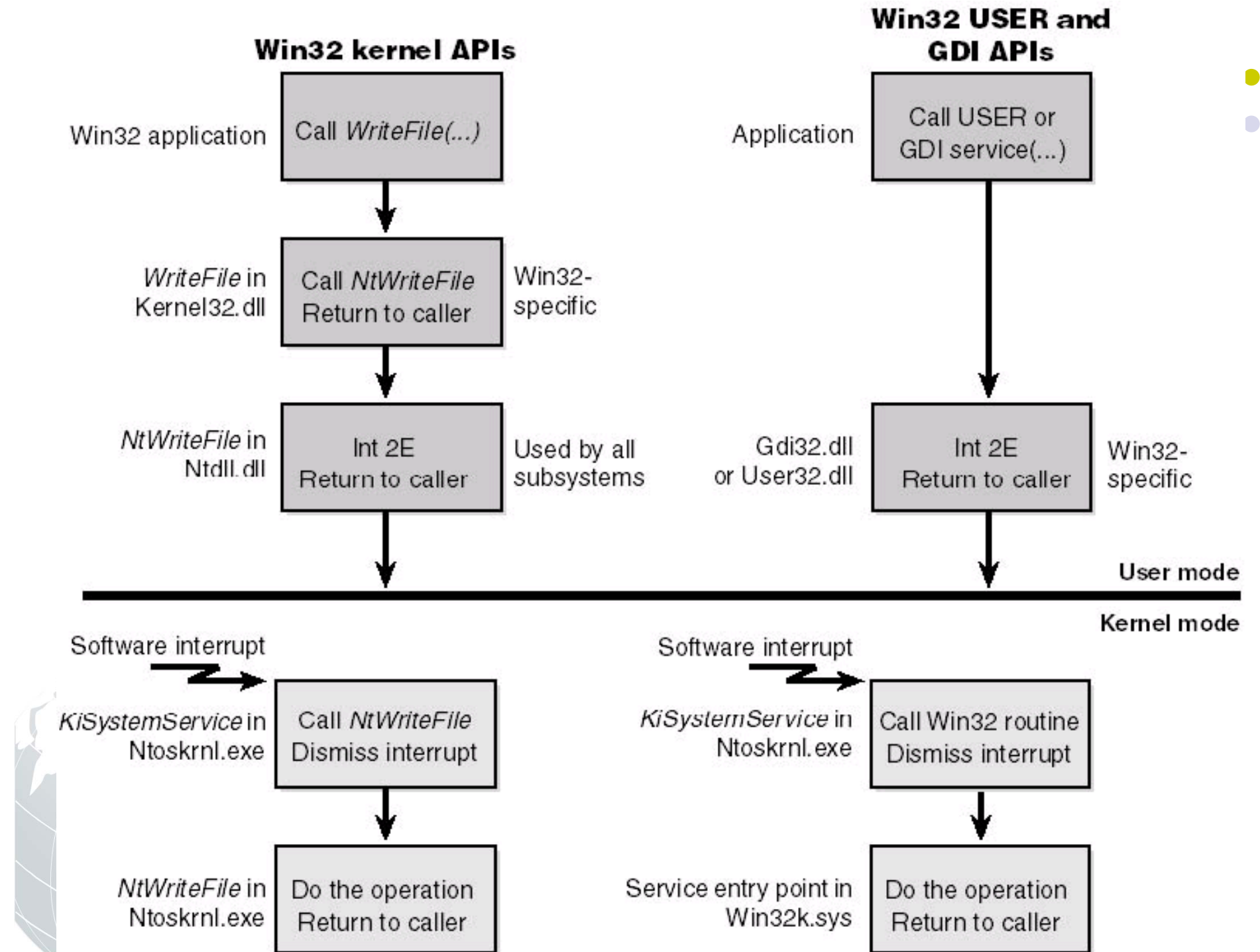




Windows系统服务调用

- Windows系统服务调用(System Call)
转发请求到内核； 用户态切换到内核态。
- 在Windows 2000中默认存在两个系统服务调度表：
- KeServiceDescriptorTable
ntoskrnl.exe 系统服务 kernel32.dll/ advapi32.dll
- KeServiceDescriptorTableShadow
USER和GDI服务 User32.dll/Gdi32.dll
- Win32内核API经过Kernel32.dll/advapi32.dll进入NTDLL.dll后使用int 0x2e中断进入内核，最后在Ntoskrnl.exe中实现了真正的函数调用； Win32 USER/GDI API直接通过User32.dll/Gdi32.dll进入了内核，最后却是在Win32k.sys中实现了真正的函数调用。







ROOTKIT

- MEP (Modify Execution Path) 行为拦截挂钩技术
- Hooks（挂钩、挂接的意思）：
- 目的：拦截系统函数或相关处理例程，先转向我们自己的函数处理，这样就可以实现过滤参数或者修改目标函数处理结果的目的，实现进程、文件、注册表、端口之类的隐藏
- Hook技术分类：
 - Inline Hook(比如修改目标函数前几个字节为jmp至我们的函数)
 - IAT (Import Address Table)
 - SSDT (System Service Descriptor Table)
 - IDT (Interrupt Descriptor Table)
 - Filter Driver (I/O Request Packet (IRP))
 - Hook IRP Function, etc...





NT进程的隐藏

- 实现进程隐藏有两种思路：
 - 第一是让系统管理员看不见（或者视而不见）你的进程；
 - 第二是不使用进程。





- 能否使用第一种方式？
- 在Windows中有多多种方法能够看到进程的存在：
 - PSAPI（Process Status API）；
 - PDH（Performance Data Helper）；
 - ToolHelp API。
- 如果我们能够欺骗用户和入侵检测软件用来查看进程的函数（例如截获相应的API调用，替换返回的数据），我们就完全能实现进程隐藏。
- 但是存在两个难题：
 - 一来我们并不知道用户和入侵软件使用的是什么方法来查看进程列表；
 - 二来如果我们有权限和技术实现这样的欺骗，我们就一定能使用其它的方法更容易的实现进程的隐藏。





- 使用第二种方式最流行。
- **DLL**是Windows系统的另一种“可执行文件”。**DLL**文件是Windows的基础，因为所有的**API**函数都是在**DLL**中实现的。**DLL**文件没有程序逻辑，是由多个功能函数构成，它并不能独立运行，一般都是由进程加载并调用的。
- 假设我们编写了一个木马**DLL**，并且通过别的进程来运行它，那么无论是入侵检测软件还是进程列表中，都只会出现那个进程而并不会出现木马**DLL**，如果那个进程是可信进程，（例如资源管理器**Explorer.exe**，没人会怀疑它是木马吧？）那么我们编写的**DLL**作为那个进程的一部分，也将成为被信赖的一员而为所欲为。





用DLL实现Rootkit功能

用DLL实现功能，然后，用其他程序启动该DLL.

- 有三种方式：
 - 最简单的方式——**RUNDLL32**
 - 特洛伊**DLL**
 - 线程插入技术





- 最简单的方式——**RUNDLL32**
 - Rundll32 DllFileName FuncName
 - Rundll32.exe MyDll.dll MyFunc





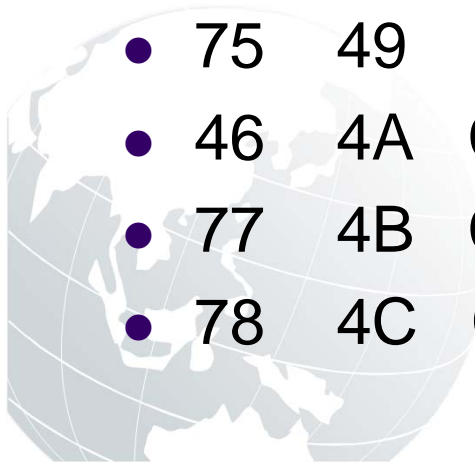
- 比较高级的方式—特洛伊**DLL**

- 特洛伊**DLL**（欺骗**DLL**）的工作原理是使用欺骗**DLL**替换常用的**DLL**文件，通过函数转发器将正常的调用转发给原**DLL**，截获并处理特定的消息。

- 函数转发器**forward**的认识。

- 是**DLL**输出段中的一个条目,用来将一个函数调用转发到另一个**DLL**中的另一个函数,
- Visual Studio 7命令提示符>dumpBin -Exports c:\windows\system32\Kernel32.dll | more

- 75 49 CloseThreadpoollo
- 46 4A CloseThreadpoolTimer
- 77 4B CloseThreadpoolWaiter
- 78 4C CloaseThreadpoolWork





函数转发器forward

- 我们也可以在在自己的程序中全用函数转发器,最简的方法是全用pragma指示符,如下面所示:
- #pragma
comment(linker, "/Export:SomeFunc=DllWork.SomeOtherFunc")
- 这个pragma告诉链接器,下在编译的DLL应该输出一个名为someFunc的函数,但实际实现somefunc的是另一个SomeOtherFunc的函数,些函数被包含在别一个名为DllWork.dll的模块中.我们必须为每个想转发的函数单独那么一行pragma.





- 特洛伊DLL的弱点：
 - system32目录下有一个dllcache的目录，这个目录中存放着大量的DLL文件，一旦操作系统发现被保护的DLL文件被篡改（数字签名技术），它就会自动从dllcache中恢复这个文件。
 - 有些方法可以绕过dllcache的保护：
 - 先更改dllcache目录中的备份再修改DLL文件
 - 利用KnownDLLs键值更改DLL的默认启动路径等
 - 同时特洛伊DLL方法本身也有一些漏洞（例如修复安装、安装补丁、升级系统、检查数字签名等方法都有可能致特洛伊DLL失效），所以这个方法也不能算是DLL木马的最优选择。





- 更高级方式——动态嵌入技术
 - DLL Rootkit的更高境界是动态嵌入技术，动态嵌入技术指的是将自己的代码嵌入正在运行的进程中的技术。多种嵌入方式：窗口Hook、挂接API、远程线程。



代码注入技术1—创建远程线程



- 提升本进程特权级为SeDebugPrivilege，获取目标进程句柄
- 将线程中所需函数地址及字符串保存在远程参数中
- 在目标进程中为远程线程和线程参数申请内存空间
- 将线程代码和参数结构拷贝到分配的内存中
- 启动远程线程 CreateRemoteThread
- 等待远程线程退出 WaitForSingleObject
- 释放申请的空间，关闭打开的句柄





创建远程线程技术

- 远程线程技术指的是通过在另一个进程中创建远程线程的方法进入那个进程的内存地址空间。
- 通过**CreateRemoteThread**也同样可以在另一个进程内创建新线程，新线程同样可以共享远程进程的地址空间。





- HANDLE CreateRemoteThread(

- HANDLE hProcess,
- PSECURITY_ATTRIBUTES psa,
- DWORD dwStackSize,
- PTHREAD_START_ROUTINE pfnStartAddr,
- PVOID pvParam,
- DWORD fdwCreate,
- PDWORD pdwThreadId);

一个地址





- `DWORD WINAPI ThreadFunc(PVOID pvParam);`
- `HINSTANCE LoadLibrary(PCTSTR pszLibFile);`

- 两个函数非常类似






- 需解决的问题：
 - 第一个问题，获取LoadLibrary的实际地址。
 - PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryA");
 - 第二个问题，把D L L路径名字符串放入宿主进程。使用：
 - VirtualAllocEx, VirtualFreeEx , ReadProcessMemory , WriteProcessMemory 等函数。



```
const DWORD THREADSIZE=1024*4;
HANDLE pRemoteThread,hRemoteProcess;
PTHREAD_START_ROUTINE pfnAddr = NULL;
DWORD pld = 0;
void *pFileRemote = NULL;
HWND hWinPro=::FindWindow ("ProgMan",
NULL); //取窗体句柄
if(!hWinPro)
{
    return 0;
}
else
{
```





```
::GetWindowThreadProcessId(hWinPro,&pId); //获取目标句柄的PID  
hRemoteProcess=::OpenProcess(PROCESS_ALL_ACCESS,false,pId); //打开进程  
pFileRemote=::VirtualAllocEx(hRemoteProcess,0,THREADSIZE,MEM_COMMIT|MEM_RESERVE,PAGE_EXECUTE_READWRITE);//分配内存空间  
if(!::WriteProcessMemory(hRemoteProcess,pFileRemote,"d://RemoteDll.dll",THREADSIZE,NULL))  
    return;  
pfnAddr=(PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryA"); //获取API地址  
pRemoteThread=::CreateRemoteThread(hRemoteProcess,NULL,0,pfnAddr,pFileRemote,0,NULL);//注入线程  
if(pRemoteThread==NULL)  
    return;  
else MessageBox("success!"); //注入成功  
}  
}
```



- 操作步骤做一个归纳:
- 1) 使用VirtualAllocEx函数，分配远程进程的地址空间中的内存。
- 2) 使用WriteProcessMemory函数，将DLL的路径名拷贝到第一个步骤中已经分配的内存中。
- 3) 使用GetProcAddress函数，获取LoadLibraryA或LoadLibraryW函数的实地址（在Kernel32.dll中）。
- 4) 使用CreateRemoteThread函数，在远程进程中创建一个线程，它调用正确的LoadLibrary函数，为它传递第一个步骤中分配的内存的地址。





- 5) 使用VirtualFreeEx函数，释放第一个步骤中分配的内存。
- 6) 使用GetProcAddress函数，获得FreeLibrary函数的实地址（在Kernel32.dll中）。
- 7) 使用CreateRemoteThread函数，在远程进程中创建一个线程，它调用FreeLibrary函数，传递远程DLL的HINSTANCE。





代码注入技术2—插入DLL

- 利用注册表
HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_DLLs
- 使用系统范围的Windows钩子
SetWindowsHookEx
- 利用远程线程 DWORD HMODULE
- 特洛伊DLL





代码注入技术3—地址跳转

- 操作线程上下文

选择并挂起目标进程中的一个线程；

将要执行的代码注入目标进程的内存中，将该线程将执行的下一个指令的地址设置为注入的代码，然后恢复该线程的运行；

在注入代码的末尾安排跳转，作该线程原本该继续作的事。

- 在新进程中插入代码 **CreateProcess**



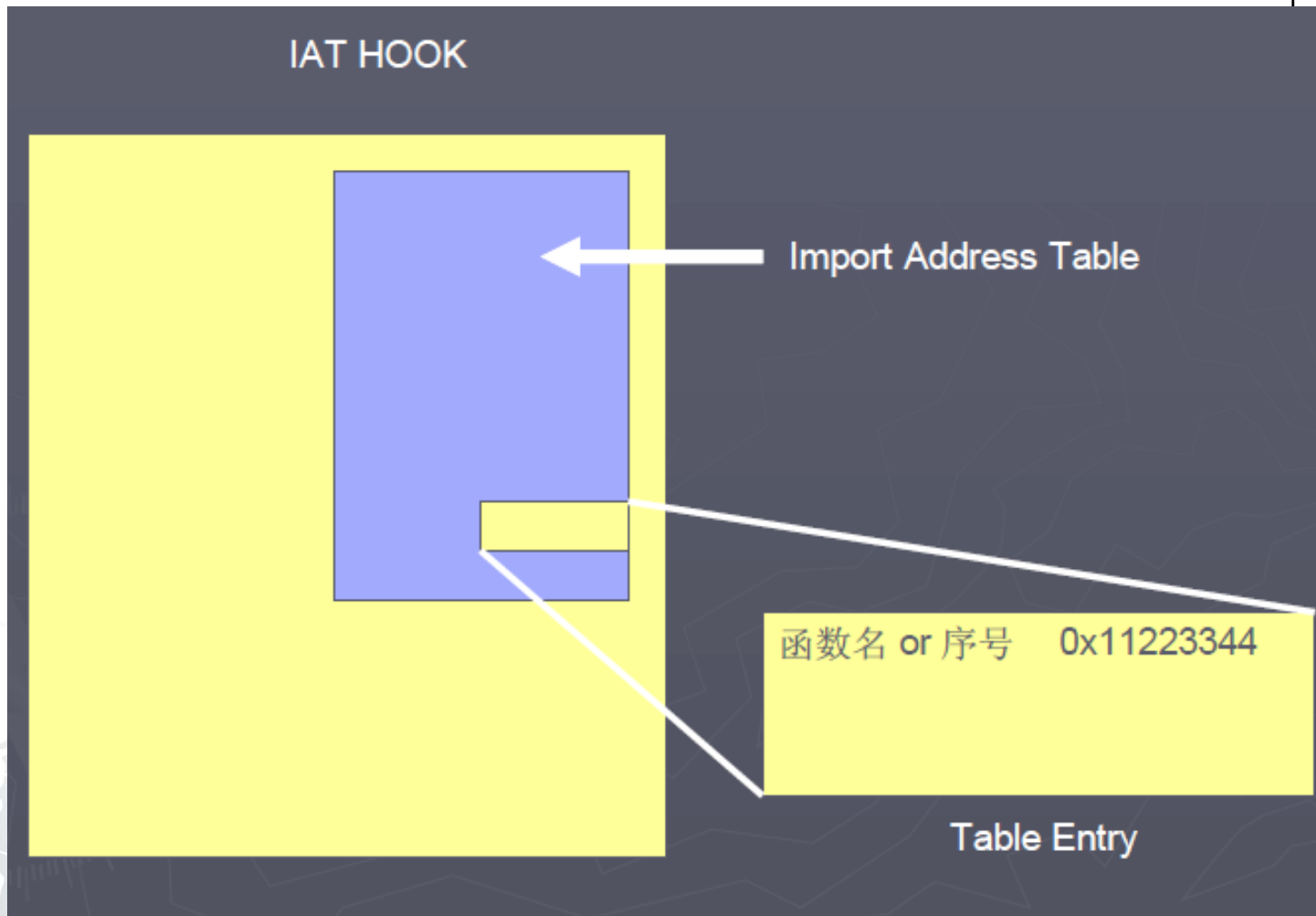


- <http://www.codeproject.com/Articles/4610/Three-Ways-to-Inject-Your-Code-into-Another-Process>
- <http://www.windbg.info/>





代码拦截技术—重定向IAT表

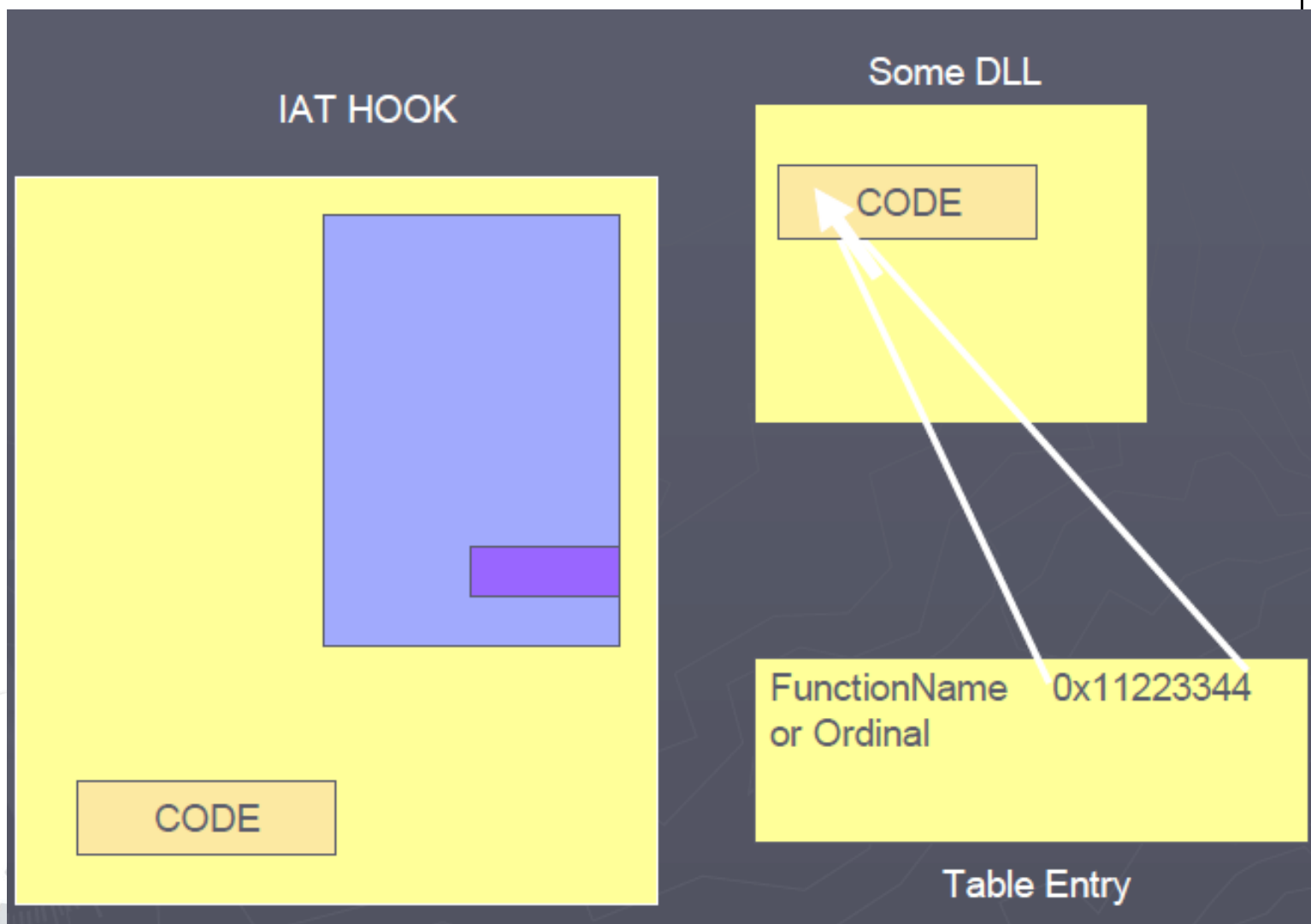


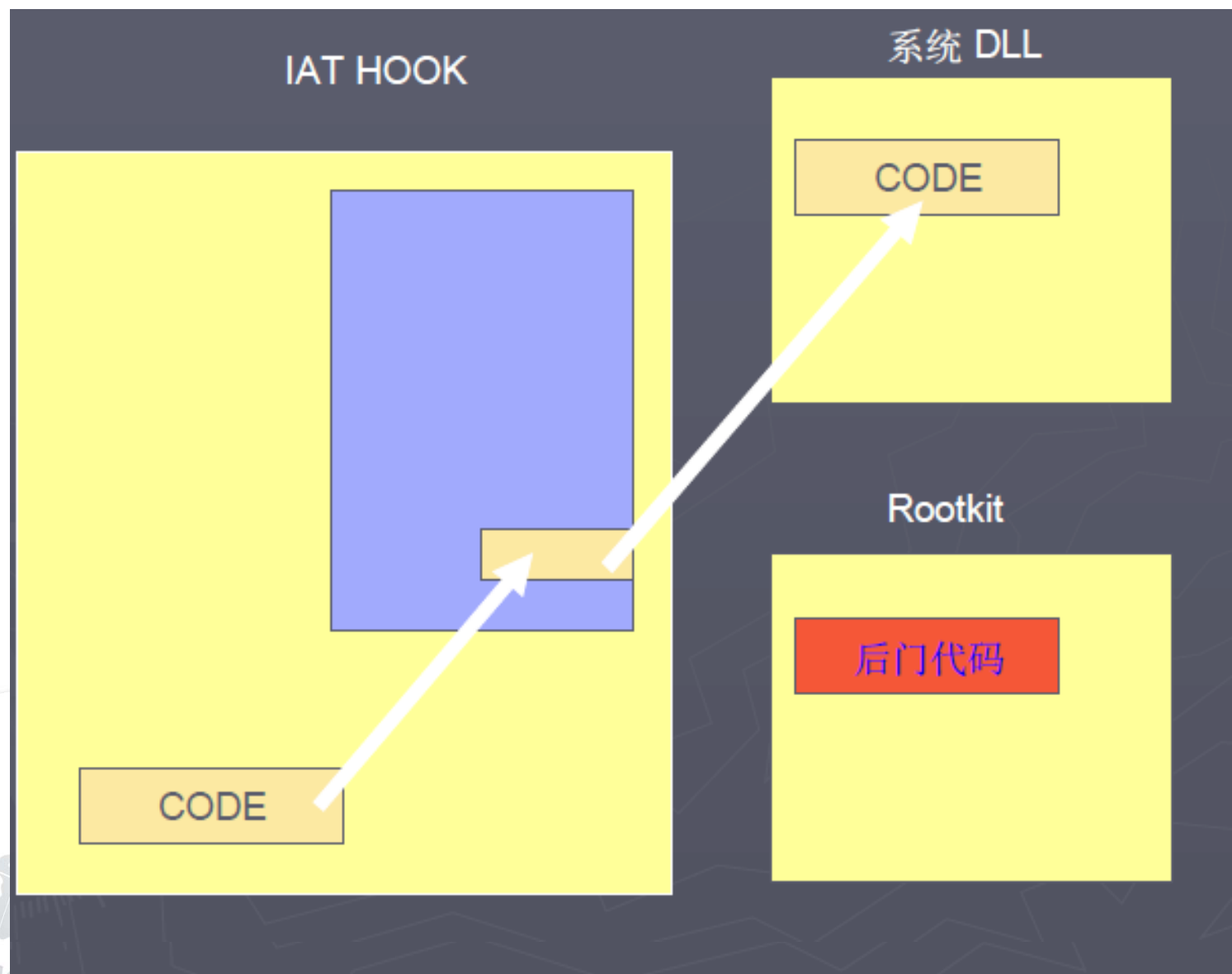


代码拦截技术—重定向IAT表

+-----+	- offset 0
MS DOS标志和DOS块	
+-----+	
PE 标志 ("PE")	
+-----+	
.text	- 代码
+-----+	
.data	- 已初始化的(全局静态)数据
+-----+	
.idata	- 导入函数的信息和数据
+-----+	Import Address Table
.edata	- 导出函数的信息和数据
+-----+	
调试符号	
+-----+	









代码拦截技术4—无条件跳转

- 获取目标函数的地址
- 将页保护属性改为
PAGE_EXECUTE_READWRITE
- 在目标函数地址写入5个字节的跳转指令，**jmp**
跳转地址
- 恢复页保护属性





内核态的代码拦截1——SSDT钩子

- SSDT（System Service Descriptor Table，系统服务描述符表）内核调用表
- ntdll.dll
 - 从用户层跳转到内核层的接口
- ntoskrnl.exe
 - NT系统真正内核程序
- 例：CreateProcess-> NtCreateProcess-> int 2Eh（Sysenter）



内核态的代码拦截1——SSDT钩子



- 系统服务的用户模式接口 NTDLL.DLL中有说明
- Win32 API函数
检查参数 转换为Unicode 调用NTDLL
- NTDLL中的函数用所请求的系统服务的ID填写EAX，用指向参数栈的指针填写EDX，并发送INT 2E指令，切换到内核态，参数从用户栈拷贝到内核栈。
- NTOSKRNL初始化时创建了系统服务分配表(SSDT)，每一项包含一个服务函数的地址。被调用时用EAX寄存器中保存的服务ID查询这个表，并调用相应的服务。
- Hook系统服务:查询系统服务分配表，修改函数指针，使之指向开发者的其他函数。





SSDT钩子

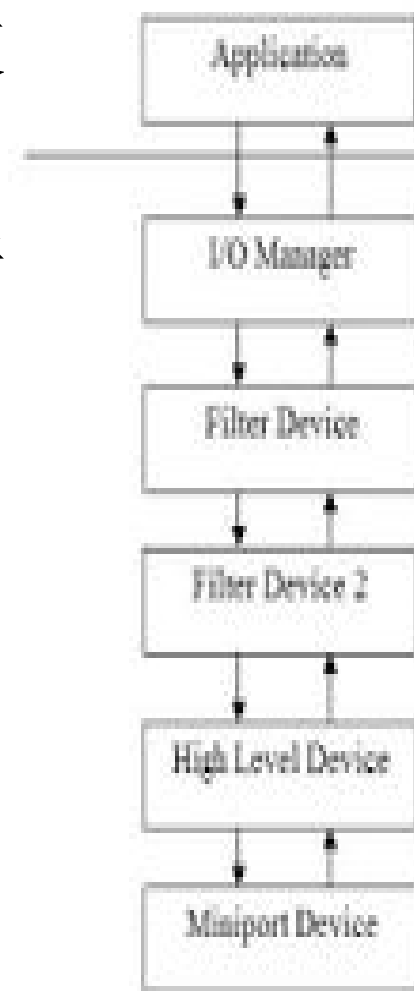
- 把SSDT里对于获取进程标识的服务号对应的原生API地址修改为指向自己位于Ring0层的驱动入口



内核态的代码拦截2——过滤驱动



- 拦截的层次越低，越不容易被发现，越不通用
- 拦截磁盘操作需要操作各种文件系统
- 挂钩文件系统驱动的派遣程序
MajorFunction IRP_MJ_XXX
- 设置过滤器，修改
KeServiceDescriptorTable，来
挂钩系统服务，如filemon。





Example--注册表监控

ZwOpenKey
ZwQueryKey
ZwQueryValueKey
ZwEnumerateValueKey
ZwEnumerateKey
ZwClose
ZwDeleteKey
ZwSetValueKey
ZwCreateKey
ZwDeleteValueKey

```
NTSTATUS (*OldZwOpenKey)
( OUT PHANDLE,
  IN ACCESS_MASK,
  IN POBJECT_ATTRIBUTES );

NTSTATUS MyZwOpenKey(
  OUT PHANDLE hKey, IN
    ACCESS_MASK Access,
  IN POBJECT_ATTRIBUTES OA )
{ ntstatus =
  OldZwOpenKey(hKey, Access,
    OA);
  ...
  return ntstatus;}
```





FSD Hook

- 文件系统（File System, FS)
- windows系列操作系统是采用IOS（Input/Output Supervisor, 输入输出管理程序）
- “可安装文件系统”（Installable File System, IFS）
- “FSD”（File System Driver, 文件系统驱动）
- FSD Filter Driver（文件系统驱动过滤器）





FSD Inline Hook

- 直接将操作系统厂商编写的相关功能使用自己的函数去取代了



不足



- 频繁地拦截系统操作，会使系统性能有所下降
- 与一些采用实时监控的软件可能不能共存，否则可能导致系统崩溃
- 隐藏无法彻底





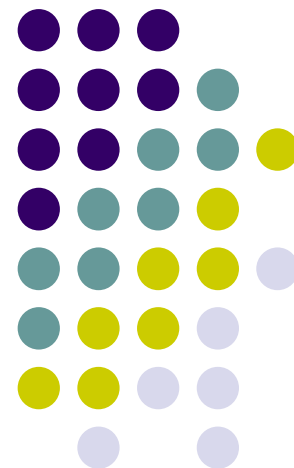
其他tools

- SSM
- IceSword
- 微点
- Unhooker
-

www.rootkit.com



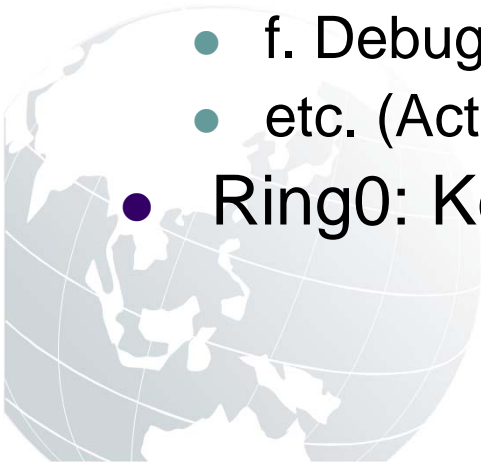
具体方法列举





代码注入

- Ring3:
 - a. CreateRemoteThread + WriteProcessMemory
 - 1. 线程注入 2. 代码注入
 - b. SetWindowsHookEx
 - c. HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\ApplInit_DLLs
 - d. Winlogon通知包
 - e. 感染PE文件（1、全部插入 2、感染IAT）
 - f. DebugActiveProcess + SetThreadContext
 - etc. (Activx, SPI, BHO...)
- Ring0: KeAttachProcess...





非常规进Ring0

- 1. 通过中断门/任务门/调用门/内存映射等技巧(只适用Wn9x,比如CIH)
- 2. \Device\PhysicalMemory对象
- 3. SetSystemInformation函数中SystemLoadAndCallImage参数,加载驱动
- 4. 感染HAL.DLL或者Win32k.sys等文件,添加调用门
- 5. 常规调用操作Windows服务的函数加载驱动(因常规而不隐蔽)
- 6. 直接调用本机函数NtLoadDriver加载驱动





网络通信隐藏

- 总之越底层越隐蔽,穿透防火墙的几率就越高
- A.
 - 1. 代码注入到防火墙默认允许访问网络的系统进程(如IE)
 - 2. Hook Socket API 或者 SPI技术 或基于TDI等实现端口复用
 - 3. TDI层面通信
 - 4. 在NDIS层面上通信... (pt,mp...) [难点: 自己实现的细节多, 自己写TCP/IP协议栈, 当然也效果最好, 能穿透软件防火墙)
- B .http隧道; 伪装为DNS协议包。为了穿透边界防火墙...





进程隐藏

- 1. 代码注入（DLL注入，线程注入，进程注入...），实现无进程
- 2. 挂钩应用层上的Process32First、Process32First等函数
- 3. 挂钩系统服务NtQuerySystemInformation
- 4. 从进程控制块中的活动进程链表（ActiveProcessLinks）中摘除自身
- 5. 从csrss.exe进程中的句柄表中摘除自身
- 6. 挂钩SwapContext，自己实现线程调度
- 7. 从PspCidTable表中摘除自身
- etc...





文件隐藏

- 1. 采用病毒技术，感染寄生于其他文件,实现无文件
- 2. 挂钩应用层上的FindFirstFile、FindNextFirst等函数
- 3. 挂钩内核态中系统服务ZwQueryDirectoryFile
- 4. 文件过滤驱动
- 5. 修改 FSD IRP Fuction 函数地址，再对相关IRP处理...
- 6. Inline Hook FSD
- etc...





类似地

- 具体实现 之 注册表隐藏...
- 具体实现 之 服务隐藏...
- 具体实现 之 模块隐藏...
- 具体实现 之 端口隐藏...
- ...





RK技术新挑战

- 突破主动防御以及进程行为监控(绕过注册表监控、代码注入监控、驱动加载监控等)
 - 加壳脱壳与加密解密
 - 加花指令与程序入口点修改
 - 内存、文件特征码的定位与修改
 - 文件植入与捆绑



遇到特征码定位在**jmp**指令上面的 构造替换 **push xxxxx**
ret。

举例: **jmp xxxxx**
构造替换 **push xxxxx**
ret

2. 遇到特征码定位在**call**指令上的。

举例:

call xxxxx
构造替换: **push @@**
jmp xxxxx
@@:

; @@的标号表示的是你**jmp xxxxx**指令后面的内存地址。 **@f**
也就是引用@@ 的标号，所以此时**@f**这里填写的就是**jmp**
xxxxxx指令后面的内存地址。。

3. 遇到特征码定位在**ret**上

举例: **ret**
构造替换:
jmp dword ptr [esp]





4. 遇到特征码定位在 **test eax, eax je xxxx or eax, eax,**
je xxxxx cmp eax, 0 jexxxxxx

举例: **test eax, eax**

je xxxxxx

构造替换: **xchg eax, ecx**

jecxz xxxxx

5. 遇到特征码定位在 **push [xxxxxx]**上的。

举例: **push [xxxxxx]**

构造:

在其之前通过 **xchg [xxxxxx], ebx**

然后用寄存器传参: **push ebx**

最后在下面在通过 **xchg [xxxxxx], ebx** 交换回来。



Anti-Rootkit

- ARK工具的运作原理和Rootkit大相径庭，它们也是通过驱动模块将自身投入系统内核中





参考资料

- 1. <http://www.rootkit.com>
- 2. 《Subverting the Windows Kernel》
- 3. RAIDE: Rootkit Analysis Identification Elimination
- 4. 《Windows防火墙与封包拦截技术》





其它程序攻击

- 邮件炸弹与垃圾邮件
 - 常用攻击工具
 - upyours4、KaBoom3、HakTek、Avalanche等
- IE攻击
 - Javascript炸弹





第6章 程序攻击

- 逻辑炸弹攻击
- 植入后门
- 病毒攻击
- 特洛伊木马攻击
- 其它程序攻击





第6章 程序攻击

● 课后习题

- 试说明逻辑炸弹与病毒有哪些相同点与不同点？
- 为什么后来的木马制造者制造出反弹式木马，反弹式木马的工作原理是什么？画出反弹式木马的工作流程图
- 嵌入式木马不同于主动型木马和反弹式木马的主要特点是什么？为什么这种木马更厉害，更不易被清除？
- 木马技术包括哪些，这些技术有什么特点？

