透视黑客技术发展焦点，把握黑客攻防技术跳动脉搏，全面收录流行黑客技术

黑客防线

《黑客防线》编辑部 编

2011

精华奉献本 上册

- 黑客编程实战大演练
- 黑器免杀与入侵进阶
- 加密与破解经典实例
- 网络安全与加固精讲

2CD-ROM

人民邮电出版社
POSTS & TELECOM PRESS

黑客防线

2011

精华奉献本 上册

《黑客防线》编辑部 编

人民邮电出版社
北京

图书在版编目（CIP）数据

黑客防线2011精华奉献本 / 《黑客防线》编辑部编
. -- 北京 : 人民邮电出版社, 2011.3
ISBN 978-7-115-24795-7

Ⅰ. ①黑… Ⅱ. ①黑… Ⅲ. ①计算机网络－安全技术
Ⅳ. ①TP393.08

中国版本图书馆CIP数据核字(2011)第018576号

内 容 提 要

《黑客防线》是国内最早创刊的网络安全技术媒体之一。本书收录了《黑客防线》总第109期至第120期的精华文章。

《黑客防线》一直秉承“在攻与防的对立统一中寻求突破”的核心理念,关注网络安全技术的相关发展,并一直跟踪国内网络安全技术的发展,经过2001年创刊至今,已经成为国内网络安全技术的顶尖媒体。本书选取了编程解析、工具与免杀、网络安全顾问、密界寻踪、首发漏洞、特别专题、漏洞攻防、脚本攻防、溢出研究以及渗透与提权等方面的精华文章,并配有两张CD光盘,其中包含安全技术工具、代码和录像,为读者阅读、理解提供了非常便捷的途径。

本书分为上、下两册,适合高校在校生、网络管理员、网络安全公司从业人员、黑客技术爱好者阅读。

黑客防线2011精华奉献本（上下册）

◆ 编　　《黑客防线》编辑部
责任编辑　张　涛
◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市海波印务有限公司印刷
◆ 开本：787×1092　1/16
印张：38.25
字数：1 105千字　2011年3月第1版
印数：1 - 8 000册　2011年3月河北第1次印刷

ISBN 978-7-115-24795-7

定价：49.80元（附2张光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第0021号

前　言

2010年匆匆过去了。对于网络安全的相关技术而言，这一年其实很不平凡。在这一年里，纷乱的网络环境得到治理。在技术层面上，抑制了浮躁，同时也引发了深入研究底层核心技术的思考。这一年，恰逢《黑客防线》创刊十周年。十年磨一剑，作为一本技术月刊，基本上守住了纯技术这样一块净土，坚持了核心技术成就未来的理念，同时在方法上，一直倡导在攻与防的对立统一中寻求突破的方法论，因而过去的一年也可以说是本刊丰收的一年。

这本精华奉献本，就是2010年全年技术月刊精华文章的集合。

需要说明的是：由于多种原因，技术月刊从去年第7期开始停止了纸张版本的出版，只有网站提供的电子版半月刊，所以，这本精华本就更显得弥足珍贵。特别是对于没有按期订阅电子版月刊的读者，更需要购买的就是2011精华本了。

如果需要及时跟踪和捕捉新的技术，还是希望读者随时到黑客防线网站订阅电子版半月刊。

本书附带的光盘，是为了方便读者进行源程序测试。精华本中的每篇文章基本上都提供了源程序。如果光盘中找不到某一篇文章涉及的源程序，那就是当时作者投稿时未提供，读者只要研究领会文章所述的技术要义，然后自己完成程序并不是很难。

在这里，要感谢十年来坚持阅读《黑客防线》并且一直践行技术创新和突破的读者朋友们，更要感谢作者朋友们一直进行着艰苦卓绝的技术研究和探索，不断将新技术突破，并整理成文章与大家分享。

在此，吁请具备一定技术能力的读者，积极参与到作者队伍中来，我们的作者群其实就是一支虚拟的技术团队——黑客防线技术团队，这是网络信息安全方面的一支精英团队。这个团队希望不断有新生力量参与进来。

让我们从现在做起、从我做起，一起关注网络安全技术的快速发展势头，抓住网络安全技术的新亮点，一起开创中国网络安全技术的新篇章！

郑重声明：本书所讲述的内容仅供学习参考，切勿用于非法用途，由此引起的后果自己负责，出版社不负任何责任。通过阅读本书，希望读者能树立良好的网络安全意识，提高网络安全防御水平。同时提醒读者，应在受控的环境（比如单独用来作为测试的计算机）里分析、使用书中的代码，切勿在企业的业务或生产网络中使用这些程序，以免造成不必要的损失。

《黑客防线》编辑部

2011年3月

《黑客防线》2011精华奉献本目录(上册)

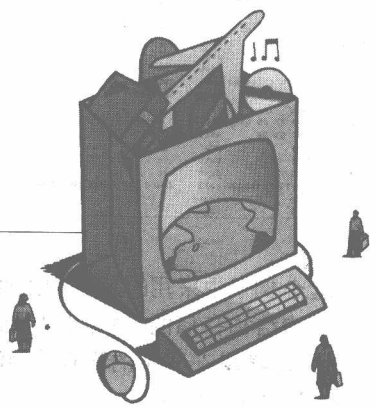
编程解析

工具与免杀

网络安全顾问

前置知识:汇编
关键词:编程、数据编码、反查杀

AV对抗技术之数据编码



文/图 xfish

时间匆匆流逝,在安全体系越来越完善的今天,AV(Anti Virus,反病毒)对抗技术也越来越显得十分重要。因为没有它,木马、病毒、蠕虫等将一无所用。我举个例子,比如自己编写的木马很可能在第一次完成编译后,就面临被大部分具有虚拟机启发或DNA查杀的AV所杀。其次,最重要的是还要考虑到木马、病毒等后期如果一旦被AV所杀,如何能快速地针对大面积的AV进行处理,这是我们所要思索的问题,否则很难持续在这个方面走下去。

本文讲述的是AV对抗技术之数据编码,在后续的文章中还会和大家探讨AV对抗技术的其余方面。

初看到数据编码这个词,大家可能会有很多的联想。其实这里的数据编码指的是将我们的数据信息进行重新编码,让其按照自己的格式来编译。如果以前写过木马、病毒、exploit等部分敏感数据信息,大家肯定都进行加密处理。但是我如果让你把所有的敏感数据信息(也包括局部数据)都进行加密处理,且解密头或密钥是随机的,那么我想,你肯定会觉得这是一个非常庞大的工程。目前的大部分AV在启发技术中都包含了基因码检测技术,也就是俗称的DNA检测技术。基因码检测技术最具代表性的产品就是我们非常熟悉的"大蜘蛛"、"小红伞"等,其中我在实践中对"大蜘蛛"一直是无视的,"小红伞"则相对来说要麻烦些。基因码检测技术有时候十分变态,比如它会分析你的程序文件结构,通过文件结构来分析引入函数,其次则是对程序中的数据信息与库中的特征进行匹配,有时候会根据重定位来分析引用数据信息所在的代码部分是否存在一些解密代码等。本文涉及的数据编码技术就是让我们的木马、病毒等能高效、方便地对抗AV的数据信息匹配技术。

编写木马时,如果大部分函数中的参数都是AV敏感的数据信息,我们该如何做?举个例子,要动态获取CreateRemoteThread函数的地址,肯定不能把这个函数名称真实暴露在程序中,通常一般的病毒做法是通过hash算法计算出这些函数名称的hash值,根据hash值,通过自己编写的GetPorcAddress函数来获取地址。这样做的方法在木马编写中并不实用,因为如果一旦数据过多且是其他各种各样函数的参数,处理起来会非常困难,而数据编码技术就是要解决这个问题。下面我们就用字符串数据来做例子进行讲解,也就是通过修改原有的字符串压入参数的结构,使数据并不是以字符串结构来存放。通常大部分病毒所采用的压入的字符串形式结构有如下两种:

```
//第一种方法
call   s
db 'xfish',0
s:
//第二种方法
push   'h'
push   'xfish'
mov    edx,esp
```

不过以上都不是很好的方法,因为数据信息还是会暴露。如果在编译期间我们的字符串不是以明文存放,那该多好,并且每一步还包含一个解密的代码。接着我们再看一些例子,如图1所示。

```
00401000 >/$  55              push    ebp
00401001  |.  89E5            mov     ebp, esp
00401003  |.  E8 09000000    call    00401011                        ; PUSH ASCII "kernel32"
00401008  |.  6B 65 72 6E 6)ascii   "kernel32",0
00401011  |>  5A              pop     edx
00401012  |.  52              push    edx                             ; /FileName
00401013  |.  FF15 02104000 call    dword ptr [<&KERNEL32.LoadLibrar>; \LoadLibraryA
00401019  |.  E8 13000000    call    00401031                        ; PUSH ASCII
"CreateRemoteThread"
0040101E  |.  43 72 65 61 7>ascii   "CreateRemoteThre"
0040102E  |.  61 64 00       ascii   "ad",0
00401031  |>  58              pop     edx
00401032  |.  52              push    edx                             ; /ProcNameOrOrdinal
00401033  |.  50              push    eax                             ; |hModule
00401034  |.  FF15 7E104000 call    dword ptr [<&KERNEL32.GetProcAdd>; \GetProcAddress
0040103A  |.  C9              leave
0040103B  \.  C3              retn
```

图1

这个例子中采用的是我们上面所说的第一种字符串结构存储的方式,可以看到这些字符串信息都会暴露在我们的程序中。我们看看修改后的代码,如图2所示。

```
00401020  |.  E8 13000000    call    00401044
00401031  |.  0031           add     byte ptr [ecx], dh
00401033  |.  26:2237        and     dh, byte ptr es:[edi]
00401036  |.  26:1126        adc     dword ptr es:[esi], esp
00401039  |.  2E:2C 37       sub     al, 37
0040103C  |.  26:17          pop     ss
0040103E  |.  2831           sub     esi, dword ptr [ecx]
00401040  |.  26:2227        and     ah, byte ptr es:[edi]
00401043  |.  43             inc     ebx
00401044  |$  51             push    ecx
00401045  |.  52             push    edx
00401046  |.  6A 13          push    13
00401048  |.  59             pop     ecx
00401049  |.  BA 31104000    mov     edx, 00401031
0040104E  |>  8074^0A FF 43  /xor    byte ptr [edx+ecx-1], 43
00401053  |.^ E2 F9          \loopd   short 0040104E
00401055  |.  5A             pop     edx
00401056  |.  59             pop     ecx
00401057  |.  5A             pop     edx
00401058  |.  52             push    edx                             ; /ProcNameOrOrdinal
00401059  |.  50             push    eax                             ; |hModule
0040105A  |.  FF15 A6104000 call    dword ptr [<&KERNEL32.GetProcAdd>; \GetProcAddress
00401060  |.  C9             leave
00401061  \.  C3             retn
```

图2

这是第一种结构,我们来对比一下第二种结构修改前后的区别。原始结构如图3所示,修改后的结果如图4所示。

```
00401000 >/$  55              push    ebp
00401001  |.  89E5            mov     ebp, esp
00401003  |.  6A 00           push    0
00401005  |.  68 656C3332     push    32336C65
0040100A  |.  68 6B65726E     push    6E72656B
0040100F  |.  89E2            mov     edx, esp
00401011  |.  52              push    edx                             ; /FileName
00401012  |.  FF15 8A104000 call    dword ptr [<&KERNEL32.LoadLibrar>; \LoadLibraryA
00401018  |.  83C4 0C         add     esp, 0C
0040101B  |.  68 61614000     push    406161
00401020  |.  68 54657265     push    65726554
00401025  |.  68 6D6F7465     push    65746F6D
0040102A  |.  68 74655265     push    65526574
0040102F  |.  68 43726561     push    61657243
00401034  |.  89E2            mov     edx, esp
00401036  |.  52              push    edx                             ; /ProcNameOrOrdinal
00401037  |.  50              push    eax                             ; |hModule
00401038  |.  FF15 8610A000 call    dword ptr [<&KERNEL32.GetProcAdd>; \GetProcAddress
0040103E  |.  83C4 0E         add     esp, 0E
00401041  |.  C9              leave
00401042  \.  C3              retn
```

图3

```
00401000 >/$  55              push    ebp
00401001  |.  89E5            mov     ebp, esp
00401003  |.  6A 00           push    0
00401005  |.  5A              pop     edx
00401006  |.  C1CA 06         ror     edx, 6
00401009  |.  52              push    edx
0040100A  |.  90              nop
0040100B  |.  68 AC1900BC     push    BC0019AC
00401010  |.  5A              pop     edx
00401011  |.  C1CA 06         ror     edx, 6
00401014  |.  52              push    edx
00401015  |.  90              nop
00401016  |.  90              nop
00401017  |.  68 B05A999C     push    9C995AB0
0040101C  |.  5A              pop     edx
0040101D  |.  C1CA 06         ror     edx, 6
00401020  |.  52              push    edx
00401021  |.  90              nop
00401022  |.  90              nop
00401023  |.  90              nop
00401024  |.  89E2            mov     edx, esp
00401026  |.  52              push    edx                             ; /FileName
00401027  |.  FF15 C4104000 call    dword ptr [<&KERNEL32.LoadLibrar>; \LoadLibraryA
0040102D  |.  83C4 0C         add     esp, 0C
```

图4

第一种修改方式中,是取得字符串中第一个字符的ASCII码来作为密钥,所以随着字符串的不同,我们的密钥也是不同的。第二种修改方式是通过循环取得字符串中的4字节字符, "rol edx,6"进行加密后压入堆栈,然后根据字符串的长度循环插入nop指令,使结构都不存在相同的特征。

看了以上的对比,我们可以很清楚地看到,先前的敏感数据信息都被我们的格式随机代码给打乱了,AV很难匹配出我们的数据信息来。那么我们如何高效地达到以上目的呢?由于我平常编写木马都是使用FASM编写,现在即使是写一些大型程序,我还是以FASM和高级语言相结合,让自己的ASM loader加载高级语言编写的模块,这样对抗AV的效果每次只需要修改loader就可以了。下面我们就看看上面两种修改方式的汇编源代码。

```
//第一种方法
include 'AntiData.inc'
entry    $
push     ebp
mov      ebp, esp
x_call   'kernel32'
pop              edx
invoke    LoadLibrary, edx
x_call   'CreateRemoteThread'
pop              edx
invoke    GetProcAddress, eax, edx
leave
ret
//第二种方法
include 'AntiData.inc'
entry   $
push    ebp
mov             ebp, esp
x_push 'kernel32'
mov             edx, esp
invoke  LoadLibrary, edx
x_end
x_push 'CreateRemoteThread'
mov             edx, esp
invoke  GetProcAddress, eax, edx
x_end
leave
ret
```

善于观察的可能会发现,源代码中多了x_

call、x_push、x_end,没错,这几个指令才是重点,因为所有的工作都是它们完成的。这些指令其实是我写的宏,宏在A S M编写中十分重要,充分利用它会发觉有意想不到的效果。好了,既然主角已经出现,就让我们看看它的真面目吧!

```
macro alignsize value { rb (value-1)-($+value-1)
mod value }
macro x_call str
{  local size, .x, .s, .l, x, v
if str eqtype ''
call    .s
.x db str, 0
size = ($ - .x)
if ~ size eq
load  x BYTE from (.x)
repeat size
load  v BYTE from .x+%-1
store BYTE v xor x at .x+%-1
end repeat
.s:
push    ecx
push    edx
mov     ecx, size
mov     edx, .x
.l:
xor     BYTE [edx+ecx-1], x
loop    .l
pop     edx
pop     ecx
end if
else
push str
end if
}
macro x_push str
{
local .x, .v, .t
virtual at 0
db str, 0
alignsize 4
size = $
end virtual
.x = size
repeat size/4
virtual at 0
db str, 0
alignsize 4
load .v DWORD from (.x - 4)
.v = ((.v shl 6) and 0FFFFFFFFh) or (.v shr (32 -
6)) , .v = .v rol 6
end virtual
push    .v
```

```
pop     edx     , mov edx, .v
ror     edx, 6
push    edx
times   % nop
.x = .x - 4
end repeat
}
macro x_end
{
add     esp, size
}
```

alignsize宏没什么可说的,相信有一定编程基础的都应该了解。我们重点看x_call、x_push宏。

x_call用于判断参数类型是否为字符串类型,如果是则先将字符串进行定义,然后压入字符串地址。"if str eqtype…size=($ - .x)"这段代码判断字符串大小是否为0,如果不为0则取字符串第一个字符作为密钥保存到x常量中。然后根据字符串大小,循环取字符串中的每个字符与x常量异或后再重写回去。"if ~size eq… end repeat"这段代码,由于刚刚已经把我们之前定义的字符串通过store加密重写回去了,所以下面我们肯定要加上解密指令了,即"push ecx"到"pop ecx"这段代码。

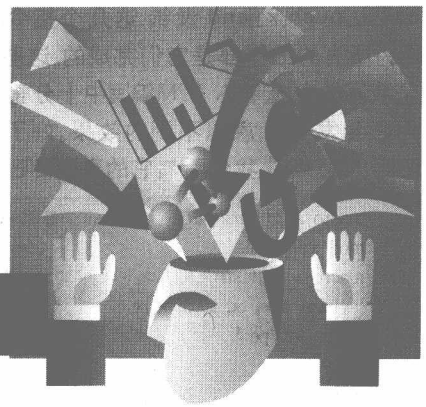
x_push是定义一段虚拟内存区域(4字节对齐)用来存放我们的字符串,然后取其字符串大小。"virtual at 0…end virtual"这段代码,通过字符串大小,循环倒序取字符串中的4字节字符,然后通过rol 6加密,将其传送到edx寄存器。通过ror 6进行解密后压入堆栈,最后是依据循环次数插入等同数量的nop指令。

由于我们的x_push宏是通过堆栈来存储数据的,所以要在函数调用完成后恢复堆栈,即通过"add esp,size"来进行恢复堆栈平衡,而这就是x_end要实现的功能。

至此,本文就告一段落了,主要给大家带来的是通过修改自身的字符串参数结构来对抗A V数据查杀技术。其实我觉得更重要的是带来一种思路,举一反三,就会在A V对抗技术领域中飞得更高!

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)

前置知识:VC
关键词:编程、MAC地址、NDIS



Hook NDIS 实现MAC过滤

文/ 陈布雨

本文的主要目的是为了实现突破MAC绑定来锁定机器码的一个简单欺骗,并针对网卡驱动未提供自定义MAC接口(比如Intel 的很多无线网卡驱动都没有这个接口),期望能够修改底层网卡 MAC。经过一段时间的研究,发现前者是可行的,而后者由于涉及厂家对MAC地址的处理,一般来说是无法直接实现的。在下面的演示中将逐步讲解这个问题。

首先我们来看如何得到网卡的MAC地址:

```
static void GetMACaddress(void)
{
IP_ADAPTER_INFO AdapterInfo[16];
DWORD dwBufLen = sizeof(AdapterInfo);
DWORD dwStatus = GetAdaptersInfo(
AdapterInfo,
&dwBufLen);
assert(dwStatus == ERROR_SUCCESS);
PIP_ADAPTER_INFO pAdapterInfo = AdapterInfo;
do {
PrintMACaddress(pAdapterInfo->Address);
pAdapterInfo = pAdapterInfo->Next;
}
while(pAdapterInfo);
}
```

看到这里有读者可能就要问了,挂钩这个函数不就可以了吗?但我们碰到的案例却恰好是从驱动层来获取MAC地址的(这个程序本身就是防火墙程序),用户态挂钩显然不适用。经过简单的搜索,发现防火墙程序要实现包过滤肯定需要得到MAC地址,而通过对这个函数的跟踪,确定最终请求会转到NDIS相关的驱动中,于是我们找到DDK,搜索一下网卡接口:

Using the Network Driver Design Guide (NDIS 5.1)

Windows-based operating systems support several types of kernel-mode network drivers. The network drivers documentation describes how to write these drivers. This topic briefly describes the supported network driver types and explains which sections of this guide you should read before writing each type of network driver.

Microsoft Windows 2000 and later versions of the operating system support four types of kernel-mode network drivers.

Miniport drivers

A *miniport driver* directly manages a network interface card (NIC) and provides an interface to higher-level drivers.

Intermediate drivers

An *intermediate driver* interfaces between upper-level protocol drivers, such as a legacy transport driver, and a miniport driver. A typical reason to develop an intermediate protocol driver is to perform media translation between an existing legacy transport driver and a miniport driver that manages a NIC for a new media type unknown to the transport driver.

Protocol drivers

An *upper-level protocol driver* implements a TDI interface, or possibly an application-specific interface, at its upper edge to provide services to users of the network. At its lower edge, a protocol driver provides a protocol interface to pass packets to and receive incoming packets from the next-lower driver.

Another type of protocol driver is a connection-oriented call manager. A call manager provides call setup and tear-down services for connection-oriented clients, which are also protocol drivers.

Filter-hook driver

A *filter-hook driver* is used to filter packets. This

```
driver extends the functionality of the IP filter driver,
which is supplied with the operating system.
```

从以上文档可以发现，位于底层的是Miniport Drivers,因此所有对于网卡的读写信息都是从这里返回的。幸运的是,DDK的随机文档中有一份网卡的Sample驱动代码,我们何不拿出来读读呢?找到名为e100bex的项目,简单搜索后找到以下代码:

```
case OID_802_3_PERMANENT_ADDRESS;
pInfo = Adapter->PermanentAddress;
ulBytesAvailable = ulInfoLen = ETH_LENGTH_
OF_ADDRESS;
break;
case OID_802_3_CURRENT_ADDRESS;
pInfo = Adapter->CurrentAddress;
ulBytesAvailable = ulInfoLen = ETH_LENGTH_
OF_ADDRESS;
break;
```

可以看出这里就是返回信息的地方,向上回溯,找到这段代码来自以下函数:

```
NDIS_STATUS MPQueryInformation(
IN  NDIS_HANDLE   MiniportAdapterContext,
IN  NDIS_OID      Oid,
IN  PVOID         InformationBuffer,
IN  ULONG         InformationBufferLength,
OUT PULONG        BytesWritten,
OUT PULONG        BytesNeeded
)
```

而这个函数恰好在EntryPoint的时候调用了。

```
NdisZeroMemory(&MPChar, sizeof(MPChar));
MPChar.MajorNdisVersion = MP_NDIS_MAJOR_
VERSION;
MPChar.MinorNdisVersion = MP_NDIS_MINOR_
VERSION;
...
MPChar.QueryInformationHandler =
MPQueryInformation;
...
DBGPRINT(MP_LOUD, ("Calling
NdisMRegisterMiniport...\n"));
Status = NdisMRegisterMiniport(
NdisWrapperHandle,
&MPChar,
sizeof(NDIS_MINIPORT_CHARACTERISTICS));
```

由此可以看出,网卡驱动调用NdisMRegisterMiniport来注册自己的回调函数事件。因此,我们若需要欺骗上层驱动,只需要挂钩这个函数即可。而接下来我们就要讨论如何实现对这个回调函数挂钩的技巧。先来看看NdisMRegisterMiniport函数的原型,代码如下:

```
NDIS_STATUS
NdisMRegisterMiniport(
IN NDIS_HANDLE  NdisWrapperHandle,
IN PNDIS_MINIPORT_CHARACTERISTICS
MiniportCharacteristics,
IN UINT  CharacteristicsLength
);
```

注意看结构PNDIS_MINIPORT_CHARACTERISTICS,其代码如下:

```
typedef struct _NDIS_MINIPORT_
CHARACTERISTICS {
UCHAR MajorNdisVersion;
UCHAR MinorNdisVersion;
UINT Reserved;
W_CHECK_FOR_HANG_
HANDLER CheckForHangHandler;
W_DISABLE_INTERRUPT_
HANDLER DisableInterruptHandler;
W_ENABLE_INTERRUPT_HANDLER
EnableInterruptHandler;
W_HALT_HANDLER HaltHandler;
W_HANDLE_INTERRUPT_HANDLER
HandleInterruptHandler;
W_INITIALIZE_HANDLER InitializeHandler;
W_ISR_HANDLER ISRHandler;
W_QUERY_INFORMATION_HANDLER
QueryInformationHandler;
W_RECONFIGURE_HANDLER ReconfigureHandler;
W_RESET_HANDLER ResetHandler;
W_SEND_HANDLER SendHandler;
W_SET_INFORMATION_HANDLER
SetInformationHandler;
W_TRANSFER_DATA_HANDLER
TransferDataHandler;
```

这个结构存放着回调函数,接下来我们就要开始玩弄我们的NDIS了。这里有几个办法,比较正式的办法是写一个Intermediate Layer驱动架在网卡与上层防火墙之类的驱动之间,如图1所示。

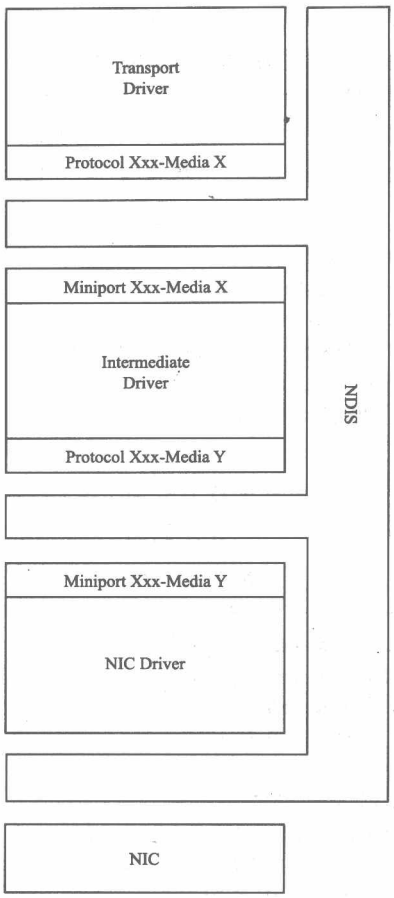


图1

但这显然是没有必要的,写一个这样的驱动太耗费时间,因此我们自然而然地想到了Hook。不过用传统的方法挂钩NdisMRegisterMiniport后调用原始函数的方法存在问题,因为当我们修改传入的MiniportCharacteristics结构数据为自己的函数的时候,会回调下面的函数:

```
NDIS_STATUS
MiniportQueryInformation(
IN NDIS_HANDLE   MiniportAdapterContext,
IN NDIS_OID   Oid,
IN PVOID   InformationBuffer,
IN ULONG   InformationBufferLength,
OUT PULONG   BytesWritten,
OUT PULONG   BytesNeeded
);
```

我们只能得到MiniportAdapterContext,所以还需要挂钩注册Context函数,通过一个Map数据结构来存放用于记录原始函数指针的数据结构,完成MiniportAdapaterContext到我们存放原始信息结构的转换:

```
NDIS_STATUS
MiniportInitialize(
OUT PNDIS_STATUS   OpenErrorStatus,
OUT PUINT   SelectedMediumIndex,
IN PNDIS_MEDIUM   MediumArray,
IN UINT   MediumArraySize,
IN NDIS_HANDLE   MiniportAdapterHandle,
IN NDIS_HANDLE   WrapperConfigurationContext
);
```

这样我们就要挂钩两个函数,并且还需要挂钩相关的DeInitalize函数来释放用于存放这个Map的数据,此处我们将利用一个技巧来避免繁琐的挂钩操作。首先定义一个叫做TThuck的结构用于存放机器码:

```
#pragma pack (push, 1)
struct TThunk
{
UCHAR popeax;
UCHAR push;
ULONG addr;
UCHAR pusheax;
UCHAR push2;
ULONG addr2;
UCHAR ret;
};
#pragma pack (pop)
```

然后在我们的回调挂钩函数中写入以下代码:

```
NDIS_STATUS NTAPI MyNdisMRegisterMiniport(IN
NDIS_HANDLE   NdisWrapperHandle,IN PNDIS_
MINIPORT_CHARACTERISTICS   MiniportCharacteristics,IN
UINT   CharacteristicsLength)
{
//if(Status == NDIS_STATUS_SUCCESS)
{
TThunk * Thunk = (TThunk*)ExAllocatePool(Non
PagedPool,32);
Thunk->popeax = 0x58;
Thunk->push = 0x68;
Thunk->addr = (ULONG)MiniportCharacteristics->
QueryInformationHandler;
Thunk->pusheax = 0x50;
Thunk->push2 = 0x68;
Thunk->addr2 = (ULONG)
MyMiniportQueryInformation;
Thunk->ret = 0xC3;
MiniportCharacteristics->QueryInformationHandler
= (W_QUERY_INFORMATION_HANDLER)Thunk;
```

```
    }
    NDIS_STATUS Status =
    DNdisMRegisterMiniport.pTrampolineFunc(NdisWra
pperHandle,MiniportCharacteristics,CharacteristicsLength);
    DbgPrint("MyNdisMRegisterMiniport Status =
    0x%08X \n",Status);
    return Status;
    }
```

这段代码仅仅是生成一段x86机器码,并把这段机器码的入口替换为MiniportCharacteristics->QueryInformationHandler,这样当调用MiniportCharacteristics->QueryInformationHandler的时候,会首先执行我们生成的机器码,然后再调用原始例程并返回。

我们来看看这段机器码。首先使用pop eax将调用者的返回地址从Stack 弹出放到寄存器eax中(recall,在x86中调用者使用call 调用目标函数,当进入目标函数时Stack顶端放置的是函数的返回地址),然后将原始的QueryInformationHandler通过Push指令压到Stack中,把刚才弹到eax的函数地址再次放入Stack顶端,最后为了避免修正Jmp 的Relative Address的麻烦,使用push imm32 /ret的方法将我们的回调MyMiniportQueryInformation 放入并跳到该地址,于是这个时候在MyMiniportQueryInformation入口,我们的堆的内容如下:

```
Stack top
Esp + 0    Caller Returning Address Of
QueryInformationHandler
Esp + 4    Origin Address Of
QueryInformationHandler
Esp + 8    Arg0 —> MiniportAdapterContext
Esp + C    Arg1 —> OIC
...
Bottom
```

因此,我们就要声明自己的以下函数:

```
NDIS_STATUS NTAPI
MyMiniportQueryInformation(
TMiniportQueryInformation *
PrevMiniportQueryInformation,
IN NDIS_HANDLE  MiniportAdapterContext,
IN NDIS_OID  Oid,
IN PVOID  InformationBuffer,
IN ULONG  InformationBufferLength,
OUT PULONG  BytesWritten,
OUT PULONG  BytesNeeded)
```

相比原来的,仅仅是多了一个原始函数的参数一,其他不变,这样我们就可以直接调用到原始函数而不需要通过Map等方法查询了,具体实现代码如下:

```
{
NDIS_STATUS Status =
PrevMiniportQueryInformation(
MiniportAdapterContext,
Oid,
InformationBuffer,
InformationBufferLength,
BytesWritten,
BytesNeeded);
DbgPrint("Status= 0x%08X,
PrevMiniportQueryInformation = 0x%08X
MyMiniportQueryInformation Oid :
0x%08X\n",Status,PrevMiniportQueryInformation,Oid);
if(Status == NDIS_STATUS_SUCCESS)
{
if(Oid == OID_802_3_PERMANENT_ADDRESS
| Oid == OID_802_3_CURRENT_ADDRESS)
{
PUCHAR P = (PUCHAR)InformationBuffer;
DbgPrint("Mac : %02X %02X %02X %02X %02X
%02X\n",P[0],P[1],P[2],P[3],P[4],P[5]);
*BytesWritten = sizeof Mac;
memcpy(InformationBuffer,Mac,sizeof Mac);
//return NDIS_STATUS_SUCCESS;
}
}
return Status;
}
```

我们首先调用原始函数,如果函数成功,检索是否是我们需要的Oid请求,如果是就过滤掉。最后我们需要把自己的Hook代码驱动安装在网卡驱动初始化之前,如何实现就留给大家来发挥了。

我们用简单的几十行代码就完成了对Mac地址的过滤,但这仅仅能够欺骗上层驱动和用户态通过相关函数来获取Mac地址的方法,我在文章开头就已经说过,真正的网卡与交换机交换的时候的Mac地址,我们在不借助网卡驱动的情况下是无法直接修改的。下面继续用Sample NIC Driver代码来说明问题:

```
// Read NetworkAddress registry value
// Use it as the current address if any
if (Status == NDIS_STATUS_SUCCESS)
{
NdisReadNetworkAddress(
```

```
&Status,
&NetworkAddress,
&Length,
ConfigurationHandle);
// If there is a NetworkAddress override
// in registry, use it
if ((Status == NDIS_STATUS_SUCCESS) &&
(Length == ETH_LENGTH_OF_ADDRESS))
{
if ((ETH_IS_MULTICAST(NetworkAddress)
| ETH_IS_BROADCAST(NetworkAddress))
| !ETH_IS_LOCALLY_ADMINISTERED
(NetworkAddress))
{
DBGPRINT(MP_ERROR,
("Overriding NetworkAddress is invalid - %02x-
%02x-%02x-%02x-%02x-%02x\n",
NetworkAddress[0], NetworkAddress[1],
NetworkAddress[2],
NetworkAddress[3], NetworkAddress[4],
NetworkAddress[5]));
}
else
{
ETH_COPY_NETWORK_ADDRESS(Adapter->
CurrentAddress, NetworkAddress);
Adapter->bOverrideAddress = TRUE;
}
}
Status = NDIS_STATUS_SUCCESS;
}
```

这里网卡通过读取注册表来重写CurrentAddress,换言之,如果网卡不提供这个选项,我们就永远无法修改CurrentAddress,因为CurrentAddress来自以下代码:

```
if (Adapter->bOverrideAddress)
{
ETH_COPY_NETWORK_ADDRESS(Adapter->
CurrentAddress, Adapter->PermanentAddress);
}
```

而这个PermanentAddress则来自以下代码:

```
// Read node address from the EEPROM
for (i=0; i< ETH_LENGTH_OF_ADDRESS; i += 2)
{
usValue = ReadEEprom(Adapter->PortOffset,
(USHORT)(EEPROM_NODE_ADDRESS_BYTE_0 +
(i/2)),
Adapter->EepromAddressSize);
*((PUSHORT)(&Adapter->PermanentAddress[i])) =
usValue;
}
```

但无论是Adapter数据结构的偏移,还是EEPROM的数据偏移和地址都只有厂商自己知道,在软件层是没有办法修改的。从技术上,我们可以通过Patch相应的网卡驱动来实现地址的修改,但这就没有一个比较通用或者挂钩的办法来实现了。

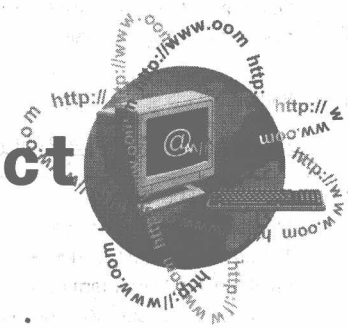
本文的主要目的在于给读者对NDIS结构层的一个概括以及系统对于MAC地址查询请求的处理及其与网卡的交互。另外,本文比较取巧的挂钩方式也为这类应用提供了很大的方便。

(编辑提醒:本文涉及的代码已收入随书光盘,请到光盘中查找)。

前置知识:VC

关键词: Hook、ObcreateObject、进程创建

Hook ObCreateObject 实时监控进程创建



文/图 腾袭

目前比较通用的拦截进程的方法有许多种,像SSDT Hook NtCreateProcessEx、inline Hook ObReferenceObjectByHandle等方法来实现拒绝进程的创建,不过现在网上流

传的方法与Ring3都不具有很好的互交性,一般都是被动的全部拒绝。现在,笔者就来谈谈一种以链表的形式储存信息来与Ring3互交的方式实现实时监控进程创建。鉴于像HookObReferenceObjectByHandle等方法已经在网上被广泛运用,这里笔者介绍另一种方法来实现拦截进程创建。

原 理

我们首先来看看Windows进程的创建过程,这样有助于我们选择一个适合的函数来Hook。

首先是CreateProcess,在非Unicode的环境下,CreateProcess调用的是CreateProcessA,实际上CreateProcessA只是把ASCII字符串转换成Unicode字符串,然后就调用CreateProcessW。而在Unicode的环境下,CreateProcess调用的就直接是CreateProcessW。紧接下来,在CreateProcessW中经过一系列的处理后,我们可以发现接下来它调用了NtCreateProcess:

```
NTSTATUS STDCALL
NtCreateProcess(OUT PHANDLE ProcessHandle, IN
ACCESS_MASK DesiredAccess,
        IN POBJECT_ATTRIBUTES
ObjectAttributes OPTIONAL,
        IN HANDLE ParentProcess, IN
BOOLEAN InheritObjectTable,
        IN HANDLE SectionHandle OPTIONAL,
        IN HANDLE DebugPort OPTIONAL,
IN HANDLE ExceptionPort OPTIONAL)
{
  KPROCESSOR_MODE PreviousMode;
  NTSTATUS Status = STATUS_SUCCESS;

  PAGED_CODE();
  PreviousMode = ExGetPreviousMode();
  if(PreviousMode != KernelMode)
  {
    _SEH_TRY    _SEH_END;
  }

  if(ParentProcess == NULL)
  {
    Status = STATUS_INVALID_PARAMETER;
  }
  else
  {
    Status = PspCreateProcess(ProcessHandle,
  DesiredAccess, ObjectAttributes,
                              ParentProcess,
  InheritObjectTable, SectionHandle,
                              DebugPort,
  ExceptionPort);
  }
  return Status;
}
```

从NtCreateProcess的源码中我们可以发现它调用了PspCreateProcess,不过由于这个函数是未导出函数,Hook起来有点麻烦,所以我们接着往下看:

```
NTSTATUS
PspCreateProcess(OUT PHANDLE ProcessHandle,
IN ACCESS_MASK DesiredAccess,
        IN POBJECT_ATTRIBUTES
ObjectAttributes OPTIONAL,
        IN HANDLE ParentProcess OPTIONAL,
IN BOOLEAN InheritObjectTable,
        IN HANDLE SectionHandle OPTIONAL,
        IN HANDLE DebugPort OPTIONAL,
IN HANDLE ExceptionPort OPTIONAL)
{
  ...

  PreviousMode = ExGetPreviousMode();

  ...

  if(ParentProcess != NULL)
  {
    Status = ObReferenceObjectByHandle(ParentP
rocess,
                        PROCESS_CREATE_
PROCESS, PsProcessType,
                        PreviousMode,
(PVOID*)&pParentProcess, NULL);
    ...
  }
  else
  {
    pParentProcess = NULL;
  }

  ...

  if (SectionHandle != NULL)
  {
      Status = ObReferenceObjectByHandle(Secti
onHandle,
                                    0,
```

```
    MmSectionObjectType, PreviousMode,
    (PVOID*)&SectionObject, NULL);
            ...
    }

    Status = ObCreateObject(PreviousMode,
    PsProcessType, ObjectAttributes,
                    PreviousMode, NULL,
    sizeof(EPROCESS), 0, 0, (PVOID*)&Process);

    ...

    KProcess = &Process->Pcb;
    RtlZeroMemory(Process, sizeof(EPROCESS));
    Status = PsCreateCidHandle(Process,
    PsProcessType, &Process->UniqueProcessId);

    ...

    Process->DebugPort = pDebugPort;
    Process->ExceptionPort = pExceptionPort;

    ...

    KeInitializeDispatcherHeader(&KProcess->
    DispatcherHeader,
                                ProcessObject,
    sizeof(EPROCESS), FALSE);

    ...
}
```

从PspCreateProcess的部分源码中我们可以看到它调用了ObReferenceObjectByHandle、ObCreateObject、PsCreateCidHandle、KeInitializeDispatcherHeader等一系列的函数,这里通过比较,笔者选择了Inline Hook ObCreateObject。

综上所述,可将过程简化为[CreateProcessW() > NtCreateProcess() > PspCreateProcess() > ObCreateObject()]。

实 现

选择好了函数,我们就开始做准备Hook了,先看看ObCreateObject的原型吧:

```
NTSTATUS
ObCreateObject (
    __in KPROCESSOR_MODE ProbeMode,
// 决定是否要验证参数
    __in POBJECT_TYPE ObjectType,
// 对象类型指针
    __in POBJECT_ATTRIBUTES ObjectAttributes,
// 对象的属性,最终会转化成ObAllocateObject需
// 要的OBJECT_CREATE_INFORMATION结构
    __in KPROCESSOR_MODE OwnershipMode,
    __inout_opt PVOID ParseContext,
    __in ULONG ObjectBodySize,
// 对象体大小
    __in ULONG PagedPoolCharge,
    __in ULONG NonPagedPoolCharge,
    __out PVOID *Object
//接收对象体的指针
    );
```

再看看PspCreateProcess调用ObCreateObject的方式:

```
    Status = ObCreateObject(PreviousMode,
PsProcessType, ObjectAttributes,
                PreviousMode, NULL,
sizeof(EPROCESS), 0, 0, (PVOID*)&Process);
```

从中我们可以看出创建进程时ObjectType为PsProcessType,接下来就是实现Hook了:

```
//Hook
void uInit()
{
    KIRQL oldIrql = KeRaiseIrqlToDpcLevel();

    __asm
    {
        cli
        push eax
        mov eax, cr0
        and eax, 0FFFEFFFFh
        mov cr0, eax
        pop eax
    }

    RtlCopyMemory(ProxyFuncData,HookFunc,5);
    ProxyFuncData[5] = 0xe9;
    *(PULONG)(&ProxyFuncData[6]) = (ULONG)
HookFunc - (ULONG)(&ProxyFuncData) - 5;
    *(PUCHAR)HookFunc = 0xe9;
    *(PULONG)((ULONG)HookFunc + 1) = (ULONG)
FakeFunc - (ULONG)HookFunc - 5;
```