文章编号:1001 - 9081(2006)09 - 2134 - 03

利用核心态钩挂技术防止代码注入攻击

朱若磊
(广东商学院 信息学院,广东 广州 510320)
(hg-hgzzz@sohu.com)

摘 要:为防止代码注入攻击,利用钩挂技术来监视有关的 API 函数调用十分必要。由于 Windows NT 系统中存在着严格的进程隔离机制,此种钩挂要在核心态下才有效。提出并讨论了实现此种技术的一种简便的方法。实践表明,在 Windows XP 系统条件下,利用它能够成功阻止木马利用代码注入实现攻击。

关键词:代码注入; 钩挂; 核心态
中图分类号: TP309.5 **文献标识码**:A

Preventing code injection attack with hook in kernel mode

ZHU Ruo-lei
(*School of Information Science and Technology, Guangdong University of Business Studies, Guangzhou Guangdong* 510320, *China*)

Abstract: To prevent code injection attack, it is necessary to monitor involved API(Application Programming Interface) by hooking them. Because there exists rigid process isolation in Windows NT, hooking these APIs must be done in kernel mode. A relatively simple way to do this was introduced. It is proved that in Windows XP the way to hook API in kernel mode can efficiently prevent code injection attack.

Key words: code injection; hook; kernel mode

0 引言

为躲避系统的监控,一些木马病毒在攻击系统时往往将自身作为一个线程运行于其他应用程序的地址空间内(代码注入)。这样一来,通过系统进程列表根本看不出任何异常,木马进程也就在系统中彻底"消失"了,增加了查杀的难度。

据此,本文提出了一种在 Windows NT 系统中,在核心态下利用 Hook 技术来监控有关 API 函数的策略来防止代码注入的策略,并详细讨论了实现此策略的几个关键技术。

1 代码注入原理

理论上,在 Windows NT 系统中每个进程都有自己的私有地址空间,互相隔离,别的进程是不允许对这个空间进行操作的。但实际上,在 Windows NT 系统中,有三种机制可向一个远程进程注入代码并获得执行[1]。它们的关键是分别使用了两个 API 函数:SetWindowsHookEx 和 CreateRemoteThread(这两个函数分别存在于系统的 User32.dll 及 Kernel32.dll 中)。前一个函数可通过系统把包含了钩子过程的 DLL 映射到被挂钩的线程的地址空间(远程进程),从而进入此远程进程的地址空间;而后一个函数则允许一个进程在另外一个进程内创建新线程,被创建的新线程与主线程共享同一个地址空间。这样一来,系统就提供了利用 API 函数进入远程进程地址空间的可能。

另一方面,为使自己"消失"成功,还需选定一个目标,即进入哪个远程进程的地址空间。由于木马在设计时并不知道将要攻击的系统中会有哪些进程在运行,因此那些在系统运行中必不可少的进程就会首当其冲成为目标。如在 Windows 系统中,文件管理器 Explorer.exe 进程任何时刻都运行,所以,它经常被用来作为木马病毒程序的"宿主"。

2 解决的策略

通过以上分析,可以看出,要想阻止木马的隐藏,就必须阻断其隐藏的途径。很显然,针对木马隐藏的技术,利用 Hook 钩挂有关实现隐藏的 API 函数,并实时地监视其活动是一个很好的解决策略。

然而,在 Windows NT 系统中(NT/2K/XP),DLL 所在页面被映射进进程的私有空间,且具有写时拷贝属性(COW),即没有进程试图写入该页面时,所有进程共享这个页面;而当一个进程试图写入该页面时,系统的页面错误处理代码将收到处理器的异常,并检查到该异常并非访问违例,同时分配给引发异常的进程一个新页面,并拷贝原页面内容于其上且更新进程的页表以指向新分配的页[2,3]。这样一来,尽管 NT 系统很好地实现了进程间的隔离,但对于想利用 Hook 技术来钩挂处于 User32.dll 或 Kernel32.dll 中的函数并进而监视其活动的想法来说,却是十分不利的。因为,在此过程中,任何对于 DLL 代码中的修改都会触发 COW 机制,从而使这种修改的作用范围仅仅限制于本任务进程范围之内。也就是说,木马病毒在它自己的进程内调用的是映射到它自己地址空间内的 API 函数,此进程外对此 API 函数的任何修改影响不到本进程内部。

事实上,有一种方法可以避免触发系统的 COW 机制,它就是修改 CR0 寄存器的 WP 位。然而,修改 CR0 寄存器的指令需要运行于核心态(Ring0 级)下,才能有这个权限。因此,真正钩挂系统 API 函数的工作必须在 Ring0 级来完成。

综上所述,为了有效地防止木马病毒潜入到别的进程空间当中,就必须在系统范围内监视对 SetWindowsHookEx 和

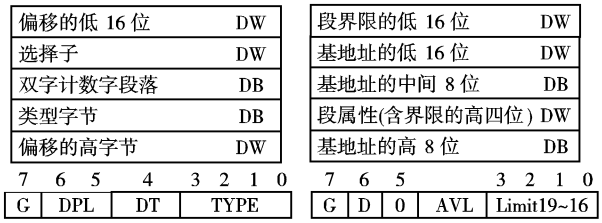
收稿日期:2006 - 03 - 16; **修订日期**:2006 - 05 - 29
作者简介:朱若磊(1973 -),男,江西南昌人,讲师,博士,主要研究方向:计算机网络、系统安全、图像处理、智能控制.

CreateRemoteThread 两个 API 函数的调用,而对这两个 API 函数的钩挂过程必须在核心态下完成。

3　关键技术的原理及实现

3.1　进入核心态

由上述,如何进入 Ring0 级是实现在系统级钩挂 API 函数的关键。本文利用了创建自己的调用门(CallGate)来实现在任务内不同特权级(Windows 中,主要是 Ring0 和 Ring3 级)的变换。



(a) 调用门格式　　(b) 代码段格式

图 1　调用门及代码段的格式

在一个任务内,要实现代码的权限从用户态(Ring3 级)向核心态(Ring0 级)的转换,一般的途径是:使用段间调用指令 CALL,通过调用门进行转移;而从核心态返回到用户态则可利用段间返加指令 RETF 来实现。

图 1(a)上为调用门的结构,可以看出整个调用门由 8 个字节组成,调用门内的选择子必须指向代码段的描述符,而偏移字节指明了在被调用代码段中的偏移。图 1(a)下为类型字节各位的意义。其中:P 位表示该描述符所描述的段是否存在;DPL 表示所描述段的特权级;DT 位说明描述符类型(系统段及门描述符为 0,数据及代码段为 1);依据 DT 位的值,TYPE 类型段的取值如表 1、表 2 所示。本文程序中此段值取为 C(1100)。综上述,此调用门的类型值为 ec。

表 1　系统段及门描述符 TYPE 字段的编码及含义

类型编码	说明	类型编码	说明
0	未定义	8	未定义
1	可用 286TSS	9	可用 386TSS
2	LDT	A	未定义
3	忙的 286TSS	B	忙的 386TSS
4	286 调用门	C	386 调用门
5	任务门	D	未定义
6	286 中断门	E	386 中断门
7	286 陷阱门	F	386 陷阱门

表 2　数据段及代码段 TYPE 字段的编码及含义

类型编码	说明	类型编码	说明
0	只读	8	只执行
1	只读、已访问	9	只执行、已访问
2	读/写	A	执行/读
3	读/写、已访问	B	执行/读、已访问
4	只读、向低扩展	C	只执行、一致码段
5	只读、向低扩展、已访问	D	只执行、一致码段、已访问
6	读/写、向低扩展	E	执行/读、一致码段
7	读/写、向低扩展、已访问	F	执行/读、一致码段、已访问

为了完成特权级的转换,还必须设置一个 DPL 值为 0 的代码段。图 1(b)上为代码段的格式;下为段属性的高位字节各位的意义。其中 G 代表段界限的粒度;在可执行段中 D 位代表指令使用地址及操作数默认大小;AVL 位是软件可利用位。段属性字段的低字节的各位意义如图 1(a)下所示。由于被调用的代码需要运行于核心态中以便于修改 CR0 寄存器,因此代码段被设为非一致性的代码。本文中这个字被设置为 cf9a。

由于处理器采用与访问数据段相同的方式控制对门描述符的调用,所以当用户态程序 CALL 调用门的时候,首先检查当前特权级(CPL)与调用门特权级(DPL)间的关系,只有 CPL≤DPL 时才允许进行此种调用。

当取出调用门内的 48 位全指针作为目标地址时,若 RPL≤DPL 且 CPL > DPL(调用门内选择子所指示代码段的特权级),则在转移到非一致代码段的目标地址过程中发生向高特权级(Ring0)的切换。而此过程中在装载 CS 高速缓冲寄存器前系统会自动调整所使用选择子的 RPL 为零,因此 RPL≤DPL 总会被满足。

以下两条语句分别为设置本文所述的调用门(buff0)及其调用门中所指示的目标代码段的格式(buff1)。

BYTE buff0[ ] = {0x00, 0x00, 0x10, 0x01, 0x00, 0xec, 0x00, 0x00};

BYTE buff1[ ] = {0xff, 0xff, 0x00, 0x00, 0x00, 0x9a, 0xcf, 0x00};

决定了这两个参数后,接下来就需要将其写入到系统的 GDT 或当前任务的 LDT 中去以备调用。然而,在 NT 的保护机制之下,这两个表在用户态下无法操作,否则此种保护也就名不副实了(容易产生象 CIH 作用于 Windows98 的问题)。那么如何才能把自己的调用门写入到系统中去呢? 在初始引导加载器阶段中,NTLDR 负责把计算机的微处理器从实模式转换为平面内存模式[3],而在工作模式转换的时候最重要的一项工作就是要创建系统的 GDT 表来管理平面模式的内存,因此可以利用修改此文件的方法来使系统初始化时就将自己设定的内容写入 GDT 当中。

很显然,要修改 NTLDR 文件就必须知道修改的位置,即自己创建的调用门及代码段描述符应该写在什么地方。为了解决这个问题,首先要知道在此文件中 GDT 的内容放在什么地方。为了找到这个位置,只需在系统中先安装一个系统的调试工具(如 SoftICE),然后调用工具来看 GDT 表项的内容,如在 SoftICE 中,利用 GDT 命令可以看到 GDT 表中的前两项如下所示:

Sel.	Type	Base	Limit	DPL	Attributes
0008	Code32	00000000	ffffffff	0	P RE
0010	Date32	00000000	ffffffff	0	P RW

由此不难得出这两个段的描述符内容,在此分别为:{0xff, 0xff, 0x00, 0x00, 0x00, 0x9a, 0xcf, 0x00} 和 {0xff, 0xff, 0x00, 0x00, 0x00, 0x92, 0xcf, 0x00},根据选择子的内容,可以知道这就是 GDT 表的第一、二项(除掉低三位)。这个内容就可以指示程序来寻找 GDT 的表项位置,在这个表项的后面找到空表项的位置将自己设计的两个描述符填入就可以了。注意,选择子的值同填入表项的位置有关,本文的调用门描述符中填入的选择子是 0x110,因此它指的是离第一个表项位置偏移量为 0x21 * 8 位置的代码段。而在本文中所创建的调用门描述符将填在离第一个表项位置偏移量为 0x20 * 8的位置。

至此,调用门中转移目的地的偏移还没有填入。也就是当程序通过 CALL 调用门后程序跳转的目的地址的偏移量还没有写入。由于 GDT 表不能够在运行时实时地修改,这里就面临一个问题,如何保证自己的程序能够在调用门中填入的

地址上等待被调用? 在此,为了解决这个问题,程序使用了NT 系统的一个特性,即,kernel32. dll 和 user32. dll 总是被系统映射到相同的内存地址[3]。也就是说,这两个 DLL 中的函数在每个任务空间中的地址总是相同的。因此本文程序在这些地址范围中去寻找一个肯定会存在的语句 ret(DLL 中由许多函数组成,每个函数结束必定会有返回语句)。然后,将它的地址填入到调用门中。这样一来,不仅保证了在任务中每次 CALL 调用门时都能够找到所要执行的语句,而且,更重要的是,当执行 ret 语句时,由 CALL 语句所引起的已入栈的当前地址会被弹出,因此,从 CALL 语句后直到段间返回 retf 语句间的程序段就会运行于核心态中。

3.2 钩挂 Windows API 函数

进入核心态之后,除了修改 CR0,取消 COW 机制之外,主要的工作就是要对有关的系统 DLL 函数安装 Hook。在本例中主要是要监视 SetWindowsHookEx 和 CreateRemoteThread 两个系统 API 函数。下面以对 CreateRemoteThread 函数的钩挂为例说明此技术的实现。

一直以来,有两种钩挂 API 函数的方案,第一,找到 API 函数入口点并修改最前面的几个字节为跳转语句到新的代码上,然后视情况返回到原函数代码中; 第二,通过修改模块的引入或引出表来改变 API 函数的入口地址到新的代码段中。在抢先式多任务的 32 位 Windows 系统中,使有后一种方法比较安全[5]。

为了改变 API 函数入口地址,就必须找到其入口地址所在的位置。作为 API 函数的 CreateRemoteThread 存在于 kernel32. dll 中,而动态链接库是通过一组输出函数地址表向系统提供服务的。因此,搞清楚 DLL 的结构是实现钩挂函数的关键。

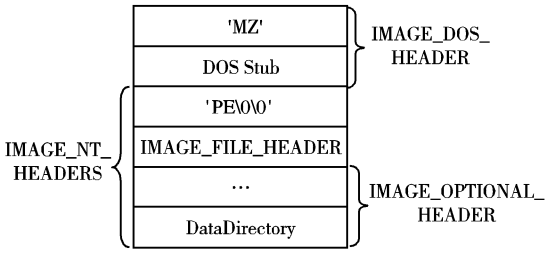


图 2 PE 文件头格式

WindowsNT 系统中,动态链接库文件是作为 PE 文件格式进行存储的。如图 2 所示,PE 文件的文件头中主要由图中所标注的三部分内容组成。其中 IMAGE_FILE_HEADER 结构的最后一个域说明了此模块是可执行文件还是动态链接库文件。图 2 中所示的 IMAGE_OPTIONAL_HEADER 结构的最后一个域 DataDirectory 是一个成员为 IMAGE_DATA_DIRECTORY 结构的结构数组,共有 16 个成员。而 MAGE_DATA_DIRECTORY 结构有两个成员组成,其一为所指示特定数据的 RVA(相对虚拟地址),其二为此特定数据的大小信息。在此结构数组中第一个成员指向的是此模块的函数引出表。

函数引出表结构中,有三个域同最终找到具体的引出函数密切相关,它们分别是:

1) AddressOfNames 域,此域位于引出表相对偏移 20H 处,它指向一组 DWORD 类型的数组,其中每个 DWORD 代表了一个 ASCIIZ 字符串的 RVA,这个字符串就是此模块所提供函数的函数名,遍历此数组可得到此模块的引出的所有函数的函数名; 2) AddressOfNameOrdinals 域,此域位于引出相对偏移 24H 处,它也指向一组 DWORD 类型的数组,该数组的每项与引出表的 Base 域的值相减,就得到对应上个域的相同偏移位置的那个函数的函数地址表的索引值; 3) AddressOfFunctions 域,此域位于引出表相对偏移 1CH 处,它同样也指向一组 DWORD 类型的数组,它的每项对应的就是索引值所代表函数的引出地址[6]。

需要特别说明的是,上述很多域使用了相对虚拟地址(RVA),而在实际使用中模块的基地址可用 GetModuleHandleA 函数得到。

4 结语

通过对上述关键技术的使用,本文实例在 XP 系统下成功的钩挂到了 kernel32. dll 所提供的 CreateRemoteThread 函数,并对其进行了屏蔽处理。使进程无法直接利用此函数启动远程线程,实现代码注入。

参考文献:

[1] ROBERT KUSTER. Three ways to inject your code. into another process[EB/OL]. http://www. codeproject. com/threads/winspy. asp.

[2] SUNWEAR. MGF 病毒技术分析[EB/OL]. http://blog. csdn. net/sunwear.

[3] RICHTER J. Windows 核心编程[M]. 北京 机械工业出版社, 2000. 299 - 309.

[4] 杨季文. 80X86 汇编语言程序设计教程[M]. 北京 清华大学出版社, 2000. 361 - 421.

[5] 冉光志, 陈旭春, 练锴, 等. Visual C++ 应用技巧与常见问题[M]. 北京: 机械工业出版社, 2003. 182 - 189.

[6] LUEVELSMEYER PE 文件格式 [EB/OL]. http://www. pediy. com/tutorial/chap8/Chap8-1-6.

(上接第 2133 页)

[2] BROWN M, HANKERSON D, LOPEZ J, *et al*. Software Implementation of the NIST Elliptic Curves Over Prime Fields[A]. Proc. of CT-RSA'2001[C]. Springer-Verlag, LNCS 2020, 2001. 250 - 265.

[3] HASAN MA, WANG MZ, BHARGAVA VK. A Modified Massey-Omura Parallel Multiplier for a Class of Finite Fields[J]. IEEE Transactions on Computers, 1993, 42(11): 1278 - 1280.

[4] ANSI X9. 62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)[S], 1999.

[5] LÓPEZ J, DAHAB R. Fast multiplication on elliptic curves over GF(2m) without precomputation [A]. CHES'99 Workshop on Cryptographic Hardware and Embedded Systems[C]. Lecture Notes in Computer Science 1717. Springer-Verlag, 1999.

[6] ITOH T, TSUFII S. A fast algorithm for computing multiplicative inverse in GF(2m) using normal bases[J]. Information and Computation, 71998, 8(3): 171 - 177.

[7] IEEE P1363/D13, Standard Specifications for Public Key Cryptography[S], 1999. 93 - 94.

[8] REYHANI - MASOLEH A, HASAN MA. Low Complexity Word -Level Sequential Normal Basis Multipliers[J]. IEEE transactions on computers, 2005, 54(2): 98 - 110.

[9] JARVINEN K, TOMMISKA M, SKYTTA J. A scalable architecture for elliptic curve point multiplication[A]. 2004 IEEE International Conference on Field-Programmable Technology, ICFPT 2004[C], 2004. 303 - 306.