《N 种内核注入 DLL 的思路及实现》

Author： sudami [sudami@163.com]

Time ： 01-11-2008

Links ： http://hi.baidu.com/sudami

内核注入，技术古老但很实用。现在部分 RK 趋向无进程,玩的是 **SYS+DLL**，有的无文件，全部存在于内存中。可能有部分人会说：“都进内核了.什么不能干？”。是啊，要是内核中可以做包括 R3 上所有能做的事，软件开发商们也没必要做应用程序了。有时，我们确实需要 R3 程序去干驱动做起来很困难或者没必要驱动中去做的事，进程 / DLL 是不错的选择，但进程目标太大，所以更多的同学趋向于注 DLL。

若要开发安全软件、小型工具，可借鉴其思路，Anti Rootkits 时，在某些极端情况下，可使用同样的技术发现、清除 RK，保证用户电脑的正常使用。在此，我将探讨几种内核注入 DLL 的思路及实现原理。

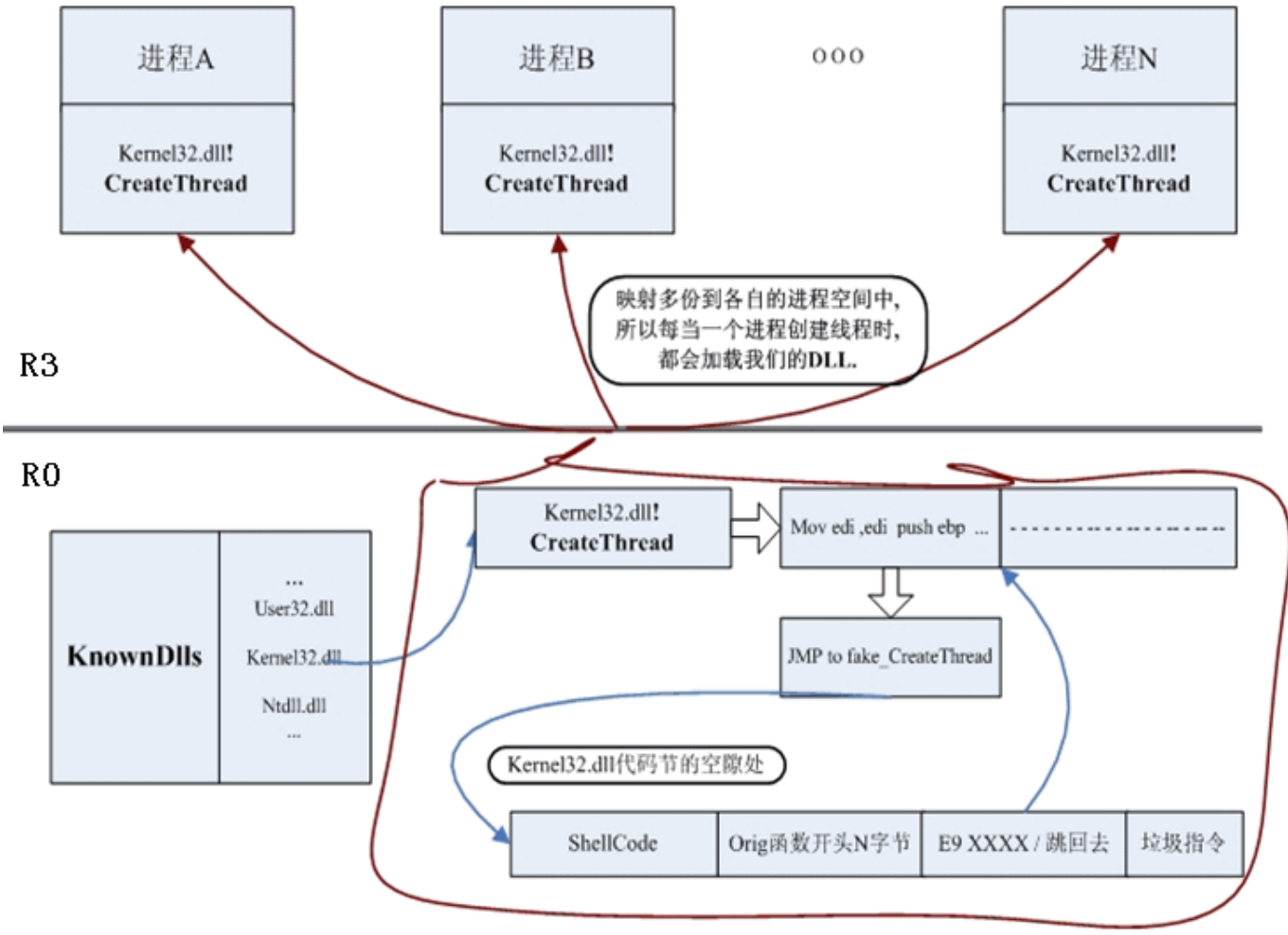
(1) APC技术

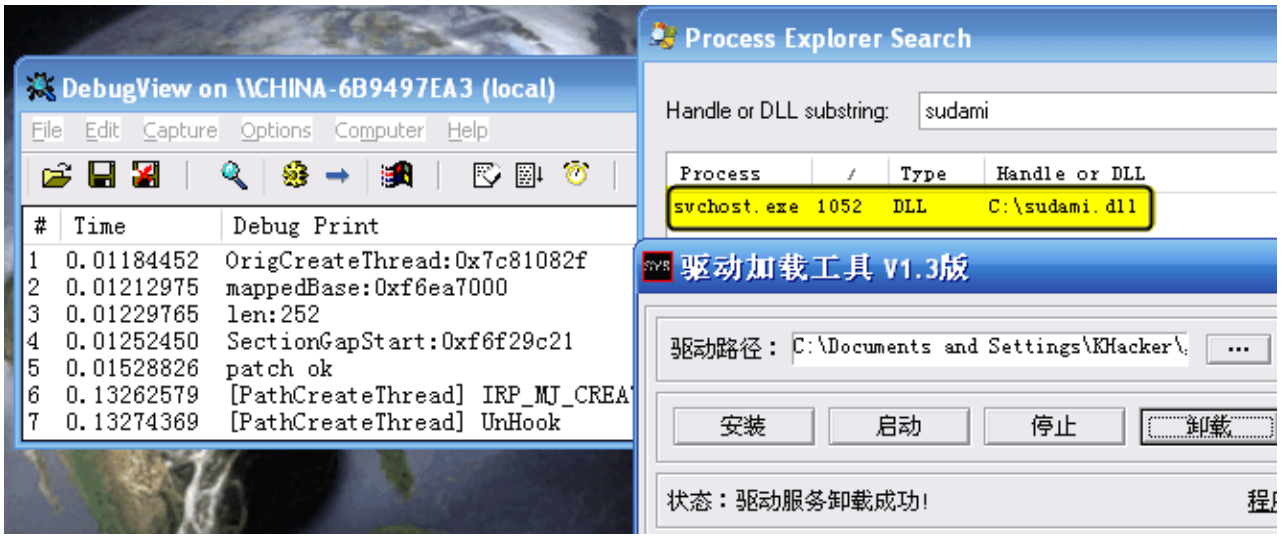
给一个 Alertbale 的用户态线程插 APC，让其执行其中的 ShellCode,来执行我们的代码。这个方法简单易行，但是不够稳定，兼容性不好。测试中发现经常出现把 Explorer.exe 等插崩溃的情况，而且有杀软在的情况下，插入有时会被拦截，起不到应有的效果。
(可参考我以前逆过的一个驱动：逆向fuck.sys--编译通过--源码)

(2) 内核Patch \\KnownDLLs\\Kernel32.dll CreateThread

\\KnownDLLs是系统加载时对象管理器加载最新磁盘DLL到内存的，当其他进程想调用某个DLL时，就不用重复从磁盘加载了，而会从这里映射一份到自己的进程空间中去。这样给我们做全局Patch提供了一个很好的机会：

ZwOpenSection打开 \\KnownDlls\\kernel32.dll，调用ZwMapViewOfSection映射一份到自己进程空间，然后寻找kernel32.dll在内存中代码节的空隙，选择这里作为我们fake函数的存储Buffer。修改CreateThread函数的开头 5 字节跳转到这个间隙，当系统任何一个线程创建时，会走到CreateThread函数，然后执行空隙中的ShellCode，其负责调用LoadLibrary加载我们的DLL。DLL一经加载，会发IOCTL通知本驱动，让驱动卸载HOOK。这样就完成了内核注DLL的过程。测试时发现**Svchost.exe**进程调用CreateThread函数很频繁，所以触发也会很快，基本 1 秒不到就能将DLL加载进去，而我们的HOOK也卸掉了。所以稳定性提高不少。示意图如下：





(3) 内核 HOOK ZwMapViewOfSection

有部分模块加载时会调用 **ZwMapViewOfSection**,比如进程创建时映射 N 份 DLL 到自己

的虚拟空间中去.我们替换 SSDT 中的这个函数,过滤出是加载 **Kernel32.dll** 的情况,从参数中取得其基址,Inline Hook 其 EAT 中的 **CreateThread** 函数,跳转到在这个进程虚拟地址空间中申请的 Buffer,在其中完成 DLL 的加载过程.

关键 API:

ZwAllocateVirtualMemory ---- 在此进程空间中分配内存,存放 Shellcode

ZwProtectVirtualMemory ---- 使当前内存块具有可读可写属性

IoAllocateMdl ---- 创建 MDL

关键 Code 如下:

```
NTSTATUS
Fake_ZwMapViewOfSection(
    IN HANDLE SectionHandle,
    ...
    )
{
    NTSTATUS status;

    status = Orig_ZwMapViewOfSection( ... );
    if( NT_SUCCESS( status ) ) {
        if( ObReferenceObjectByHandle(SectionHandle,SECTION_MAP_EXECUTE,
                    *MmSectionObjectType, KernelMode, &Section,NULL )
                    == STATUS_SUCCESS )
        { // 检查是否为image section
/*
    lkd> dt _SECTION_OBJECT
nt!_SECTION_OBJECT
   +0x014 Segment          : Ptr32 _SEGMENT_OBJECT

   lkd> dt _SEGMENT_OBJECT
nt!_SEGMENT_OBJECT
   +0x000 BaseAddress      : Ptr32 Void  -->_CONTROL_AREA

   nt!_CONTROL_AREA
   +0x020 u                : union __unnamed, 2 elements, 0x4 bytes
      +0x000 LongFlags
      +0x000 Flags         : struct _MMSECTION_FLAGS, 31 elements, 0x4 bytes
         +0x000 FailAllIo  : Bitfield Pos 4, 1 Bit
         +0x000 Image      : Bitfield Pos 5, 1 Bit  -->正是我们关注的标志位
         +0x000 Based      : Bitfield Pos 6, 1 Bit
         +0x000 File       : Bitfield Pos 7, 1 Bit
*/
```

```
        _asm
        {
        mov  edx, pSection      // _SECTION_OBJECT
        mov  eax, [edx+14h]     // _SECTION_OBJECT._SEGMENT_OBJECT
        add  eax, imageOffset
        mov  edx, [eax]         // edx = _SEGMENT_OBJECT._CONTROL_AREA
        test byte ptr [edx+20h], 20h // _SEGMENT_OBJECT._CONTROL_AREA.u

        // 加一句,因为我老是忘记:test若相等,会把标志位置1.否则置0
        jz   _sudami_over_
        mov  bISimageSection, TRUE
        mov  eax, [edx+24h]         // eax = _FILE_OBJECT
        add  eax, 30h               // _FILE_OBJECT.FileName
        mov  objectName, eax
_sudami_over_:
        nop
        }

        if( bISimageSection && IsKernel32dll(objectName) ) {
              AllocateOurMemory(); // 分配Shellcode的内存

              // 填充Shellcode,HOOk Kernel32.dll!CreateThread
              InjectOurDLL("C:\\sudami.dll") ;
        }
        ObDereferenceObject( pSection );
        }
    }
    return status;
}
```

同方法 2 相比，原理类似。但修改时机不同，效果差不多，只是注入 DLL 的时间会慢一些。至于 Shellcode 的编写,就大同小异了.萝卜白菜各有所爱，主要看个人发挥。要是闲写 shellcode 麻烦，请到看雪学院去查资料，模板很多，在这里就不 YY 了。

【看雪读书月】学习ShellCode编写

[note]一个简单的Shellcode

shellcode之小小琢磨

Add_Section

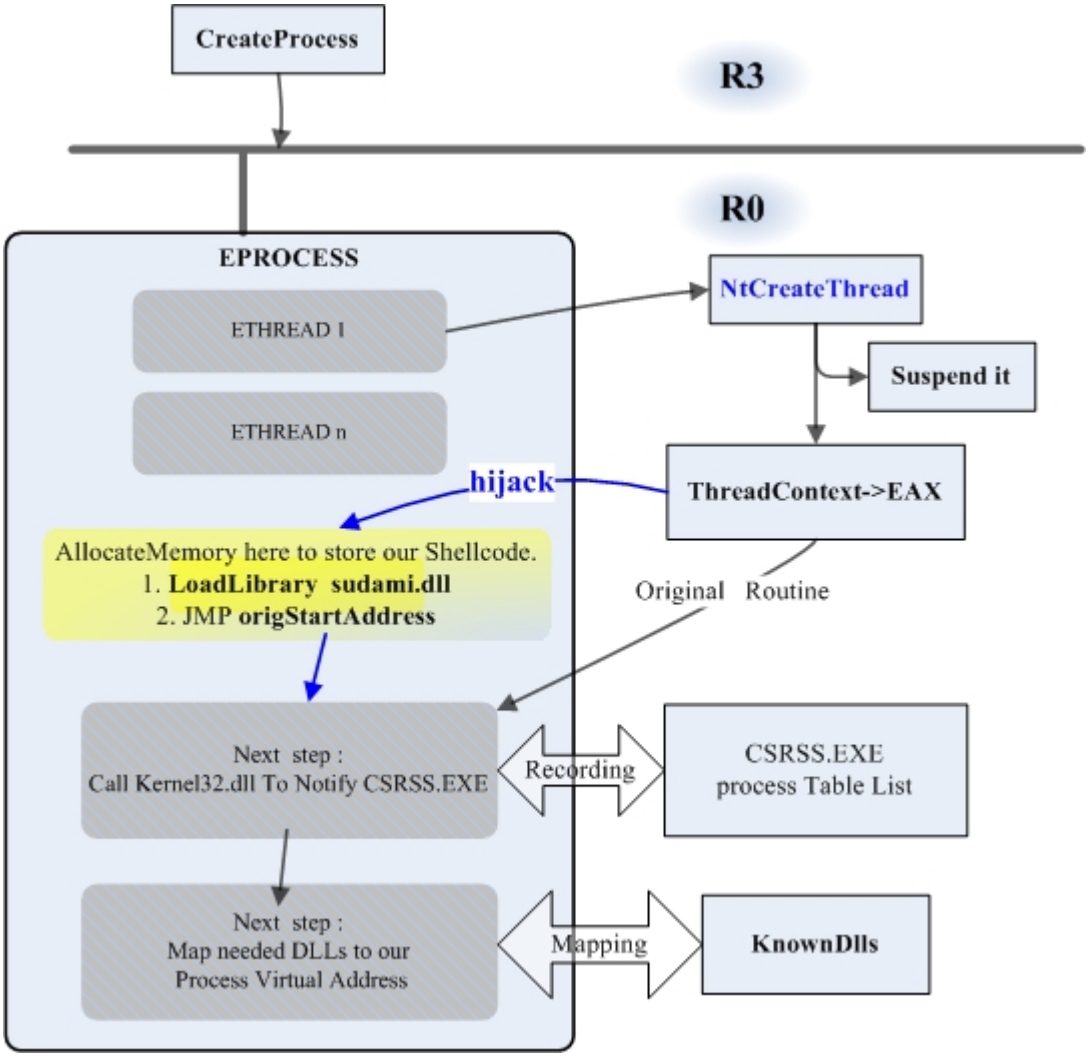
(4) 内核 HOOK NtCreateThread

跟踪**进程创建的流程**，会很明晰的发现有多点可以 patch 来实现 DLL 的注入。

进程创建完时是一个**空水壶**，里面没有沸腾的热水（threads），于是系统调用 NtCreateThread 创建其主线程（给空水壶注水 – **凉水**），在这个暂停的线程里面折腾了一阵后完事了也厌倦了，于是系统跳了出来，回到进程空间中，调用 Kernel32.dll 去通知 CSRSS.EXE，对它说：“这里有一个新进程出生了，你在你的表里标记一下”。然后就开始加载 DLL 啦，把系统 KnownDLLs 中的自己需要的 DLL 都 Map 一份到这个大水壶中。接着 KiThreadStartup 加热水壶中的凉水，于是水就开始**沸腾**了，此时主线程开始工作。。。

拦截 NtCreateThread，取得**当前线程上下文**，保存它要返回的地址（会回到空水壶中去），劫持为我们自己分配的地址，在其中填充 ShellCode 来加载目的 DLL。至于选择 Buffer，思路很多。这里可简单的 Attach 到当前进程，在充足的虚拟 2GB 进程地址空间中分配属于你

自己的一块小内存，够放 ShellCode 足矣。示意图如下：



(5) 内核感染常用模块，让感染模块帮我们Load DLL

这个方法就有点绕远了，开始了最本质最原始的感染，可增加新节，可插空隙，总之，让别人的模块Load进内存时顺路的帮我们加载下DLL，DLL一旦加载就可以恢复感染，清除痕迹。至于感染代码，网上一堆。只要不是驱动感染驱动（多了个校验和），其他性质都一样，看自己发挥啦。

(6) 拦截NtCreateUserProcess、NtCreateSymbolicLinkObject

前者在Vista下才有. 拦截后通过 **PsLookupProcessThreadByCid** 得到ETHREAD / EPROCESS, 判断是否是 **CSRSS.EXE** 引起的，若是则在此进程空间内分配一块内存，调用 **NtGetContextThread** 得到当前的线程上下文，调用 ZwWriteVirtualMemory 填充 Shellcode 区域，取得 **LdrUnloadDll**、**LdrGetDllHandle** 等函数地址，通过他们加载 DLL。然后调用 **NtSetContextThread** 恢复原始的 Context。关于这种方法，可参考 DriverDevelop 上某人发的 BIN。

["内核实现DLL注入.可以完美绕过KAV瑞星等杀毒软件"]

（7）内核拦截NtResumeThread

(8) NtUserSetWindowsHookEx 注入

顺便提下 R3 上 DLL 的注入:

1. **CreateRemoteThread** (or **NtCreateThreadEx** (Used in Vista))
2. **SetThreadContext** (change the EIP)
3. **NtQueueAPCThread**
4. **RtlCreateUserThread**
5. **SetWindowHookEx**

小结:

纵观进程启动的全过程，可 patch 的地方很多，只要保证进程、线程上下文不被破坏，注入的手法可多种多样。只要保证我们的 DLL 注入时间足够短，稳定性足够高即可。当我们迫不得已要从内核注入 DLL 到用户进程去时，系统已经中毒很深，此时运用类似上面提到的技术来加载 DLL，让 DLL 做我们驱动无法完成的任务，是可以接受的。

上面提到的思路是我暂时想到并且已经实现了的，详细过程可参见代码。欢迎积极探讨更好更稳定而且**不邪恶**的方法。

参考资料：

1. Windows Internals---Processes,Threads,Jobs
2. 漫谈兼容内核系列:WINDOWS进程的用户空间
3. WRK 1.2 ,windbg