进程隐藏学习总结

转载

bcbobo21cn 于 2016-05-08 15:11:54 发布 3718 收藏 6
分类专栏： 转载 VC++ 安全编程 操作系统研发和研究 文章标签： 进程隐藏

怎么隐藏进程

工具/原料
HideToolz
步骤/方法
1
在百度上面搜索HideToolz，打开第一个搜索结果，点击进入下载。把HideToolz 下载到你的电脑里面。

2
鼠标双击打开该压缩包，再直接双击该软件即可打开该软件了。并且在软件里面可以看见目前的进程数为多少个。

3
按Ctrl + Alt + .【启动任务管理器】，在 HideToolz 里面找到你想要隐藏的运行程序，鼠标单击右键——选择隐藏，然后在任务管理器里面就很清楚的看见你想隐藏的运行程序已经不见了。

4
既然隐藏了运行程序，那么想显示出来了又该怎么办呢？在 HideToolz 里面找到你已经隐藏了的运行程序，鼠标单击右键——选择显示即可。

5
当然，运行 HideToolz 的时候最好也把 HideToolz 的托盘图标也隐藏了。

END
注意事项
HideToolz 下载方式如下图所示：
除了第三方软件可以隐藏进程，还有使用其他的方法也可以隐藏进程，不过相比较第三方软件去隐藏进程的步骤要复杂些。
========

进程隐藏工具

进程隐藏工具可以免费激活了,并增加定时提醒、定时关机和自动粘贴功能。进程隐藏工具集成了窗口隐藏老板键、文件隐藏软件、进程隐藏软件的主要功能，操作方便、功能实用。其独有的开机自动隐藏功能可轻松实现BT、eMule、迅雷等工具自动隐藏下载。亦是上班族在上网、玩游戏、炒股时用来快速隐藏窗口及隐藏图标的最佳工具。
中文名 进程隐藏工具 软件大小 717 KB 软件语言 简体中文 软件类别 国产软件
目录

软件信息
软件大小：717 KB

软件语言：简体中文
软件类别：国产软件
应用平台：Win 7/Windows Vista/Win2003/WinXP/Win2000
软件介绍
进程隐藏工具以强化隐私保护为核心设计理念，隐藏彻底且不露痕迹。　主要特点：　1、可使被指定为自动隐藏的程序以隐藏方式运行，即使用这些程序自身的快捷键也无法令其显示。2、

可通过键盘或鼠标来迅速隐藏指定的程序。3、设置访问口令以防止他人查看隐藏列表或更改您的设置。4、进程、文件、文件夹隐藏管理。5、快速隐藏时可自动关闭声音。6、可设置窗口透
明度(半透明窗口在几米之外很难察觉)7、可设置虚拟桌面,可以将不同的程序运行在不同的桌面下。8、可快速锁定屏幕,使他人无法操作电脑。
========

进程隐藏的各种方法 以及分析比较以及实现链接

典型进程隐藏技术

1 基于系统服务的进程隐藏技术

在 W I N 9X 系列操作系统中, 系统进程列表中不能看到任何系统服务进程, 因此只需要将指定进程注册为系统服务就能够使该进程从系统进程列表中隐形
在win9x下用RegisterServiceProcess函数隐藏进程，NT架构下用不了 即win2000 xp等什么的用不了此方法。

2 基于API HOOK的进程隐藏技术

API HOOK指的是通过特殊的编程手段截获WINDOWS系统调用的API函数,并将其丢弃或者进行替换。 通过API HOOK编程方法,截获系统遍历进程函数并对其进行替换,可以实现对任意进程的隐藏

3 基于 DLL 的进程隐藏技术:远程注入Dll技术

DLL文件没有程序逻辑,不能独立运行,由进程加载并调用,所以在进程列表中不会出现DLL文件。如果是一个以DLL形式存在的程序,通过某个
已有进程进行加载, 即可实现程序的进程隐藏。在windows系统中, 每个进程都有自己的私有地址空间, 进程不能创建属于另一个进程的内存指针
。而远程线程技术正是通过特殊的内核编程手段, 打破进程界限来访问另一进程的地址空间, 以达到对自身 进行隐藏的目的。
远程线程注入DLL技术指的是通过在某进程中创建远程线程的方法进入该进程的内存空间, 然后在其内存空间中加载启动DLL程序。

4 基于远程线程注入代码的进程隐藏技术

这种方法与远程线程注入 DLL 的原理一样,都是通过在某进程中创建远程线程来共享该进程的内存空间。所不同的是,远程线程注入代码通过直接
拷贝程序代码到某进程的内存空间来达到注入的目的。因为程序代码存在于内存中, 不仅进程列表中无法检测, 即使遍历进程加载的内存模块也无法
找到被隐藏程序的踪迹。

5 Rootkit方式

Intel CPU 有4 个特权级别： Ring 0， Ring 1， Ring 2， Ring 3。Windows 只使用了其中的 Ring 0 和 Ring 3 两个级别。
操作系统分为内核和外壳两部分：内核运行在Ring0级，通常称为核心态（或内核态），用于实现最底层的管理功能，在内核态
可以访问系统数据和硬件，包括处理机调度、内存管理、设备管理、文件管理等；外壳运行在 Ring 3 级，通常称为用户态，是基于内核提供的交互功能而存在的界面，它负责指令传递和解释。通常情况下，用户态的应用程序没有权限访问核心态的地址空间。
Rootkit 是攻击者用来隐藏自己的踪迹和保留 root 访问权限的工具，它能使攻击者一直保持对目标机器的访问，以实施对目标计算机的控制[1]。从 Rootkit 运行的环境来看，可将
其分为用户级 Rootkit 和内核级 Rootkit。
用户态下，应用程序会调用 Wi n32 子系统动态库（包括Kernel32.dll， User32.dll， Gdi32.dll等） 提供的Win32 API函数，它们是 Windows 提供给应用程序与操作系统的接口，运行在Ring 3 级。用户级 Rootkit 通常就是通过拦截 Win32 API，建立系统钩子，插入自己的代码，从而控制检测工具对进程或服务的遍历调用，实现隐藏功能。
内核级 R o o t k it 是指利用驱动程序技术或其它相关技术进入Windows 操作系统内核，通过对 Windows 操作系统内核相关的数据结构或对象进行篡改，以实现隐藏功能。
由于Rootkit 运行在 Ring 0 级别，甚至进入内核空间，因而可以对内核指令进行修改，而用户级检测却无法发现内核操作被拦截。

下面介绍两种使用 Rootkit 技术来实现进程隐藏的方法。册表来实现启动,因而易于被检测出来。显然,要增强进程的隐蔽性,关键在于增强加载程序文件的隐藏性。

<1> SSDT Hook

参考本文最下面的链接

<2> DKOM （Direct Kernel Object Manipulation，直接内核对象操作）

使用DKOM方法进行进程隐藏。在Windows操作系统中，系统会为每一个活动进程创建一个进程对象EPROCESS，为进程中的每一个线程创建一个线程对象 ETHREAD。

在 EPROCESS 进程结构中有个双向链表 LIST_ENTRY，LIST_ENTRY结构中有FLINK 和BLINK 两个成员指针，分别指向当前进程的前驱进程和后继进程。

如果要隐藏当前进程，只需把当前进程的前驱进程的BLINK 修改为当前进程的BLINK，再把当前进程的后继进程的FLINK修改为当前进程的FLINK。

进程隐藏方法	隐蔽性	健壮性	实现难度	加载方式	进程存在方式	操作系统
注册系统服务	较差	很好	简单	简单	存在于系统服务	win9x 系列
API HOOK	很好	很好	较难	简单	仍以进程存在	win NT及以上
注册表注入DLL	一般	很差	简单	简单	以DLL形式存在于所有加载user32.dll的进程中	winNT及以上
Rundll32加载DLL	一般	好	简单	简单	以DLL形式存在于Rundll32.exe进程中	winNT及以上
远程线程注入DLL	较好	很好	较难	较复杂	以DLL形式存在于任一进程中,包括系统进程	winNT及以上
远程线程注入代码	很好	较好	很难	很复杂	存在于任一进程的内存空间中,包括系统进程	winNT及以上

========

进程隐藏与进程保护（SSDT Hook 实现）（一）

http://www.cnblogs.com/BoyXiao/archive/2011/09/03/2164574.html

文章目录：

1. 引子 – Hook 技术：

前面一篇博文呢介绍了代码的注入技术(远程线程实现)，博文地址如下：

http://www.cnblogs.com/BoyXiao/archive/2011/08/11/2134367.html

虽然代码注入是很老的技术了，但是这种技术也还是比较常见，

当然也比较好用的，比如在 Spy++ 中就使用了远程线程注入技术，

同时，如果有兴趣的阅读过 Spy++ 的源码的朋友，当然也可以在其源码中阅读到关于远程线程注入技术了。

（这篇博文虽然我会截断分为两篇博文撰写，但是博文仍然会比较长，内容其实是比较多的，覆盖面也比较广，

需要有一定耐心和基础方可阅读完，有兴趣者请自备茶水以及零食，然后慢慢阅读全文，

PS：这话引用自园子里某位园友）

（然后的话就是漫漫长夜，心情不佳，于是写了篇博文，刚好又喝了点，所以估计会有些许疏漏之处，还请见谅 ~）

在这一篇博文中呢，介绍的是一种 Hook 技术，对于 Hook 技术，可以分为两块，

第一块是在 Ring3 层的 Hook，俗称应用层 Hook 技术，

而另外一块自然是在 Ring0 层得 Hook，俗称为内核层 Hook 技术，

而在 Ring3 层的 Hook 基本上可以分为两种大的类型，

第一类即是 Windows 消息的 Hook，第二类则是 Windows API 的 Hook。

关于 Hook 的几种类型呢，下面给出几个简洁的图示：

image

image

image

关于 Windows 消息的 Hook，相信很多朋友都有接触过的，因为一个 SetWindowsHookEx 即可以完成消息 Hook，
在这里简要介绍一下消息 Hook，消息 Hook 是通过 SetWindowsHookEx 可以实现将自己的钩子插入到钩子链的最前端，
而对于发送给被 Hook 的窗口(也有可能是所有的窗口，即全局 Hook)的消息都会被我们的钩子处理函数所捕获到，
也就是我们可以优先于窗体先捕获到这些消息，Windows 消息 Hook 可以实现为进程内消息 Hook 和全局消息 Hook，
对于进程内消息 Hook，则可以简单的将 Hook 处理函数直接写在这个进程内，即是自己 Hook 自己，
而对于用途更为广泛的全局消息 Hook，则需要将 Hook 处理函数写在一个 DLL 中，
这样才可以让你的处理函数被所有的进程所加载(进程自动加载包含 Hook 消息处理函数的 DLL)。
对于 Windows 消息 Hook 呢，可以有个简单的邪恶应用，就是记录键盘按键消息，
从而达到监视用户输入的键值信息的目的，这样，对于一些简单的用户通过键盘输入的密码就可以被 Hook 获取到，
因为没当用户按下一个键时，Windows 都会产生一个按键消息(当然有按下，弹起等消息的区分)，
然后我们可以 Hook 到这个按键消息，这样就可以在 Hook 的消息处理函数中获取到用户按下的是什么键了。

当然关于消息 Hook 的话，其不是这篇博文的重点，
这篇博文主要介绍的是 SSDT Hook 技术，即内核 Hook 技术的一种，
这种技术呢，也是比较老的技术了，貌似是当年 Rootkit 起火的时候出来的，
但是 SSDT Hook 现在也还比较流行，比如在很多的杀毒软件或者安全软件里面也都会使用到 SSDT Hook 技术。
关于内核 Hook 也有几种类型，下面也给出一副图示：

image

上面的几种内核级 Hook 技术，在看雪啊，debugman，xfocus 上都有很多的介绍，
而我只不过是落后这些技术很多年的小辈后生，在这里也只是将自己的学习以及一些总结的经验给列出来而已，
如果有兴趣想深入了解这些内容的话，完全可以在看雪上找到资料 ~

2. SSDT 简介：

以下介绍来自百度(PS:被百度文库弄去了很多博文，这里也抄它一下)：

SSDT 的全称是 System Services Descriptor Table，系统服务描述符表。

这个表就是一个把 Ring3 的 Win32 API 和 Ring0 的内核 API 联系起来。

SSDT 并不仅仅只包含一个庞大的地址索引表，它还包含着一些其它有用的信息，诸如地址索引的基地址、服务函数个数等。

通过修改此表的函数地址可以对常用 Windows 函数及 API 进行 Hook，从而实现对一些关心的系统动作进行过滤、监控的目的。

一些 HIPS、防毒软件、系统监控、注册表监控软件往往会采用此接口来实现自己的监控模块。

在 NT 4.0 以上的 Windows 操作系统中，默认就存在两个系统服务描述表，这两个调度表对应了两类不同的系统服务，

这两个调度表为：KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow，

其中 KeServiceDescriptorTable 主要是处理来自 Ring3 层得 Kernel32.dll 中的系统调用，

而 KeServiceDescriptorTableShadow 则主要处理来自 User32.dll 和 GDI32.dll 中的系统调用，

并且 KeServiceDescriptorTable 在 ntoskrnl.exe(Windows 操作系统内核文件，包括内核和执行体层)是导出的，

而 KeServiceDescriptorTableShadow 则是没有被 Windows 操作系统所导出，

而关于 SSDT 的全部内容则都是通过 KeServiceDescriptorTable 来完成的 ~

从下面的截图可以看出 KeServiceDescriptorTable 在 ntoskrnl.exe 中被导出：

image

然后我们再来看看在 Windows 操作系统的源码 WRK 中，KeServiceDescriptorTable 是怎么被定义的 ~

首先来看 KeServiceDescriptorTable 是如何被 Windows 操作系统源码给导出的：

从下面的截图可以看出，这个系统服务描述表是在 WRK 源码中的某一个模块划分文件(.def)中所导出的。

关于 WRK 是什么东西？则可以参阅我的另一篇博文《Windows 内核(WRK)简介》，博文地址如下：

http://www.cnblogs.com/BoyXiao/archive/2011/01/08/1930904.html

image

而在 Windows 源码 WRK 中对于系统服务描述符表的代码定义如下(KeServiceDecriptorTable 即由该结构定义)：

image

上面的这个结构定义在成员变量的名称上还看不出什么名堂，下面给出我们将在自己代码中所使用的结构体：

```
typedef struct _KSYSTEM_SERVICE_TABLE
{
    PULONG  ServiceTableBase;         // SSDT (System Service Dispatch Table)的基地址
    PULONG  ServiceCounterTableBase;  // 用于 checked builds, 包含 SSDT 中每个服务被调用的次数
    ULONG   NumberOfService;          // 服务函数的个数, NumberOfService * 4 就是整个地址表的大小
    ULONG   ParamTableBase;           // SSPT(System Service Parameter Table)的基地址

} KSYSTEM_SERVICE_TABLE, *PKSYSTEM_SERVICE_TABLE;

typedef struct _KSERVICE_TABLE_DESCRIPTOR
{
    KSYSTEM_SERVICE_TABLE   ntoskrnl;  // ntoskrnl.exe 的服务函数
    KSYSTEM_SERVICE_TABLE   win32k;    // win32k.sys 的服务函数(GDI32.dll/User32.dll 的内核支持)
    KSYSTEM_SERVICE_TABLE   notUsed1;
    KSYSTEM_SERVICE_TABLE   notUsed2;

} KSERVICE_TABLE_DESCRIPTOR, *PKSERVICE_TABLE_DESCRIPTOR;

//导出由 ntoskrnl.exe 所导出的 SSDT
extern PKSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTable;
```

有了上面的介绍后，我们可以简单的将 KeServiceDescriptor 看做是一个数组了(其实质也就是个数组)，

在应用层 ntdll.dll 中的 API 在这个系统服务描述表(SSDT)中都存在一个与之相对应的服务，

当我们的应用程序调用 ntdll.dll 中的 API 时，最终会调用内核中与之相对应的系统服务，

由于有了 SSDT，所以我们只需要告诉内核需要调用的服务所在 SSDT 中的索引就 OK 了，

然后内核根据这个索引值就可以在 SSDT 中找到相对应的服务了，然后再由内核调用服务完成应用程序 API 的调用请求即可。

基本结构可以参考下图：

image

3. 应用层调用 Win32 API 的完整执行流程：

有了上面的 SSDT 基础后，我们再来看一下在应用层调用 Win32 API(这里主要指的是 ntdll.dll 中的 API)的完整流程，

这里我们主要是分析 ntdll.dll 中的 NtQuerySystemInformation 这个 API 的调用流程，

(PS:Windows 任务管理器即是通过这个 API 来获取到系统的进程等等信息的)。

先给出一副图示(先记住这里有四个类似的 API，但是必须得注意区分开来，弄混淆了就麻烦大了)：

image

再给出这些个 API 的基本的调用流程(让大伙有个印象，至少不会迷失)：

image

首先，使用 PE 工具来打开 ntdll.dll 文件，可以看到 NtQuerySystemInformation，

image

除了 NtQuerySystemInformation 外，同时还可以看到 ZwQuerySystemInformation，

image

而实质上，在 Windows 操作系统中，

Ntdll.dll 中的ZwQuerySystemInformation 和 NtQuerySystemInformation 是同一函数，

可以通过下面的截图看出，这两个函数的入口地址指向同一区域，他们的函数入口地址都是一样的 ~

很奇怪吧 ~ 其实我也觉得奇怪 ~ 何必多此一举呢 ~

image

众所周知 Ntdll.dll 中的 API 都只不过是一个简单的包装函数而已，

当 Kernel32.dll 中的 API 通过 Ntdll.dll 时，会完成参数的检查，

再调用一个中断(int 2Eh 或者 SysEnter 指令)，从而实现从 Ring3 进入 Ring0 层，

并且将所要调用的服务号(也就是在 SSDT 数组中的索引值)存放到寄存器 EAX 中，

并且将参数地址放到指定的寄存器(EDX)中，再将参数复制到内核地址空间中，

再根据存放在 EAX 中的索引值来在 SSDT 数组中调用指定的服务 ~

经过上面的步骤后，便由 Ring3 层进入了 Ring0 层，

我们再通过 PE 工具来查看 ntoskrnl.exe 中的 ZwQuerySystemInformation 和 NtQuerySystemInformation

image

image

先来看 ntoskrnl.exe 中的 ZwQuerySystemInformation：

image

在上面的这幅截图中，可以看到在 Ring0 下的 ZwQuerySystemInformation 将 0ADh 放入了寄存器 eax 中，

然后调用了系统服务分发函数 KiSystemService，而这个 KiSystemService 函数则是根据 eax 寄存器中的索引值，

然后再 SSDT 数组中找到索引值为 eax 寄存器中存放的值得那个 SSDT 项，

最后就是根据这个 SSDT 项中所存放的系统服务的地址来调用这个系统服务了 ~

比如在这里就是调用 KeServiceDescriptorTable[0ADh] 处所保存的地址所对应的系统服务了 ~

也就是调用 Ring0 下的 NtQuerySystemInformation 了 ~

至此，在应用层中调用 NtQuerySystemInformation 的全部流程也就结束了 ~

最后，贴出一点在 Ring0 下的 NtQuerySystemInformation 的反汇编代码：

image

4. 详解 SSDT：

在这一节里面，我们将来看看 SSDT 到底是个什么东西 ~ 这里使用 WinDbg 来调试 XP SP2 系统 ~

首先来看看 KeServiceDescriptorTable 是何物？

从下面的截图中可以看到 KeServiceDesciptorTable 的首地址为 804e58a0，

然后查看分析这个地址，可以查看到第一个系统服务的入口地址为 80591bfb ！

2011-08-18_012703

我们再来看看 80591bfb 这个地址对应的究竟是何系统服务？

从下面的截图中，可以看到 SSDT 中第一个系统服务就是 NtAcceptConnectPort !!!

2011-08-18_013027

由于我们知道了 SSDT 的首地址，又知道了 Ring0 下 NtQuerySystemInformation 服务的索引号，

所以可以根据 “SSDT 中系统服务地址所在的 Address = SSDT 首地址 + 4 * 索引号”,

推算出 NtQuerySystemInformation 服务的地址，

因此有 Address = 804e58a0 + 4 * 0adh = 804E5B54；

然后我们再来看 804E5B54 这个地址的信息，信息如下截图：

从截图中，我们可以看到 NtQuerySystemInformation 的起始地址为 80586ff1，

2011-08-18_020103

下面就来验证一下地址 80586ff1 到底是不是 NtQuerySystemInformation 的首地址 ~

从下面的截图中可以肯定 80586ff1 确实就是 NtQuerySystemInformation 的首地址，

这和我们上面对 SSDT 中指定索引号的服务的地址的计算公式计算出来的结果是统一的 !!!

2011-08-18_020231

从上面的介绍，可以看出，其实 SSDT 就是一个用来保存 Windows 系统服务地址的数组而已 !!!

5. SSDT Hook 原理：

有了上面的这部分基础后，就可以来看 SSDT HOOK 的原理了，

其实 SSDT Hook 的原理是很简单的，从上面的分析中，

我们可以知道在 SSDT 这个数组中呢，保存了系统服务的地址，

比如对于 Ring0 下的 NtQuerySystemInformation 这个系统服务的地址，

就保存在 KeServiceDescriptorTable[0ADh] 中，

既然是 Hook 的话，我们就可以将这个 KeServiceDescriptorTable[0ADh] 下保存的服务地址替换掉，

将我们自己的 Hook 处理函数的地址来替换掉原来的地址，

这样当每次调用 KeServiceDescriptorTable[0ADh]时就会调用我们自己的这个 Hook 处理函数了。

下面用几幅截图来表示：

image

下面的截图则是 SSDT Hook 之后了，可以看到将 SSDT 中的服务地址修改为 MyHookNtQuerySystemInformation 了，

这样的话，每次系统调用 NtQuerySystemInformation 这个系统服务时，

实质上调用的就是 MyHookNtQuerySystemInformation 了，而我们为了保证系统的稳定性(至少不让其崩溃)，

一般会在 MyHookNtQuerySystemInformation 中调用系统中原来的服务，也就是 NtQuerySystemInformation。

image

6. 小结：

本篇博文呢尚还只是介绍了 SSDT 到底是个什么东西，而还没有给出具体的 SSDT Hook 的实现，

对于 SSDT Hook 的实现以及 Demo 我都放到(二)中完成，也就是本篇博文未完 , 待续 ……

关于 SSDT 的话，在看雪上有很多的文章，由于我也是前阵子对这东西突然感兴趣了，

所以我也算是初次了解，自然也看过了很多的文章，SSDT 在 Google 一搜索可以出来一大堆，

但是要说介绍 SSDT 最详细的话，我想还是我的这篇文章介绍的比较详细，

因为网上很多介绍 SSDT 的都只是将 SSDT 原理做了简单的介绍，然后在网上 down 一个 Demo，

把代码贴出来就完事了，甚至是代码都还无法完整编译通过的，

所以如果读者想对 SSDT 有所了解的话，可以好好看一看这篇文章的 ~

顺便这里还带出一个问题，是我这阵子脑子里突然冒出来的一个疑问，

但是由于时间或者说是个人状态问题，一直没有去研究 ~ 不晓得园子里有木有对这个有研究的 ~

众所周知，在 Windows 操作系统中，System 进程的进程 PID 为 4，

我想问的就是：System 进程的 PID 为何是 4？

欢迎大家对这个问题讨论啊 ~ 在这里先给点思路，

那就是可以通过 Windows 操作系统的启动过程，然后结合 WRK 源码进行研究 ~

========