进程隐藏与进程保护（SSDT Hook 实现）（一）

转载

cmd1115 于 2012-07-17 13:59:40 发布 675 收藏
分类专栏： C++ 文章标签： hook descriptor windows table api service

转载自 Zachary.XiaoZhen - 梦想的天空

进程隐藏与进程保护（SSDT Hook 实现）（一）

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，

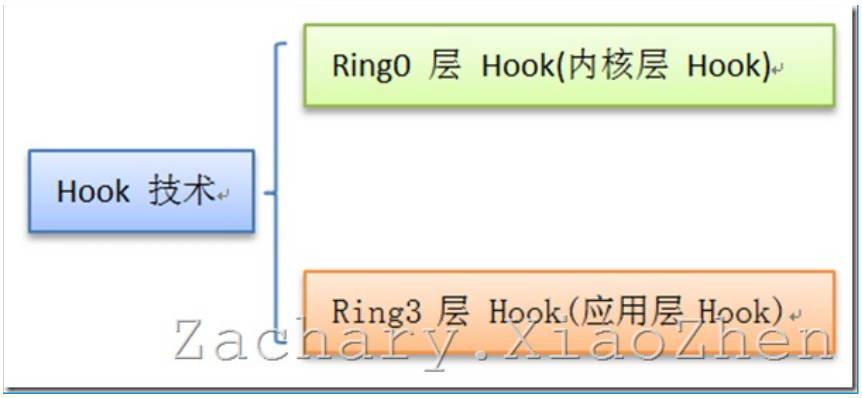
第一块是在 Ring3 层的 Hook，俗称应用层 Hook 技术，

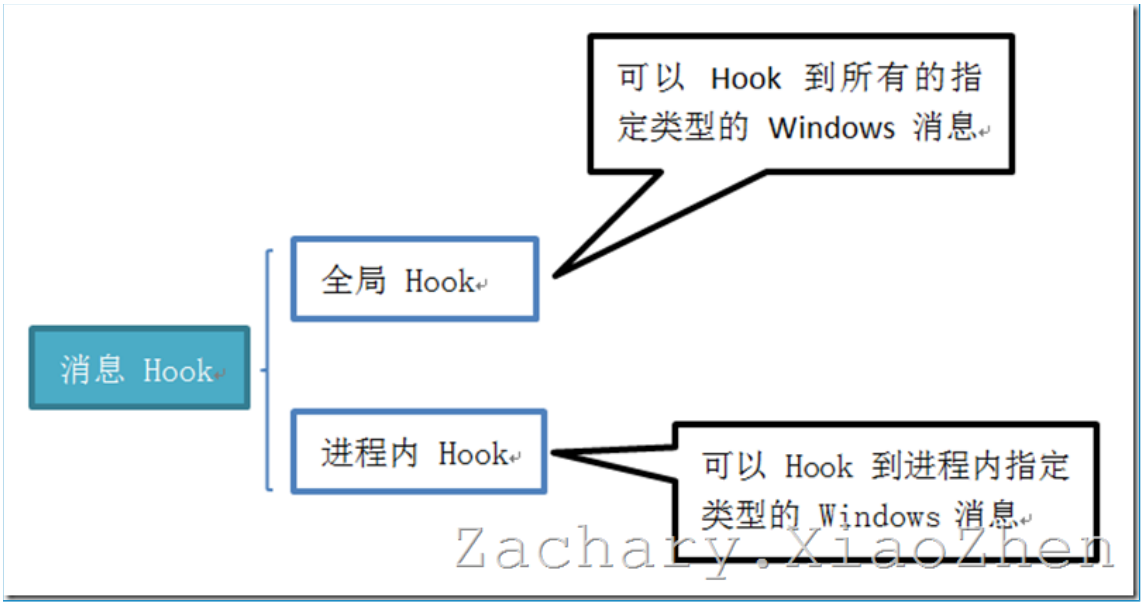
而另外一块自然是在 Ring0 层得 Hook，俗称为内核层 Hook 技术，

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：





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：

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，

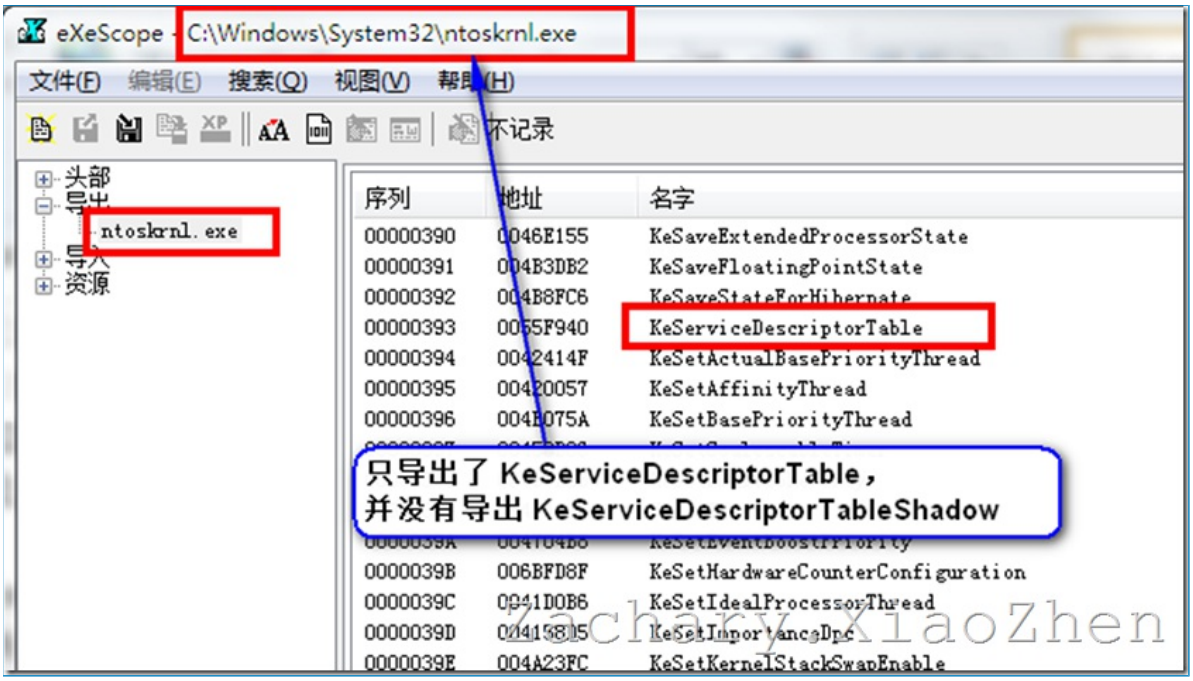
而 KeServiceDescriptorTableShadow 则主要处理来自 User32.dll 和 GDI32.dll 中的系统调用，

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：



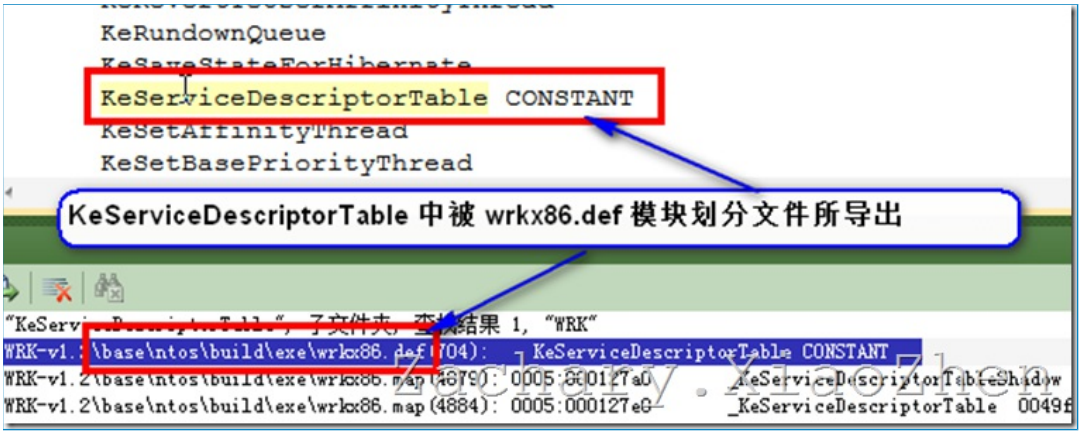
然后我们再来看看在 Windows 操作系统的源码 WRK 中，KeServiceDescriptorTable 是怎么被定义的 ~

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

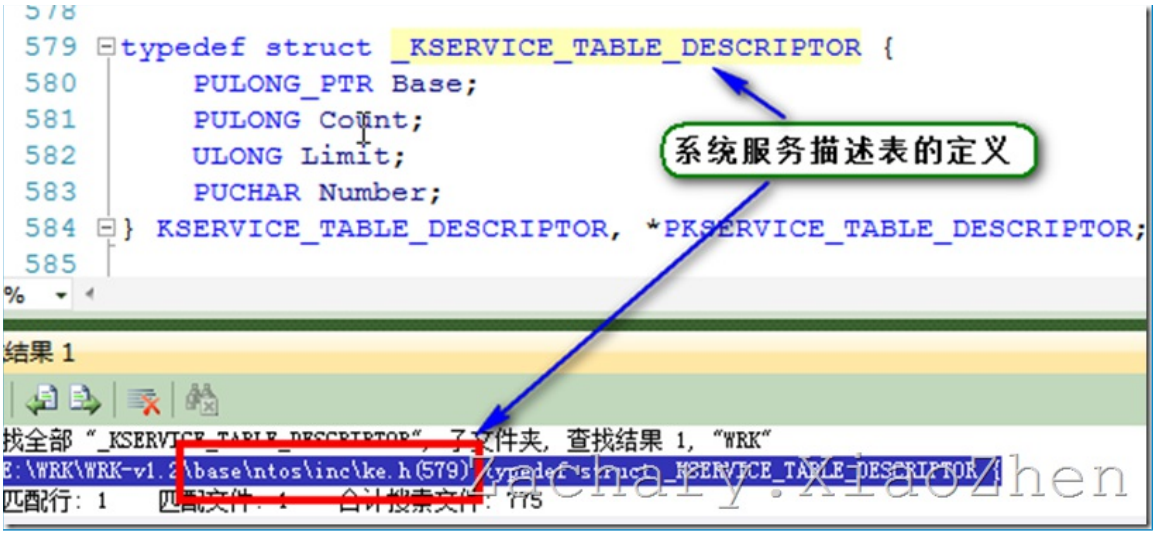
从下面的截图可以看出，这个系统服务描述表是在 WRK 源码中的某一个模块划分文件(.def)中所导出的。

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



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)：



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：

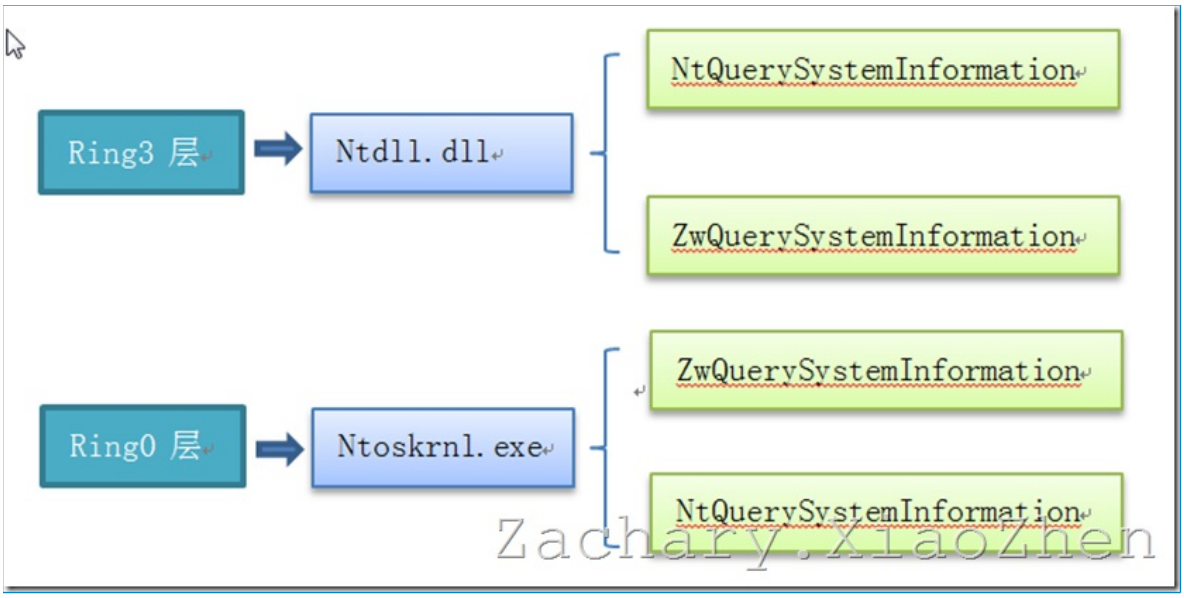
3. 应用层调用 Win32 API 的完整执行流程：

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，

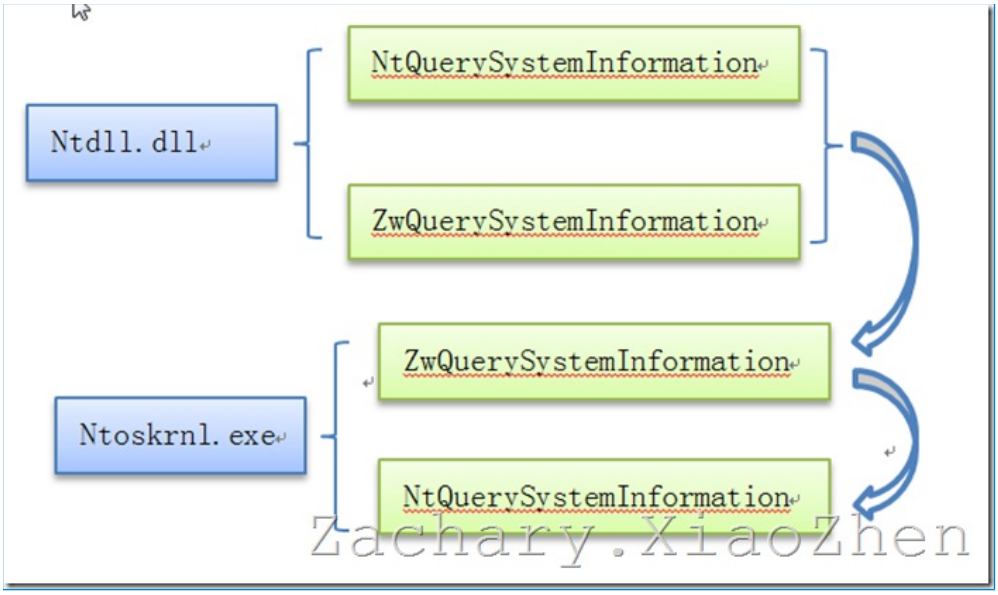
这里我们主要是分析 ntdll.dll 中的 NtQuerySystemInformation 这个 API 的调用流程，

(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)。

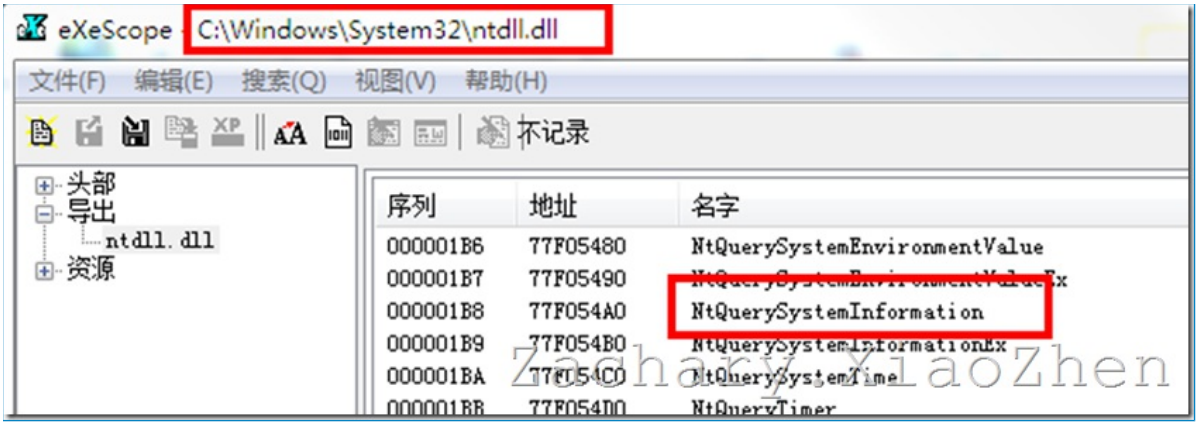
先给出一副图示(先记住这里有四个类似的 API，但是必须得注意区分开来，弄混淆了就麻烦大了)：



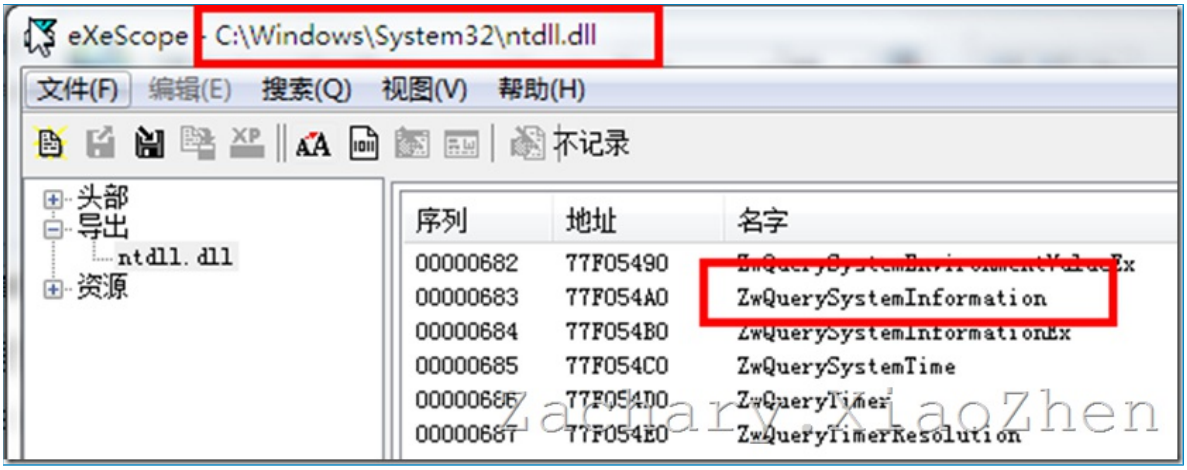
再给出这些个 API 的基本的调用流程(让大伙有个印象，至少不会迷失)：



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



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



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 ~

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，

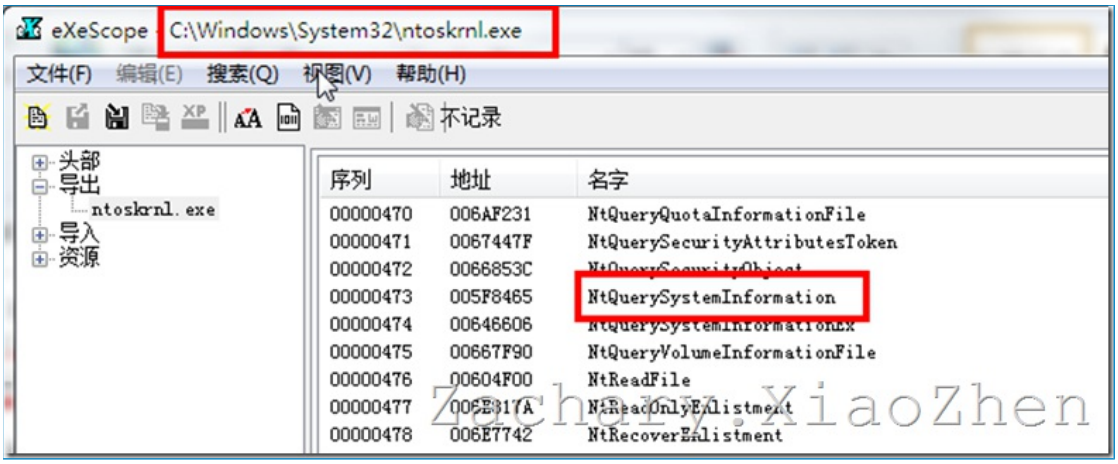
并且将所要调用的服务号(也就是在 SSDT 数组中的索引值)存放到寄存器 EAX 中，

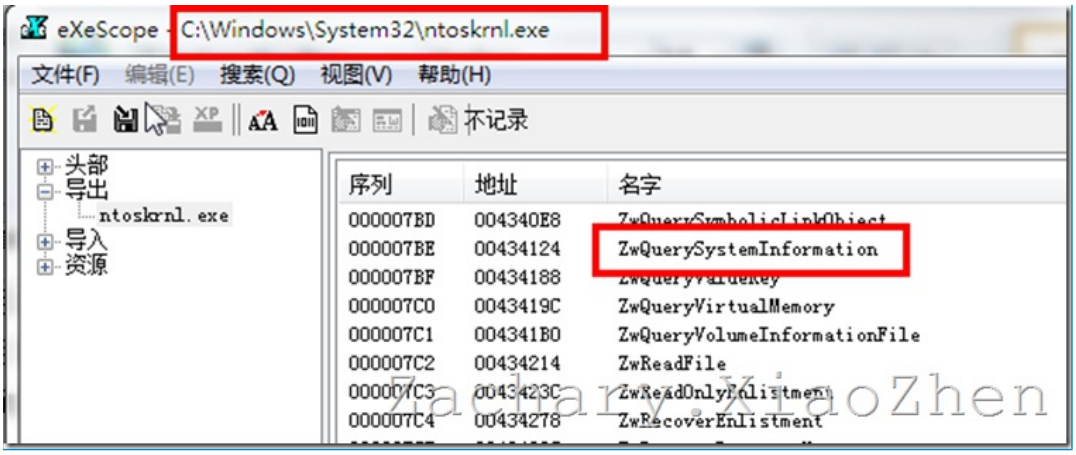
并且将参数地址放到指定的寄存器(EDX)中，再将参数复制到内核地址空间中，

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 ~

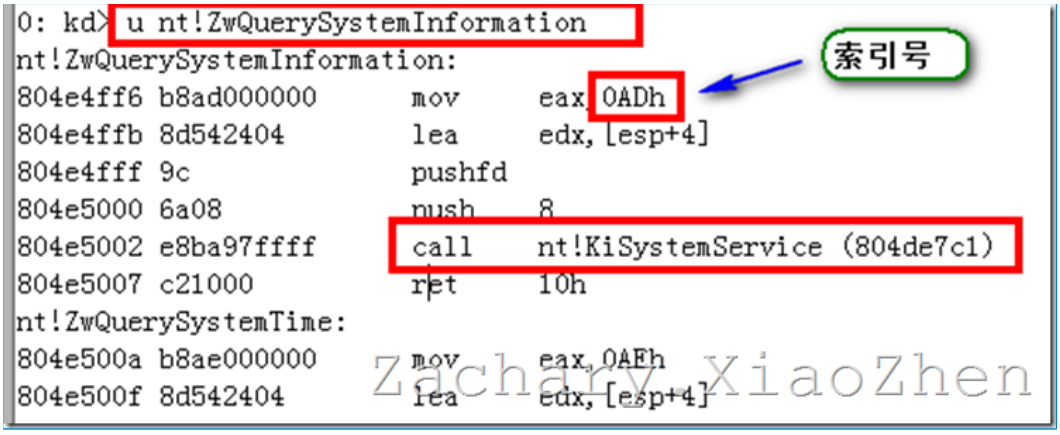
经过上面的步骤后，便由 Ring3 层进入了 Ring0 层，

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





先来看 ntoskrnl.exe 中的 ZwQuerySystemInformation：



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，

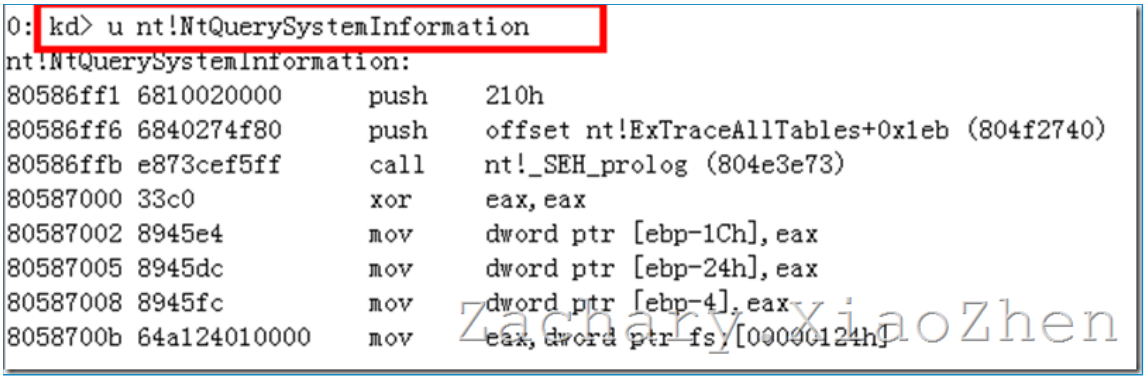
最后就是根据这个 SSDT 项中所存放的系统服务的地址来调用这个系统服务了 ~

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：



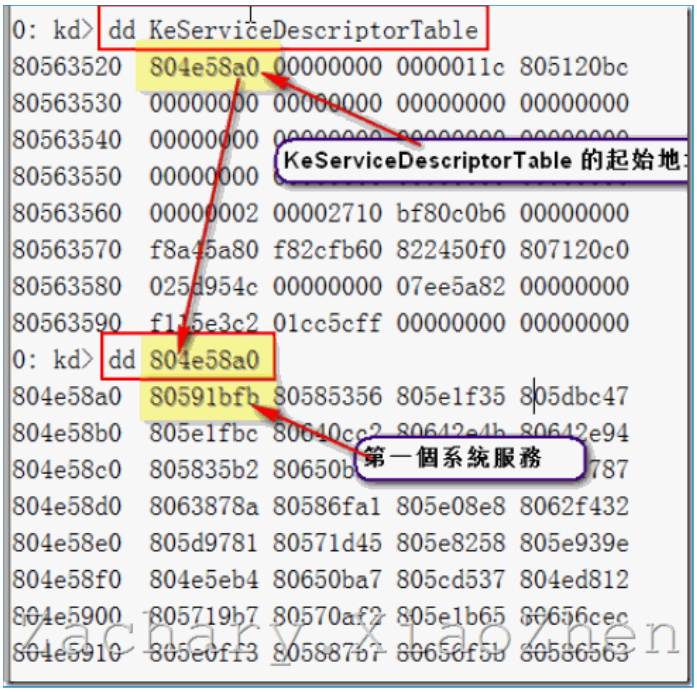
4. 详解 SSDT：

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 ！



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 !!!

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，

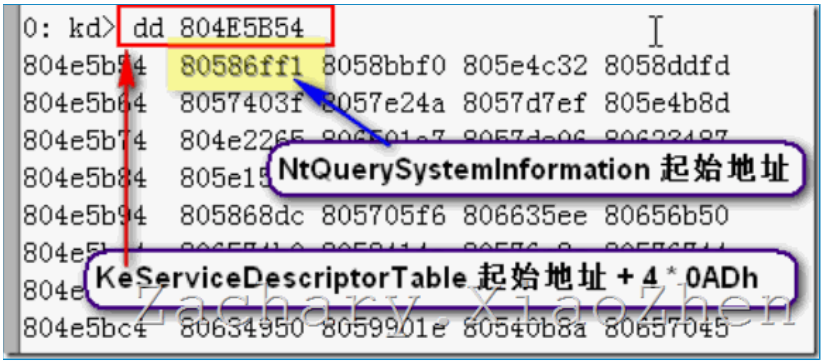
所以可以根据 “SSDT 中系统服务地址所在的 Address = SSDT 首地址 + 4 * 索引号”，

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：

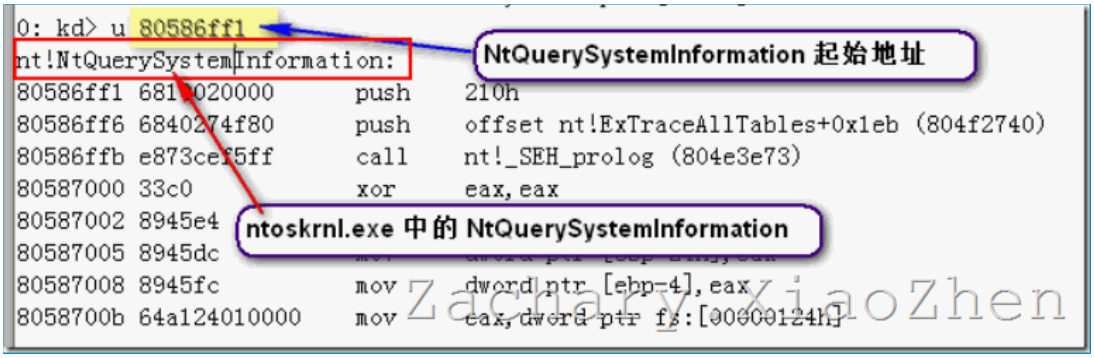
从截图中，我们可以看到 NtQuerySystemInformation 的起始地址为 80586ff1，



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 !!!



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，

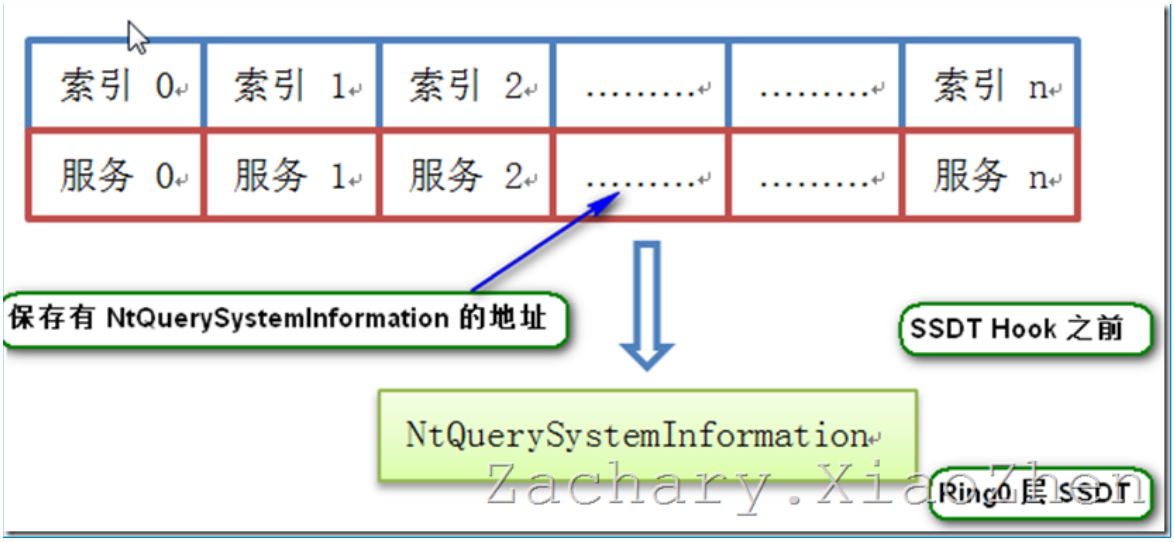
就保存在 KeServiceDescriptorTable[0ADh] 中，

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：

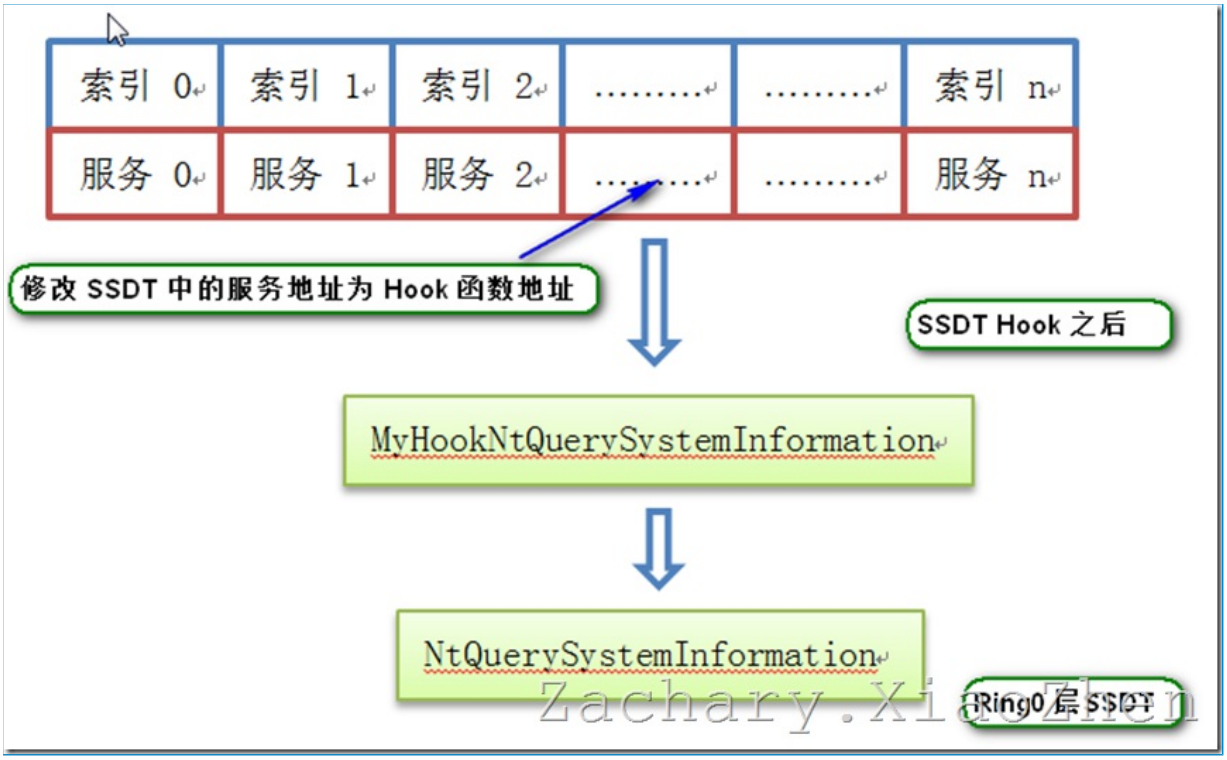


下面的截图则是 SSDT Hook 之后了，可以看到将 SSDT 中的服务地址修改为 MyHookNtQuerySystemInformation 了，

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。



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 ~