doi 10.3969/j.issn.1671-1122.2014.07.005

基于 Windows 的 CSRSS 进程漏洞分析与利用

李孟哲[1]，武学礼[2]，张涛[1]，文伟平[1]

（1. 北京大学软件与微电子学院，北京 102600 ；2. 中国石油集团东方地球物理勘探有限责任公司，陕西长庆 710021）

摘 要 ：随着技术的进步，Windows 操作系统日益完善，多种内存保护技术的结合使得传统的基于缓冲区溢出攻击越来越困难。在这种情况下，内核漏洞往往可以作为突破安全防线的切入点，一旦漏洞被病毒、木马利用，将会彻底瓦解安全软件的所有防御，沉重打击系统安全。随着 Windows NT 的开发，操作系统被设计成可以支持多个子系统，包括 POSIX、OS/2 以及 Windows 子系统（也被称为客户端 / 服务器运行时子系统或者 CSRSS）。文章展开了一系列关于 CSRSS 的研究，描述了 CSRSS 内部机制。尽管一些研究已经在少数文章中有所提及，但是直到现在没有深入的案例研究。文章详细地介绍了 CSRSS 及其通信机制，以及最近常见于现代操作系统的 CSRSS 变化。另外，站在安全的角度，文章对 Windows 内核漏洞进行了分类，并且提出了一套漏洞研究的流程。按照这套流程，研究了 CSRSS 进程的权限提升漏洞和拒绝服务漏洞。文章通过对 CVE-2011-1281 漏洞的分析，发现 use-after-free 漏洞不仅出现在浏览器漏洞中，在系统软件中同样有可能出现。

关键词 ：Windows 子系统 ；CSRSS ；Windows 内核 ；漏洞分析

中图分类号 ：TP309 文献标识码 ：A 文章编号 ：1671-1122（2014）07-0020-10

Analysis and Exploit of CSRSS Vulnerabilities based on Windows

LI Meng-zhe[1], WU Xue-li[2], ZHANG Tao[1], WEN Wei-ping[1]

(1.*School of Software&Microelectronics, Peking University, Beijing* 102600, *China*;2.*China Petroleum Group Dongfang Geophysical Exploration Co., Ltd.,Changqing Shanxi* 710021,*China*)

Abstract: With advances in technology, Windows operating system has improved steadily. The combination of many memory protection mechanisms makes the traditional buffer-overflow-based attacks to be more useless. In this case, the kernel vulnerabilities can be used to break through the security line of defense as a starting point. If these vulnerabilities are used by viruses and Trojans, the defense of security software will be collapsed. That means a heavy blow to the system security. Since the Microsoft Windows NT's development, the operating system has been designed to support a number of different subsystems, such as POSIX or OS/2. This paper opens a series of CSRSS-oriented study, aiming at describing the uncovered CSRSS mechanism internals. Although some great research has already been carried out by some articles, no thorough case study is available until now. This paper covers both the very basic ideas and their implementations, as well as the recent CSRSS changes applied in modern operating systems. In addition, standing on the point of safety, in this paper, the Windows kernel vulnerabilities are classified, a set of vulnerability research process is presented. According to the process, this article studies local privilege escalation vulnerability and denial of service vulnerability about CSRSS. Through the analysis of the CVE-2011-1281 vulnerability, use-after-free exploit not only appears in the browser vulnerabilities, but also in the software of the system.

Key words: Windows subsystem; CSRSS; Windows kernel; vulnerabilities study

收稿日期 ：2014-05-13

基金项目 ：国家自然科学基金 [61170282]

作者简介 ：李孟哲（1988-），男，湖南，硕士研究生，主要研究方向 ：系统与网络安全、漏洞分析与利用 ；武学礼 (1975 －)，男，宁夏，工程师，本科，主要研究方向 ：软件测试 ；张涛（1987-），男，江西，硕士研究生，主要研究方向 ：系统与网络安全、软件安全漏洞分析 ；文伟平（1976-），男，湖南，副教授，博士，主要研究方向 ：网络攻击与防范、恶意代码研究、信息系统逆向工程和可信计算技术等。

0 引言

Windows 环境子系统进程，即客户端 / 服务器运行时子系统（CSRSS.exe，简称 CSRSS）是 Windows 子系统的服务器进程。尽管从 Windows NT 4.0 开始，窗口管理 (包括屏幕输出、用户输入和消息传递) 和 GDI 的主体实现都移入到内核（win32k.sys）中，但 CSRSS 依然是 Windows 子系统的灵魂 , 它监视和管理着系统内所运行的所有 Windows 进程和线程。每个进程创建之后，都要先到 CSRSS 注册登记后才能运行，结束时也要到此报告注销 [1]。除了掌管着各个进程的生死存亡，CSRSS 在桌面管理，终端 Login、Console 管理、硬件错误报告和 DOS VM 等方面也起着重要作用 [2]。总之，把 CSRSS 称为 Windows 系统的管家不算为过。CSRSS 身担如此多的重任，Windows 系统真的离不开它。如果尝试强行杀死 CSRSS 进程 (使用 kill 或其他工具)，系统则会以蓝屏结束 [3]。

从安全的角度看，CSRSS 属于系统进程，也是系统运行的关键进程，所以不受任何防火墙限制，因此也就有可能被许多病毒或者木马利用。它们通过劫持这个进程来破坏电脑或者利用它来获得权限的提升 [4]。近年来，与 CSRSS 相关的漏洞不断被曝光，其受到了到人们越来越多的的关注，部分安全工作者出于好奇也对与 CSRSS 相关的漏洞进行了一定程度的分析，但直到现在也没有进行深入的案例研究 [5]。

1 Windows 客户端 / 服务器运行时子系统进程分析

1.1 子系统分析

随着 Windows NT 的开发，操作系统被设计成可以支持多个子系统，包括 POSIX、OS/2 以及 Windows 子系统（也被称为客户端 / 服务器运行时子系统或者 CSRSS）[6,7]。虽然 OS/2 子系统在 Windows XP 系统之后就不再被支持，POSIX 子系统也变为选择性支持，但是最原始的 Windows 子系统被保留下来，成为操作系统最核心的部分，且仍然在被不断地开发与完善。

我们知道，子系统中的动态链接库提供的 Windows 应用程序接口包括 kernel32.dll, user32.dll, gdi32.dll, advapi32.dll 等 [8]，它们当中的大多数被 Windows 开发者所使用。

任何一个进程都与一个特定的子系统相关联。在编译过程中，一些属性被链接程序所设定，并且驻留在以下 PE 结构中 [9] ：

```
WORD IMAGE_PE_HEADERS.IMAGE_OPTIONAL_HEADER.Subsystem
```

下面列出其中可用的常量值 :

```
#define IMAGE_SUBSYSTEM_UNKNOWN    0
#define IMAGE_SUBSYSTEM_NATIVE     1
#define IMAGE_SUBSYSTEM_WINDOWS_GUI    2
#define IMAGE_SUBSYSTEM_WINDOWS_CUI    3
#define IMAGE_SUBSYSTEM_OS2_CUI        5
#define IMAGE_SUBSYSTEM_POSIX_CUI      7
#define IMAGE_SUBSYSTEM_NATIVE_WINDOWS    8
#define IMAGE_SUBSYSTEM_WINDOWS_CE_GUI    9
#define IMAGE_SUBSYSTEM_EFI_APPLICATION    10
#define IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER    11
#define IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER    12
#define IMAGE_SUBSYSTEM_EFI_ROM    13
#define IMAGE_SUBSYSTEM_XBOX    14
```

这种可移植、可执行的格式是非常强大的，可以被不同的处理器（如 Intel x86、Intel x86-64、ARM）和系统（Microsoft Windows、Windows CE、EFI、XBOX）用于构建可执行程序。

一个子系统环境用于显示一个严格定义的关于用户应用程序的基本函数子集 [10]。考虑到 Windows NT 的架构和子系统的基本性质，CSRSS 最开始由两个主要部分组成 ：

1）客户端方面的动态链接库。它被映射到客户端进程环境中，并且提供能够直接被 Windows 应用开发者调用的公开且被文档定义过的接口（如 kernel32.dll、user32.dll）。

2）高权限进程 csrss.exe（运行在安全的本地系统用户环境中）。通过接受来自客户端应用的请求，并执行相应的操作，实现 Windows 子系统的实际功能。

1.2 CSRSS基本分析

图 1 是用内核工具 XueTr 观察 CSRSS 进程加载模块。

图1 CSRSS进程加载模块

CSRSS 负责执行一些小任务，如管理控制台窗口（Windows 7 版本之前）、管理进程与线程、支持 DOS 虚拟机以及其他一些功能。由图 1 可知，CSRSS 进程除了加载如 kernel32、ntdll 等基础模块之外，basesrv.dll、csrsrv.dll

winsrv.dll 是 CSRSS 进程的核心模块，它们的主要功能描述如下：

1）BASESRV.DLL（进程和线程列表，DOS 虚拟机支持）。

2）WINSRV.DLL（控制台管理，用户服务）。

3）CSRSRV.DLL（各种其他功能）。

CSRSRV 动态链接库文件里有一个未导出符号叫做 CsrRootProcess，指向一个 CSR_PROCESS 结构体 [11]。

任何一个进程都会对应一个名为 CSR_PROCESS 的结构体，而对应的 CSR_PROCESS 结构体通过 struct _LIST_ENTRY ListLink 形成了一条链表状的数据结构，我们可以利用 CsrRootProcess 来遍历这个链表 [12]。每当一个进程创建的时候，CSRSS 会创建一个新的 CSR_PROCESS 结构体，然后插入到这个 LIST 中；同样，当一个 PROCESS 退出时，CSRSS 会将该进程对应的 CSR_PROCESS 结构体从 LIST 中移除 [13]。

CSRSrv 动态链接库文件里还有一个未导出的 STMBOL CsrThreadHashTable，它是一个拥有 29 个元素的数组，数组中的每一个元素都对应一个 CSR_THREAD 结构体 [14]，该结构体如下所示：

```
typedef struct _CSR_THREAD
{
    union _LARGE_INTEGER      CreateTime;
    struct _LIST_ENTRY        Link;
    struct _LIST_ENTRY        HashLinks;
    struct _CLIENT_ID         ClientId;
    struct _CSR_PROCESS*      Process;
    struct _CSR_WAIT_BLOCK*   WaitBlock;
    void*                     ThreadHandle;
    unsigned long             Flags;
    unsigned long             ReferenceCount;
    unsigned long             ImpersonateCount;
} CSR_THREAD, *PCSR_THREAD;
```

此外，每一个线程也同样对应一个 CSR_THREAD 结构体，成员 struct _CSR_PROCESS* Process 指向这个线程所属进程对应的 CSR_PROCESS 结构体。一个进程的任意线程对应的 CSR_THREAD 通过成员 struct _LIST_ENTRY Link 也构建了一条链表，该链表的头部为 CSR_PROCESS.ThreadList。

2 漏洞研究流程

我们将漏洞研究过程分为 4 个环节：漏洞重现、漏洞分析、漏洞利用和漏洞总结。漏洞研究流程如图 2 所示。

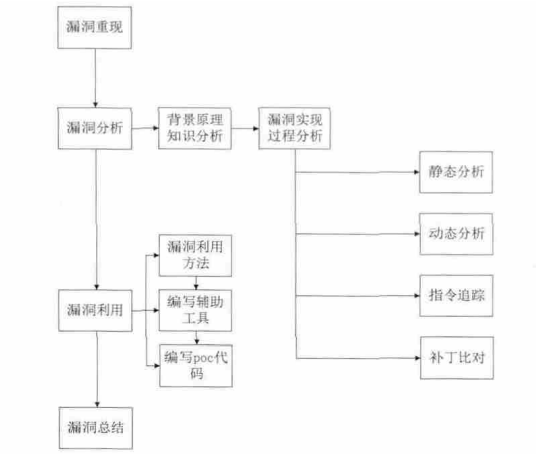

图2 漏洞研究流程

1）漏洞重现

漏洞重现环节首先需要搭建必要的测试环境，通常为虚拟机环境；其次需要重视有漏洞的内核文件或者驱动程序的版本，如果版本不对，是不可能重现的；还要确认该漏洞是否暂时还没有打补丁，如果已经打补丁，需要恢复到之前的系统版本；最后，如果该漏洞已经公开了 POC 源码，还需要对 POC 源码进行编译。我们知道，从漏洞重现到漏洞分析是一个由表及里、联系紧密的过程 [15]。

2）漏洞分析

漏洞分析环节是整个漏洞研究过程中最为核心的环节，如果不能分析清楚漏洞的前因后果，那么漏洞的利用也就无从下手了。漏洞分析其实也是一个刨根问底的过程。

在具体的分析之前，我们通常需要研究漏洞相关的背景知识，如果在不清楚背景知识的情况下直接进行分析，难度会增大很多。只有在熟悉了相关背景理论的情况下，才能对漏洞触发的整个过程有一个清晰的认识。因此，搜集漏洞相关背景理论知识在漏洞分析过程中是一个举足轻重的步骤 [16]。

3）漏洞利用

漏洞利用环节是建立在漏洞分析基础之上的，是编写能够利用该漏洞并且实现特定目标的代码，并进行测试的一个过程。

内核漏洞的主要作用是：远程代码的任意执行、本地权限提升、远程拒绝服务攻击、本地拒绝服务攻击。从漏洞利用的角度来看，远程拒绝服务和本地拒绝服务类型的漏洞利用起来比较简单，并不需要考虑过多的构造（构造

漏洞成功触发的条件和数据)。恰恰相反，远程代码的任意执行和本地权限提升漏洞利用起来相对复杂，往往需要精心的构造，包括触发条件的构造和触发数据的构造。

4) 漏洞总结

漏洞总结环节是在完成了漏洞重现、漏洞分析、漏洞利用过程之后，回头来审视造成这个漏洞的根本原因，并提出修补方法的过程。如果把上面的过程作为攻击，那么漏洞总结应该站立在攻击的对立面上，才能有所感悟与体会，才能寻求到突破。通过漏洞总结，能够将研究过程中获取的知识升华成一种经验和能力[17]。

3 CSRSS 漏洞分析与利用

Microsoft Windows 的 Win32 子系统中的 CSRSS 不能正确限制进程中控制台的数量。本地用户可通过触发不正确内存分配的特制应用程序获取特权或导致拒绝服务。Microsoft Windows CSRSS 子系统在 AllocConsole() 的实现上存在漏洞，本地攻击者可利用此漏洞在内核模式中执行任意代码。此漏洞源于多个控制台对象关联到一个进程时，CSRSS 子系统的 AllocConsole() 实现上存在问题，造成在原始进程终止后，新进程获取孤立对象[18]。

3.1 漏洞重现

我们在做漏洞重现时，首先要搭建好漏洞实现的环境。CVE-2011-1281 的环境包括虚拟机 (VMware 8.0.2) 和系统版本。由于在 Windows 7 中控制台管理几乎完全从 CSRSS 转移到 CONHOST.EXE（在本地用户的上下文中运行），因此 CVE-2011-1281 漏洞出现于 Windows Vista 之前的所有版本。它所诱发漏洞的组件是 Winsrv.DLL（子系统所使用的模块之一），它所诱发漏洞的函数是 WinsrvDll!AllocConsole()[19]。

利用此漏洞，需要对操作系统具有访问权限。然后，攻击者可以运行一个为利用此漏洞而特制的应用程序，从而完全控制受影响的系统。

3.2 漏洞分析

该漏洞是一个关于基于句柄的释放后采用的情形，这种漏洞在之前是从未出现过的。为了更好更清楚地分析该漏洞，我们首先需要分析一下 Windows 句柄的分配机制。

1) 句柄的数据结构

每个进程都会创建一个句柄列表，这些句柄指向各种系统资源，如信号量、线程、文件和注册表等，属于这个进程中的所有线程都可以访问这些资源。

在用户态下如果调用函数 CloseHandle( ) 表示不再需要使用这个对象，内核中进程便会删除句柄 (释放对象的引用)。对象管理器也会将内核对象的引用计数值减 1, 当对象的句柄引用值为 0 时 , 对象管理器便会释放掉这个对象[20]。

句柄表最基本的功能就是句柄和目标对象之间的映射。图 3 描述了进程与句柄之间的映射关系 (有些数据域要经过处理)。

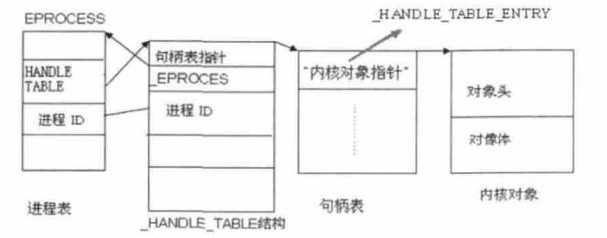

图3 句柄映射关系

_HANDLE_TABLE 是句柄表的信息的结构体 , 在内核中句柄是句柄表中表项的索引，这里可以简单地理解为由索引 (句柄) 在句柄表中查找进程引用的内核对象。

在 Windbg 中查看到的 _HANDLE_TABLE 结构体如下所示。

```
d> dt _HANDLE_TABLE
t!_HANDLE_TABLE
   +0x000 TableCode          : Uint4B      //指向第一层局部表,并记录层数
   +0x004 QuotaProcess       : Ptr32 _EPROCESS  //指向进程_EPROCESS 块
   +0x008 UniqueProcessId    : Ptr32 Void       //进程 ID
0x03c HandleCount         : Int4B       //句柄计数,当前使用的句柄个数
d> dt _EPROCESS      //进程_EPROCESS 块信息
t!_EPROCESS
0x084 UniqueProcessId   : Ptr32 Void        //进程 ID
0x0c4 ObjectTable : Ptr32  HANDLE  TABLE   //指向 HANDLE TABLE 结构体
```

内核和 SDK 中所定义的句柄都是 typedef void *HANDLE，这就表明句柄是一个无符号的整数 , 实际上有效句柄值是有范围的。如果采用普通数组来存储句柄，则需要耗费过多的内存,所以Windows句柄表使用了稀疏数组。

句柄表存放的主要是对象地址与属性信息 , 当然还要存放句柄表相关的其他一些信息 (审计、空闲项)。每个句柄表项是由 _HANDLE_TABLE_ENTRY 来描述的[21]，一个 _HANDLE_TABLE_ENTRY 共占 8 个字节，定义如下：

```
kd> dt _HANDLE_TABLE_ENTRY
nt!_HANDLE_TABLE_ENTRY
   +0x000 Object              : Ptr32 Void     //对象指针
   +0x000 ObAttributes        : Uint4B
   +0x000 InfoTable           : Ptr32 _HANDLE_TABLE_ENTRY_INFO
   +0x000 Value               : Uint4B
   +0x004 GrantedAccess       : Uint4B
   +0x004 GrantedAccessIndex  : Uint2B
   +0x006 CreatorBackTraceIndex : Uint2B
   +0x004 NextFreeTableEntry : Int4B
```

_HANDLE_TABLE_ENTRY 的源代码定义如下：

```
    typedef struct _HANDLE_TABLE_ENTRY {
union {
  PVOID Object;                          // 对象指针
  ULONG ObAttributes;                    // 对象属性
  PHANDLE_TABLE_ENTRY_INFO InfoTable;
  ULONG_PTR Value;                       // 值
  };
    union {
 union {
   ACCESS_MASK GrantedAccess;  // 访问掩码
    struct {
       USHORT GrantedAccessIndex
      };
    };
  LONG NextFreeTableEntry;// 下一个空闲的句柄表项 , 空闲链表索引
};
    } HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

表示的意义是：

（1）对象指针（Object）如果是有效的，则第二个域为访问掩码（GrantedAccess）。

（2）第一个域为 0，第二个域则可能是 NextFreeTableEntry，也可能为审计。后面会有相关算法用到这个域，我们需要根据上下文来判断 [22]。

这里的 Object 并非真正的对象指针，而是包含了对象指针域的对象属性域。由于在内核中对象遵守 8 字节对齐的原则，则指向对象的指针的最低 3 位就总是 0。Microsoft 把这 3 位也利用上了，Object 的最低 3 位作为对象的属性。我们看下面的一组宏定义：

```
#define OBJ_HANDLE_ATTRIBUTES (OBJ_PROTECT_CLOSE | OBJ_INHERIT | OBJ_AUDIT_OBJECT_CLOSE)
```

第 0 位 OBJ_PROTECT_CLOSE：表示句柄表项是否被锁定了，1 为锁定，0 为未锁定。

第 1 位 OBJ_INHERIT：指向该进程，它所创建的子进程是否可以继承这个句柄，即是否可以将该句柄项复制到它的句柄表中。

第 2 位 OBJ_AUDIT_OBJECT_CLOSE：表示关闭该对象的时候，是否会产生一个审计事件。

2）句柄的分配与回收

在句柄表的 _HANDLE_TABLE 结构体中，FirstFree 域记录了当前句柄表中空闲的句柄链，这是一个单链表的数据结构，但它并非通过指针链接起来，而是用句柄的 index 值来链接。句柄的 index 值按 _HANDLE_VALUE_INC 逐个递增，HANDLE_VALVE_7NC 是个宏定义，其值为 4。FirstFree 域指示了链表头部的句柄的 index 值，_HANDLE_TABLE_ENTRY 结构体中的 NextFreeTableEntry 指向接下来一个空闲句柄的 index 值。因此，当一个进程在运行过程中需要创建新的句柄时，执行体可以从空闲句柄链表的链首得到一个句柄，新的链表头部变成原来链表头部的 NextFreeTableEntry ；而当释放掉句柄的时候，将等待被释放的句柄的 index 值赋给 FirstFree 域，且将这个句柄项的 NextFreeTableEntry 赋为原来的 FirstFree。另外，_HANDLE_TABLE 结构体的 NextHandleNeedingPool 域记录了下一次句柄表扩展的起始句柄的 index 值，相当于在当前句柄表中所有已分配页面都已经被占用的情况下，下一个页面的起始句柄索引。所以，Windows 进程的句柄表只是简单地线性增长，只有当确实不够用的时候才会增长 [23]。

_HANDLE_TABLE 结构体中有一个标志位叫 StrictFIFO，这个单比特的标志位决定使用怎样的回收算法（如果是后挂上句柄链表的先取下来，这种算法叫做 LIFO，否则就是 FIFO）。

3）漏洞的触发

一个 Windows 控制台的最基本的假设支持是一个单独的进程最多可以被分配一个控制台。这个声明可以在 MSDN 中找到文档，如 AllocConsole() 和 AttachConsole() 引用：

（1）一个进程能够被关联一个控制台，如果进程已经拥有了一个控制台，再调用 AllocConsole() 会失败。

（2）一个进程能够最多附加一个控制台，如果调用的进程已经附加了一个控制台，返回的错误信息是 ERROR_ACCESS_DENIED。

我们来看一下 Kernel32.dll 中 AllocConsole() 的实现 ：

```
text:7C87235E          mov    eax, large fs:18h
text:7C872364          mov    [ebp+var_43C], eax
text:7C87236A          mov    eax, [eax+30h]
text:7C87236D          mov    eax, [eax+10h]
text:7C872370          cmp    [eax+10h], ebx
text:7C872373          jz     short loc_7C872387
text:7C872375          push   5          ; dwErrCode
text:7C872377          call   _SetLastError@4 ; SetLastError(x)
text:7C87237C          mov    [ebp+var_42C], ebx
```

这些汇编代码可以很容易被翻译成 C 语言 ：

```
if(CurrentProcessPEB()->ProcessParameters->ConsoleHandle != NULL)
{
SetLastError(ERROR_ACCESS_DENIED);
return (FALSE);
}
```

很明显，这种情形在应用客户端做了正确的验证。根据这些代码，大量 Kernel32 内部函数的调用被发起，包括 AllocConsoleInternal()，这个函数主要负责发送实际的控制台创建请求到 Windows 子系统。可以通过包装配置数据到特定缓冲区域来实现，同时通过调用 ntdll!CsrClientCallServer() 和 SrvAllocConsole() 操作代码 [24]。

当控制台请求被发送和分发给 CSRSS 后，没有太多的控制，一条 LPC 消息首先被 csrsrv!CsrApiRequestThread() 所接收，然后传递给一个适当的操作处理程序，即 winsrv!SrvAllocConsole()，它从以下汇编代码开始执行 ：

```
.text:764FD1A7 ; int __stdcall SrvAllocConsole(LPHANDLE lpTargetHandle, int)
.text:764FD1A7 __stdcall SrvAllocConsole(x, x) proc near ; DATA XREF: .text:764E8A80o
.text:764FD1A7
.text:764FD1A7 var_C         = byte ptr -0Ch
.text:764FD1A7 var_4         = dword ptr -4
.text:764FD1A7 lpTargetHandle  = dword ptr  8
.text:764FD1A7
.text:764FD1A7             mov    edi, edi
.text:764FD1A9             push   ebp
.text:764FD1AA             mov    ebp, esp
.text:764FD1AC             sub    esp, 0Ch
.text:764FD1AF             push   ebx
.text:764FD1B0             mov    ebx, [ebp+lpTargetHandle]
.text:764FD1B3             push   esi
.text:764FD1B4             push   edi
.text:764FD1B5             lea    esi, [ebx+28h]
.text:764FD1B8             mov    eax, large fs:18h
.text:764FD1BE             mov    eax, [eax+3Ch]
.text:764FD1C1             mov    eax, [eax+20h]
.text:764FD1C4             mov    eax, [eax+74h]
.text:764FD1C7             mov    edi, ds:CsrValidateMessageBuffer(x, x,x,x)
.text:764FD1CD             push   1
.text:764FD1CF             push   dword ptr [esi+4]
```

可以看出，程序从验证输入缓冲区开始，进一步进行控制台的分配，但是这段代码并没有检查是否已经有一个控制台关联到该客户端进程。上面的观察可以得到一个简单的结论 ：只有在请求者的本地环境中，才能对阻止进程请求分配一个以上控制台的情形实施绕过 [25]。

下面的函数能够清除控制台句柄，这样多个 kernel32 !AllocConsole() 调用将不会再失败 :

```
VOID ClearHandle()
{
// fs:[0x18] points to the virtual address
// of the Thread Environment Block (TEB) structure.
__asm("mov eax, fs:[0x18]");
// kd> dt _TEB
// nt!_TEB
//   +0x000 NtTib          : _NT_TIB
//   +0x01c EnvironmentPointer : Ptr32 Void
//   +0x020 ClientId       : _CLIENT_ID
//   +0x028 ActiveRpcHandle  : Ptr32 Void
//   +0x02c ThreadLocalStoragePointer : Ptr32 Void
//   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
__asm("mov eax, [eax+0x30]");
// kd> dt _PEB
// nt!_PEB
//   +0x000 InheritedAddressSpace : UChar
//   +0x001 ReadImageFileExecOptions : UChar
//   +0x002 BeingDebugged    : UChar
//   +0x003 SpareBool       : UChar
//   +0x004 Mutant          : Ptr32 Void
//   +0x008 ImageBaseAddress : Ptr32 Void
//   +0x00c Ldr            : Ptr32 _PEB_LDR_DATA
//    +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
__asm("mov eax, [eax+0x10]");
// kd> dt _RTL_USER_PROCESS_PARAMETERS
// nt!_RTL_USER_PROCESS_PARAMETERS
//   +0x000 MaximumLength    : Uint4B
//   +0x004 Length          : Uint4B
//   +0x008 Flags           : Uint4B
//   +0x00c DebugFlags      : Uint4B
//   +0x010 ConsoleHandle    : Ptr32 Void
__asm("mov dword ptr [eax+0x10], 0");
}
```

鉴于上述过程，我们可以成功执行以下的代码 ：

```
AllocConsole(); // (1)
ClearConsole();
AllocConsole(); // (2)
```

结果使 CSRSS 创建了两个控制台窗口，虽然第一个仍然显示在屏幕上并且有其时间派送，但它现在只是一个僵尸控制台。也就是说，我们不能通过任何方式去引用这个控制台窗口，除非有另外一个进程附加到该控制台上。当然，

我们假设这种情况没有出现，甚至在它的父应用进程终止后，控制台窗口仍然出现在用户的桌面上。这个缺陷包含了一个潜在的攻击去消耗计算机资源（内存），并且保持计算机资源被消耗的状态直到计算机被重新启动[26]。还有一种是一个典型的拒绝服务情形，即运行以下代码片段的结果：

```
while(1)
{
AllocConsole();
ClearHandle();
}
```

可能造成的资源消耗如图 4 所示。

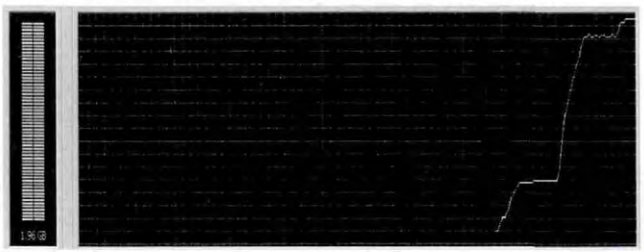

图4 拒绝服务环境资源消耗

这就取得了一个可行的 DoS 环境，然后考虑这个漏洞是否可以进一步操作以获得更多的用处。由于被设计成每个进程最多管理一个控制台，当创建一个内部结构定义时，CSRSS 不能存储除开发人员最初设计以外的更多信息，因此，每个由 Windows 子系统所管理的进程结构只包含一个字段来存储控制台句柄。以上假设可以通过调查 ReactOS（比较精确地复制了 Windows 的操作系统）的源代码进行验证。

基于此，在没有对引用结构造成破坏的前提下，在一个进程的上下文中不可能分配多个控制台。因此，要分配第二个控制台给已经拥有一个控制台的进程，会使 CSRSS 重写之前的分配信息：ProcessData->Console = Console。而上述行为存在的问题是，由于 SrvAllocConsole() 错误地编码，之前的控制台并没有被释放掉。因此，控制台的引用计数并没有减少。通常情况下，只有终止了其中一个控制台客户端，CSRSS 才会减少控制台的引用计数[27]。

这就给我们留下了一个或者多个悬空的控制台，它们本身并不构成非常严重的问题，只是会占用子系统进程的虚拟空间。值得指出的是，这些控制台对象仍然保存着客户端的一些相关信息（如父进程句柄值），从控制台开始分配时就可能已经失效。我们需要考虑的是是否能够通过 CSRSS 来利用这些过期的数据。

为了实现控制台的一些功能，CSRSS 进程有时候会回调控制台窗口的拥有者或者客户端，进一步说，一般在以下两种情况下使用回调：

（1）一个 Control 事件是在控制台环境下产生的。它可以通过使用 GenerateConsole- CtrlEvent API 或者手动敲击 Ctrl+C 或 Ctrl+Break 热键两种方法来实现。

（2）用户想要设置窗口的属性，通过右键点击窗口，选择属性 / 默认值选项。

第一种情况是一个已经被定义好的 API 调用，应用程序本身可以提出自己的回调程序要求控制事件发生。而另外一种情况是一种内部解决方案，没有被任何一个官方文件所指出。为了成功实行回调，子系统使用控制台描述符中的信息，其中最重要的是客户端进程句柄[28]。

CSRSS 客户端回调的触发可以通过 CSRSS 在客户端进程环境下创建一个新的线程实现[29]。

```
.text:75B48504 ; int __stdcall InternalCreateCallbackThread(HANDLE hProcess, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter)
.text:75B48504 _InternalCreateCallbackThread@12 proc near ; CODE XREF: WowExitTask(x)+40 p
.text:75B48504                                         ; CreateCtrlThread(x,x,x,x,x)+10E p ...
.text:75B48504
.text:75B485A5          lea     eax, [ebp+ThreadId]
.text:75B485A8          push    eax             ; lpThreadId
.text:75B485A9          push    ebx             ; dwCreationFlags
.text:75B485AA          push    [ebp+lpParameter] ; lpParameter
.text:75B485AD          lea     eax, [ebp+ThreadAttributes]
.text:75B485B0                  push    [ebp+lpStartAddress] ; lpStartAddress
.text:75B485B3          push    ebx             ; dwStackSize
.text:75B485B4          push    eax             ; lpThreadAttributes
.text:75B485B5          push    [ebp+hProcess]  ; hProcess
.text:75B485B8      call    ds:__imp__CreateRemoteThread@28 ; CreateRemoteThread(x,x,x,x,x,x,x)
```

其中，lpStartAddress 这个参数是被控制台父进程所控制的。我们再调查 InternalCreateCallbackThread() 引出了以下两条回调链[30]：

（1）winsrv!ProcessCtrlEvents → winsrv!CreateCtrlThread → winsrv!InternalCreateCallba-ckThread

（2）winsrv!InitWindowClass → winsrv!ConsoleWindowProc → winsrv!PropertiesDlgSho-w → winsrv!InternalCreateCallbackThread

两条执行路径都可以通过攻击者的应用（调用 GenerateConsoleCtrlEvent() 或者发送消息）轻易触发。因此，一个攻击者可以通过 CSRSS 调用 CreateRemoteThread()，并使用一个被释放的进程句柄和一个被操作的起始地址[31]。

上述结果可以通过创造一个悬挂的控制台，终止父进程，选择属性选项，以及观察下面 CSRSS 的 API 调用得到证实 [32] ：

```
kd> U
winsrv!InternalCreateCallbackThread+0x17:
001b:75b7851b ff151c13b675    call    dword ptr [winsrv!_imp__NtOpenProcessToken (75b6131c)]
001b:75b78521 85c0            test    eax,eax
001b:75b78523 7d07            jge     winsrv!InternalCreateCallbackThread+0x28 (75b7852c)
001b:75b78525 33c0            xor     eax,eax
001b:75b78527 e9c4000000      jmp     winsrv!InternalCreateCallbackThread+0xec (75b785f0)
001b:75b7852c 56              push    esi
001b:75b7852d 8b350813b675    mov     esi,dword ptr [winsrv!_imp__NtQueryInformationToken (75b61308)]
001b:75b78533 57              push    edi
kd> Dd @esp
010bfbe8  0000051c 00000008 010bfbe8 01ebfa78
010bfbf8  010bfbd4 7e428dd9 00010564 000000a4
010bfbc8  010bfedc 009475b8 010bfbec 7e418b26
010bfbd8  009475b8 010bfbf8 75b617f1 00000000
010bfbe8  01ebf6d8 00000000 010bfc24 75b8f412
010bfbf8  0000051c 7c872101 00010564 00000112
010bfc08  01ec01a0 0000fff8 7e44048f 010bfe64
010bfc18  7c872101 00010564 00000000 010bfe74
kd> !handle 51c
processor number 0, process 821d2da0
PROCESS 821d2da0  SessionId: 0  Cid: 025c    Peb: 7ffd8000  ParentCid: 021c
DirBase: 09308040  ObjectTable: e15394b8  HandleCount: 2774.
Image: csrss.exe

Handle table at e1222000 with 2774 Entries in use
051c: free handle, Entry address e17fba38, Next Entry 0000030c
```

鉴于上述情况，我们最终可以在系统进程环境中，在未定义的句柄值上进行关键操作。接下来要研究的是如何利用这个漏洞获得更多的权限。

3.3 漏洞利用

在目前的情况下，我们可以通过 CSRSS 使用一个之前被释放的句柄，假设它是有效的进程描述符。为了能够利用这种情况，我们需要把句柄重新分配给另外一个拥有更高权限的进程。如果我们能够（非直接）控制子系统进程执行的句柄分配和回收的数量、类型和顺序，就可以实现句柄的重新分配 [33]。

准确的句柄释放算法详见 3.2 节的 Windows 句柄的分配与回收。操作系统管理一条简单的 LIFO（后进先出）句柄链表，叫做 freelist。每当一个进程请求一个新的句柄时（通过 OpenProcess:OpenThread，或者其他 API），链表的第一个节点就会出链，分配给请求的对象。同样地，当 Ntclose 服务被一个 ring-3 程序所调用，释放的句柄表就会被放置到链表的最首端 [34]。

当系统创建一个新的进程时，一共有 3 种句柄会按照以下的顺序和位置在 CSRSS 坏境中分配 ：

1）进程对象句柄。通过 NtDuplicateObject 分配到 basesrv!BaseSrvCreateProcess 中。

2）线程对象句柄。通过 NtOpenThread 分配到 basesrv!BaseSrvCreateThread 中。

3）端口对象句柄。通过 NtAcceptConnectPort 分配到 csrsrv!CsrApiHandleConnection- Request 中。

因此，第一个出现在该链表中的节点总是会分配给新创建的进程。另一方面，当一个简单的进程（单线程进程）终止时，句柄的释放按照以下顺序进行 ：

1）线程对象句柄。

2）端口对象句柄。

3）进程对象句柄。

CSRSS 的内部状态是高度依赖于操作系统的状态，任何时候一个进程或者线程的创建或终止，句柄链表都会发生变化。而且，创建一个系统权限的进程，潜在的可能会创建未知数量的其他进程和线程。总之，一个攻击者很难建造一条理想的 freelist，并且精确到其中的一个句柄。为了克服这种困难，可以尝试利用以下技术（我们将它称为句柄染色法）来解决 [35,36]。

这个方法的最基本的思路是，填满一条 freelist，存储大量的进程句柄（与控制台相关），而不是仅仅一个句柄。通过构造这样一个队列，它可能拥有 100 节点，我们并不用在意句柄分配给了哪个拥有系统权限的进程 [37]。

让我们测试一条常规的句柄链表，句柄的组成为以下形式 ：

Freelist → 1 → 2 → 3 → 4 → 5 → …… → 1000 → ⊠

当我们创建多个单线程进程，达到 100 个以后，我们可以得到下面的链表 ：

进程句柄链表→ 1 → 4 → 7 → 10 → …… → 298

线程句柄链表→ 2 → 5 → 8 → 11 → …… → 299

端口句柄链表→ 3 → 6 → 9 → 12 → …… → 300

Freelist → 301 → 302 → 303 → …… → 1000 → ⊠

当所有被创建的进程创建了一个僵尸控制台，我们再全部终止它们。由于线程与端口句柄回收位置的调换，链表将会变成下面的形式 ：

Freelist → 1 → 3 → 2 → 4 → 6 → 5 → …… → 1000 → ⊠

此时，所有之前分配给进程的句柄（1,4,7..）都关联一个悬空的控制台，它们将会在创建远程线程时被用到。现在，我们按照合适的顺序来创建和终止线程，进而操作 freelist 的布局（CSRSS 会保留运行在子系统中所有进程或线程的痕迹，因此它会打开所有执行单元的句柄）[38]。

现在，我们可以创建 300 个线程 ：

线程句柄链表→ 1 → 3 → 2 → 4 → 6 → …… → 300

Freelist → 301 → 302 → 303 → …… → 1000 → ☒

然后，我们释放掉 200 个句柄，得到以下句柄链 ：

线程句柄链表→ 1 → 4 → 7 → …… → 298

Freelist → 2 → 3 → 5 → 6 → …… → 1000 → ☒

最后，我们释放掉线程句柄链表中剩下的句柄，得到最终的 Freelist ：

Freelist → 1 → 4 → 7 → …… → 298 → 2 → 3 → 5 → 6 → …… → 1000 → ☒

由于这样的重新编排，我们最终得到了一条全部可操作的 Freelist, 拥有 100 个与控制台对象相关联的句柄。通过以上步骤，我们不再需要担心拥有系统权限的进程（即被用作被利用对象的进程）是否会被分配到特定的句柄，因为它终将会被分配到。

接下来要考虑可能被利用的媒介，也就是能够获得系统权限的确切途径。最常见的想法是创建一个服务进程，如使用帮助和支持中心，或者其他需要特定服务才能工作的系统功能 [39]。

经过几个简单的实验，可以发现一个简单的解决方法。Windows XP 中，在系统回话期间使用热键 WIN+U 可以令 winlogon.exe 创建几个进程，其中一个就是 utilman.exe，它是拥有系统权限的。

utilman.exe 即辅助工具管理器，负责快捷访问选项的管理，它可以对一些不常见的命令进行一些提示和帮助。更具体地说，它可以打开一些辅助应用，如放大器、屏幕键盘等。关键在于，它在登录屏幕 (登录提示) 激活时能够被使用，因此它是属于拥有高级别权限的进程。

虽然 utilman.exe 功能还可以通过创建操作系统中已经存在的拥有高级别权限的进程实现，但是为了获得一个理想的利用环境，选择使用 UTULMAN 进程。

在成功提权前我们需要完成利用的最后一步，在 UTULMAN 的内存空间里写入特定的有效代码。很明显，我们不能通过调用 OpenProcess() 和 WriteProcessMemory API 来实现。进程运行在普通用户账户下不能打开系统权限进程的句柄，也不能通过环境变量偷偷运行一些可执行的代码。由于权限等级的不同，不可能通过通信的方式传输数据 [40]。

WIN32K.SYS 是 Windows 图形化最重要的内核模块，WIN32K.SYS 模块映射到所有 GUI 进程的上下文中，一个进程变为图形化必须要调用 WIN32K.SYS 中的一个系统调用函数。该模块还管理着两块共享区域，其中一块共享区域包含了桌面窗口的一些属性，包括一些像窗口 unicode 数据。因此，恶意程序只是通过在本地桌面创建和操作普通窗口，便可以将任意代码存储在具有系统权限的进程内存空间中 [41]。

为了解决 ASRL（地址空间分布随机化）的问题，可以使用一个简单的内存覆盖技术，通过创建多个超长标题的窗口。通过这个方法，我们可以设置远程线程的 StartAddress 为一个特定值，并且做一个简单的假设，共享区域映射将足够的大能够覆盖被选择的内存区域。实验后发现，40 个 32K 长标题的窗口能够保证有 100% 的成功率。分析后发现，Windows XP 的共享区域不仅是可读的，而且是可以执行的，这个能够让我们成功地绕过一些 Windows DEP 机制的潜在威胁。

综上所述，我们可以将利用过程总结如下 ：

1) 通过创建足够多的用户对象填充 WIN32K 共享区域，本地桌面运行的所有进程将被映射到这个区域。

2) 创建 N 个进程实例，所有进程将会创建一个僵尸控制台，然后被闲置。

3) 释放 N 个进程实例。

4) 创建 3N 个本地线程。

5) 按照之前的顺序释放 2N 个线程。

6) 释放掉剩下的 N 个线程。

7) 仿真 win + u 按键 , 导致创建一个 utilman.exe 的新实例。

8) 调用 SendMessage(HWND_BROADCAST,WM_SYSCOMMAND,0xFFF7,0)，触发 N 个释放的句柄上的 CreateRemoteThread 的执行。

3.4 漏洞总结

通过对 CVE-2011-1281 漏洞的分析，发现 use-after-free 漏洞不仅出现在浏览器漏洞中，在系统软件中同样有可能出现。

CVE-2011-1281 并不是针对于内存的，而是针对于句柄的，这在之前公开的 use-after-free 漏洞中是不曾出现的，具有一定的研究价值。此外该漏洞的分析与利用是建立在对 CSRSS 组件的各项分析之下。它是一个本地拒绝服务漏洞，也是一个权限提升漏洞。多个控制台对象与一个进程相关联时出现的内存损坏情形导致了此漏洞。 当进程退出

时，额外的控制台对象被孤立，此对象句柄可以由比预期拥有较高特权的另一进程所获取[42]。

4 结束语

本文虽然在基于 Windows 平台的 CSRSS 进程的分析以及相关漏洞研究上取得了不错的实际成果，分析了 CSRSS 本地权限提升漏洞与 CSRSS 本地拒绝服务漏洞，但是还有许多问题有待于进一步的研究，主要有：

1）对 CSRSS 进程的分析不够全面，在进程与线程的列表管理以及 DOS 虚拟机仿真、图形服务等方面没有做出分析[43]。

2）在漏洞研究方面，主要选择的是 Windows XP 平台下的漏洞，没有对近期 Window 7、Windows 8 最新暴露出的 CSRSS 漏洞进行研究。

3）漏洞研究主要停留在分析与利用的阶段，不具备漏洞挖掘的能力，需要有更多的经验，学习更多的知识归纳和总结才有可能有所突破。（责编 马珂）

参考文献

[1] 文伟平，吴兴丽，蒋建春 . 软件安全漏洞挖掘的研究思路及发展趋势 [J]. 信息网络安全，2009，（10）：78-80.
[2] 张银奎．如何调试 Windows 子系统的服务器进程 [M]．北京 ：电子工业出版社，2008.
[3] 彭赓 .Windows 平台下软件安全漏洞研究 [D]. 成都 ：电子科技大学，2010.
[4] 毛德操 . 漫谈兼容内核 [M]. 北京 ：电子工业出版社，2009.
[5] 徐有福，文伟平，尹亮 . 基于补丁引发新漏洞的防攻击方法研究 [J]. 信息网络安全，2011，（07）：45-48.
[6]CVE-2011-1281 [EB/OL]. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1281.
[7]CVE-2011-0030 [EB/OL]. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0030.
[8] Custom console hosts on Windows 7 (Hack in the Box Magazine #4), [EB/OL]. http://magazine.hitb.org/issues/HITB-Ezine-Issue-004.pdf.
[9] Windows CSRSS Tips & Tricks (Hack in the Box Magazine #5), [EB/OL]. http://magazine.hitb.org/issues/HITB-Ezine-Issue-005.pdf.
[10] Windows 7 / Windows Server 2008 R2: Console Host [EB/OL]. http://blogs.technet.com/b/askperf/archive/2009/10/05/windows-7-windows-server-2008-r2-console-host.aspx.
[11] Windows Numeric Handle Allocation in Depth (Hack in the Box Magazine #6) [EB/OL]. http://magazine.hackinthebox.org/issues/HITB-Ezine-Issue-006.pdf.
[12] Windows Handle Lister project @ Google Code [EB/OL]. http://code.google.com/p/windows-handle-lister/.
[13] ntdebug, LPC (Local procedure calls) Part 1 architecture [EB/OL]. http://blogs.msdn.com/b/ntdebugging/archive/2007/07/26/lpclocalprocedure-calls-part-1-architecture.aspx.
[14]J. Goebel , T. Holz. Rishi: Identify bot contaminated hosts by ircnickname evaluation[C]. Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets, USENIX Association, 2007, pp.8-8.
[15] Haroon Meer. The Complete History of Memory Corruption Attacks[C]. BlackHat Confidence USA, 2010.
[16] J.Goebel, T. Holz. Rishi: Identify bot contaminated hosts by ircnickname evaluation[C]. Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets. USENIX Association, 2007, pp.8-8.
[17] MSDN. Console Functions [EB/OL]. http://msdn.microsoft.com/en-us/library/ms682073%28VS.85%29.aspx.
[18] Gynvael Coldwind. Enter teh ANSI Escape Code suport for internal cmd.exe commands and BAT scripts [EB/OL]. http://gynvael.coldwind.pl/?id=130.
[19] Nynaeve. The system call dispatcher on x86 [EB/OL]. http://www.nynaeve.net/?p=48.
[20] Pourabbas E,Rafanelli M.Hierarchies and Relative Operators in the OLAPEnvironment [C]. SIGMOD Record, 2000,29(1):32-37.
[21] HMM Rabiner LR. A tutorial on hidden Markov models and selected applications in speech recognition[J]. Proceedings of the IEEE, 1989, 77(2):257-286.
[22] 王颖 . Fuzzing 漏洞挖掘与溢出利用分析技术研究 [D]. 郑州 ：解放军信息工程大学，2009.
[23] 倪凯斌 . 基于 Fuzzing 的 IE 浏览器控件安全漏洞发掘技术研究 [D]. 广州 ：暨南大学，2010.
[24] 贺拓 . Flash 应用程序漏洞挖掘与利用 [D]. 西安 ：西安电子科技大学，2010.
[25] 郑亮，谢小权 .Fuzzing 漏洞挖掘技术分析 [C]. 第 24 次全国计算机安全学术交流会，2009.
[26] 毛德操 . Windows 内核情景分析 [M]. 北京 ：电子工业出版社，2009.
[27] 张佩，马勇，董鉴源 . 竹林蹊径 : 深入浅出 Windows 驱动开发 [M]. 北京 ：电子工业出版社，2011.
[28] 张帆，史彩成 .Windows 驱动开发技术详解 [M]. 北京 ：电子工业出版社，2008.
[29] Gary Nebbett. Windows NT/2000 Native API Reference[M].Indiana: Sams, 2000.
[30] 张银奎．软件调试 [M]．北京 ：电子工业出版社，2008.
[31]Ting Liu, Xiaohong Guan, Qinghua Zheng, et al. Prototype Demonstration: Trojan Detection and Defense System[C].In: Consumer Communications and Networking Conference, 2009. CCNC 2009:1-2.
[32] Kip R. Irvine. Assembly Language for Intel-Based Computers[M]. New Jersey: Prentice Hall,2006.
[33]j00ru.Kernel exploitation-r0 to r3 transitions via KeUserModeCallback[EB/OL].http://j00ru.vexillium.org/?p=614,2010-9-5.
[34] 微软安全技术中心 .Windows 内核模式驱动程序中的漏洞可能允许特权提升 [EB/OL]. https://technet.microsoft.com/library/security/ms11-054,2011-07-12.
[35] 中国互联网信息中心 . 第 29 次中国互联网络发展状况统计报告 [EB/OL]. http://www.cnnic.net.cn/hlwfzyj/hlwxzbg/hlwtjbg/201206/t20120612_26720.htm,2012-01-16.
[36] 王清，张东辉，周浩，等 . 0day 安全 : 软件漏洞分析技术 (第 2 版)[M]. 北京 ：电子工业出版社，2011.
[37] Bill Blunden,The Rootkit Arsenal [M]. Massachusetts: Jones & Bartlett Publishers,2009.
[38] Bisbey,R., D.Hollingsworth,Protection Analysis Project Final Report[R].Information Sciences Institute,University of Southern California,Marina Del Rey,CA,1978.
[39]C.E.Landwehr, A.R.Bull, J.P.McDermott,ea al.A taxonomy of computer program security flaws[J].ACM Computing Surveys, Vol.26(3),pp.211-254，1994.
[40] 王铁磊 . 面向二进制应用程序的漏洞挖掘关键技术研究 [D]. 北京 : 北京大学 2010.
[41] 彭赓 .Windows 平台下软件安全漏洞研究 [D]. 成都 ：电子科技大学，2010.
[42] 谭文，邵坚磊 . 天书夜读 : 从汇编语言到 Windows 内核编程 [M]. 北京 ：电子工业出版社，2008.
[43] Greg Hoglund，Jamie Butler. Rootkits-Subverting the Windows Kernel [M]. Boston:Addison-Wesley Professional,2005.