数字驱动分析笔记之360SelfProtection

	推荐使用的环境	备注
操作系统	Windows 7 SP3	简体中文版
虚拟机	VM	版本号：15
编译器	VS2013 + WDK8.1	
调试	Windbg	sxe ld xxxx.sys

目录：

1、Fake_ZwCreateFile

序号：0x8
检查函数类型：属于第三类，重点在于返回值
设计思路：
1、防止访问保护路径（杀软）例如：//3600、//3600safe、//3600SafeBox等
2、白名单进程访问直接放行（通用）
3、不合法性的文件句柄清零，错误返回

防止误伤过滤：
1、使用ExGetPreviousMode函数，先判断当前previousMode是否为kernelMode，如果是直接返回0
2、判断当前进程PID是否为白名单进程（zhudongfangyu.exe那几个）或则进程PID是Wininit.exe，如果是直接返回0

函数执行前判断：
略

函数执行后判断：
1、判断该文件信息是否已经存在列表中
存在：放行
不存在：二次检查

```
result = STATUS_SUCCESS;
//0、获取ZwCreateFile原始参数
IN HANDLE              In_FileHandle = *(ULONG*)((ULONG)ArgArray);
IN ACCESS_MASK         In_DesiredAccess = *(ULONG*)((ULONG)ArgArray + 4);
IN POBJECT_ATTRIBUTES  In_ObjectAttributes = *(ULONG*)((ULONG)ArgArray + 8);
IN PIO_STATUS_BLOCK    In_IoStatusBlock = *(ULONG*)((ULONG)ArgArray + 0xC);
IN PLARGE_INTEGER      In_AllocationSize = *(ULONG*)((ULONG)ArgArray + 0x10);
IN ULONG               In_FileAttributes = *(ULONG*)((ULONG)ArgArray + 0x14);
IN ULONG               In_ShareAccess = *(ULONG*)((ULONG)ArgArray + 0x18);
IN ULONG               In_CreateDisposition = *(ULONG*)((ULONG)ArgArray + 0x1C);
//1、判断上次调用原始函数返回值
if (!NT_SUCCESS(InResult))
{
    return InResult;
}
//2、判断HookPort版本合法性,我分析的就是0x3F1
//if (g_HookPort_Version >= 0x3F1)
//{
//  if ((In_ShareAccess == FILE_SHARE_WRITE) || (In_ShareAccess == (FILE_SHARE_WRITE || FILE_SHARE_READ)) || (!In_ShareAccess) || (In_ShareAccess
//  {

//  }
//}
//3、计算校验文件信息
Handle_v5 = *(HANDLE*)In_FileHandle;
Status = Safe_GetInformationFile(Handle_v5, (ULONG)&System_InformationFile, UserMode);
if (!NT_SUCCESS(Status))
{
    return result;
}
//4、查找该文件信息是否在列表中，找到返回1，失败返回0
Status = Safe_QueryInformationFileList(System_InformationFile.IndexNumber_LowPart,
    System_InformationFile.u.IndexNumber_HighPart,
    System_InformationFile.VolumeSerialNumber);
```

2、Safe_GetInformationFile函数功能（获取卷信息和文件唯一ID信息）
1、过滤掉Handle不是4的倍数（判断句柄合法性）
2、过滤掉特定文件设备类型

```
    //2、得到文件对象指针
    Status = ObReferenceObjectByHandle(Handle, FILE_ANY_ACCESS, *IoFileObjectType, AccessMode, (PVOID*)&FileObject, NULL);
    //2、1判断操作是否成功
    if (!NT_SUCCESS(Status) && !FileObject)
    {
        return Status;
    }
    //2、2 判断设备对象
    if (!FileObject->DeviceObject)
    {
        //关闭设备句柄
        ObfDereferenceObject(FileObject);
        Status = STATUS_UNSUCCESSFUL;
        return Status;
    }
    //3、过滤掉特定文件设备类型
    DeviceType = FileObject->DeviceObject->DeviceType;
    if (DeviceType != FILE_DEVICE_DISK_FILE_SYSTEM    &&   //磁盘文件系统设备
        DeviceType != FILE_DEVICE_DISK                &&   //磁盘设备
        DeviceType != FILE_DEVICE_FILE_SYSTEM         &&   //文件系统设备
        DeviceType != FILE_DEVICE_UNKNOWN             &&   //未知类型
        DeviceType != FILE_DEVICE_CD_ROM              &&   //CD光驱设备
        DeviceType != FILE_DEVICE_CD_ROM_FILE_SYSTEM  &&   //CD光驱文件系统设备
        DeviceType != FILE_DEVICE_NETWORK_FILE_SYSTEM      //网络文件系统设备
        )
    {
        if (DeviceType != FILE_DEVICE_NETWORK_REDIRECTOR)  //网卡设备
        {
            //关闭设备句柄
            ObfDereferenceObject(FileObject);
            Status = STATUS_UNSUCCESSFUL;
            return Status;
        }
    }
    if (DeviceType == FILE_DEVICE_MULTI_UNC_PROVIDER)      //多UNC设备
```

3、假设驱动名称是\\Driver\\Fastfat标志位置1（这个名称有点陌生，不太清楚干嘛的）

置1：ZwQueryVolumeInformationFile的参数是FileInternalInformation

置0：ZwQueryVolumeInformationFile的参数是FileBasicInformation

3、Safe_CheckProtectPath二次判断访问的文件目录

1、ZwQueryInformationFile的参数是FileStandardInformation判断目录

2、通过ZwQueryObject获取对象名信息

3、防止打开特定路径（用户自定义）

4、合法返回1，非法返回0

```
        //判断打开路径是不是//3600、//3600safe、//3600SafeBox等
        if (!Safe_CheckProtectPath(Handle_v5, UserMode))
        {
            Handle_v5 = *(HANDLE*)In_FileHandle;
        }
        else
        {
            result = STATUS_SUCCESS;
            return result;
        }
    }
    //保护进程直接句柄清零，禁止访问
    Safe_ZwNtClose(Handle_v5, g_VersionFlag);
    *(HANDLE*)In_FileHandle = 0;
    result = STATUS_ACCESS_DENIED;
    return result;
}
```

2、Fake_ZwOpenKey

参考资料：

1、几种常见的注入姿势

2、Determine Allow-Deny List and Application Inventory for Software Restriction Policies

3、分析了一下360安全卫士的HOOK(二)——架构与实现

序号：0x32

检查函数类型：属于第三类，重点在于返回值

设计思路：

1、拦截HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion \Windows\AppInit_DLLs（注册表注入，详情参考资料1）

2、拦截\\SHELLEXECUTEHOOKS（自启动）

3、拦截\\SAFER\\CODEIDENTIFIERS（软件限制策略，平时没接触过属于知识盲区，详情参考资料2）

如果是以上几种行为直接拦截，将句柄清零，并返回STATUS_ACCESS_DENIED

防止误伤过滤：

1、必须是用户层调用

2、权限包含（Key_Read || MAXIMUM_ALLOWED）就启动调用后检查

3、非白名单保护进程

函数执行前判断：

```
NTSTATUS    Status, result;
result = STATUS_SUCCESS;
//0、获取ZwOpenKey的原始参数
ACCESS_MASK    DesiredAccess = *(ULONG*)((ULONG)ArgArray + 4);
//1、必须是应用层调用
if (ExGetPreviousMode())
{
    //2、高权限启动函数执行后检查
    if (KEY_READ == DesiredAccess || MAXIMUM_ALLOWED == DesiredAccess)
    {
        //判断是不是保护进程，是返回：1  不是返回0
        if (Safe_QueryWhitePID(PsGetCurrentProcessId()))
        {
            //3、启动调用后检查
            *(ULONG*)ret_func = After_ZwOpenKey_Func;
        }
    }
}
return result;
```

函数执行后判断：

通过ObReferenceObjectByHandle+ObQueryNameString获取注册表路径

```
    {
        KdPrint(("ProbeRead(After_ZwOpenKey_Func: In_KeyHandle) error \r\n"));
        return result;
    }
    Status = ObReferenceObjectByHandle(*(ULONG*)In_KeyHandle, 0, 0, UserMode, &Object, 0);
    if (!NT_SUCCESS(Status))
    {
        return result;
    }
    pFileNameInfo = (POBJECT_NAME_INFORMATION)Safe_AllocBuff(NonPagedPool, NumberOfBytes, Tag);
    if (!pFileNameInfo)
    {
        ObfDereferenceObject(Object);
        return result;
    }
    //2、驱动层通过注册表_OBJECT指针查询注册表路径ObQueryNameString
    Status = ObQueryNameString(Object, pFileNameInfo, NumberOfBytes, &ReturnLength);
    //解引用
    ObfDereferenceObject(Object);
    if (!NT_SUCCESS(Status) || !pFileNameInfo->Name.Buffer || !pFileNameInfo->Name.Length)
    {
```

然后再判断上述三种违规敏感路径

```
    //3、判断各种违规路径
    // 自启动
    if (wcsstr(pFileNameInfo->Name.Buffer, L"\\SHELLEXECUTEHOOKS"))
    {
        //自启动
        ErrorFlag = FALSE;
        result = STATUS_ACCESS_DENIED;
    }
    //利用AppCertDlls注册表，将HKLM\System\CurrentControlSet\Control\Session Manager\AppCertDlls下写入dll的
    else if (wcsstr(pFileNameInfo->Name.Buffer, L"CONTROL\\SESSION MANAGER\\APPCERTDLLS"))
    {
         //AppCertDlls注入
        ErrorFlag = FALSE;
        result = STATUS_OBJECT_NAME_NOT_FOUND;
    }
    //软件限制策略（平时没关注过）
    else if (wcsstr(pFileNameInfo->Name.Buffer, L"\\SAFER\\CODEIDENTIFIERS"))
    {
        //软件限制策略
        ErrorFlag = FALSE;
        result = STATUS_ACCESS_DENIED;
    }
    else
```

3、Fake_ZwLoadDriver

参考资料：

1、分析了一下360安全卫士的HOOK(二)——架构与实现

2、总结一把，较为精确判断SCM加载

3、驱动加载监控x86

4、破译A盾禁止加载驱动失效之谜

5、拦截LPC监控服务加载和启动

预备知识：

1、问题 在ZwLoadDriver中 如果是用SCM加载驱动,那么得到的进程路径是server.exe,解决的方法是hook下面的函数

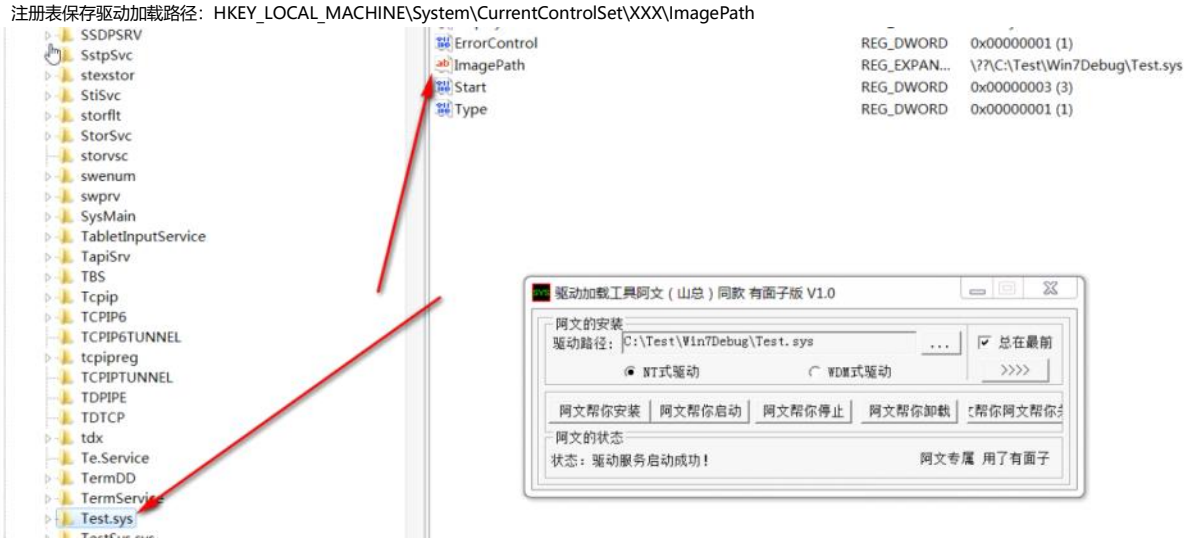
XP：

NtRequestWaitReplyPort

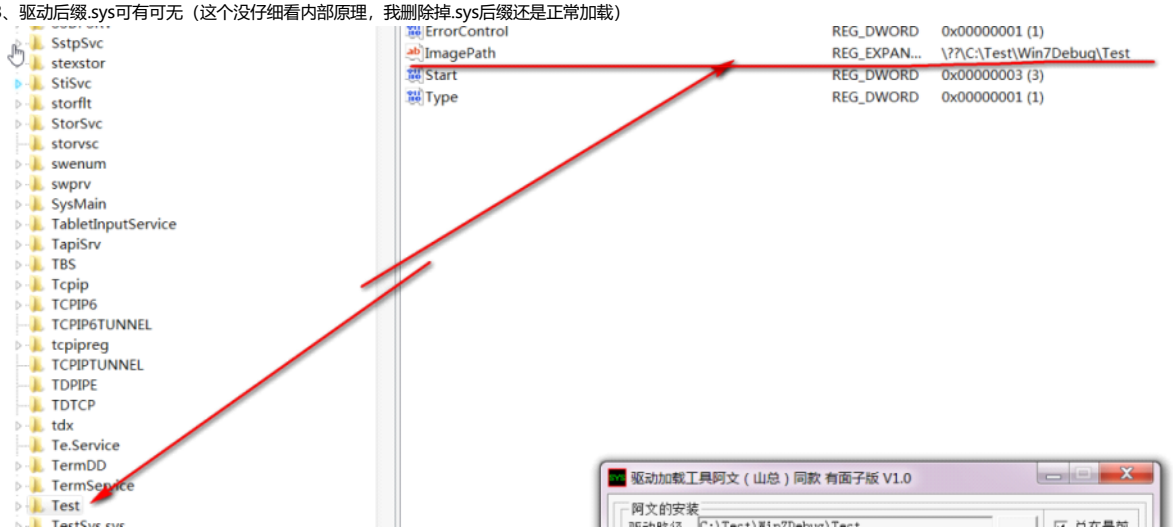
Win7:

ZwAlpcSendWaitReceivePort

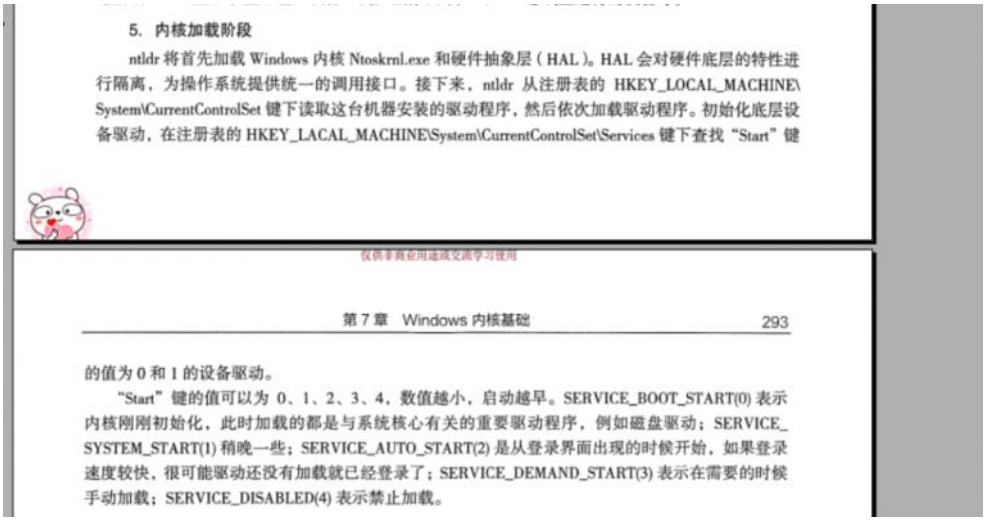
2、注册表保存驱动加载路径：HKEY_LOCAL_MACHINE\System\CurrentControlSet\XXX\ImagePath



3、驱动后缀.sys可有可无（这个没仔细看内部原理，我删除掉.sys后缀还是正常加载）



4、"Start"键值决定驱动加载顺序



5. 内核加载阶段

ntldr 将首先加载 Windows 内核 Ntoskrnl.exe 和硬件抽象层（HAL）。HAL 会对硬件底层的特性进行隔离，为操作系统提供统一的调用接口。接下来，ntldr 从注册表的 HKEY_LOCAL_MACHINE\System\CurrentControlSet 键下读取这台机器安装的驱动程序，然后依次加载驱动程序。初始化底层设备驱动，在注册表的 HKEY_LACAL_MACHINE\System\CurrentControlSet\Services 键下查找“Start”键

的值为 0 和 1 的设备驱动。

“Start”键的值可以为 0、1、2、3、4，数值越小，启动越早。SERVICE_BOOT_START(0) 表示内核刚刚初始化，此时加载的都是与系统核心有关的重要驱动程序，例如磁盘驱动；SERVICE_SYSTEM_START(1) 稍晚一些；SERVICE_AUTO_START(2) 是从登录界面出现的时候开始，如果登录速度较快，很可能驱动还没有加载就已经登录了；SERVICE_DEMAND_START(3) 表示在需要的时候手动加载；SERVICE_DISABLED(4) 表示禁止加载。

序号：0x22

检查函数类型：属于第一类，函数执行前检查

设计思路：

1、生杀大权交给用户决定

2、自带一个文件存储拦截or放行名单

3、配合NtRequestWaitReplyPort or ZwAlpcSendWaitReceivePort 获取真实的加载驱动意图者

4、加载or卸载都调用这个API

防止误伤过滤：

1、必须是用户层调用

2、SeSinglePrivilegeCheck检查权限

函数执行前判断：

1、调用SeSinglePrivilegeCheck检查权限，hook掉也可以进行拦截驱动的操作（参考A盾）

2、获取真实的加载驱动意图者（找到图2、1，找不到图2、2）

```
if ( *(a2 + 4) )
{
  if ( ExGetPreviousMode() )
  {
    v2 = ExGetPreviousMode();
    if ( SeSinglePrivilegeCheck(PrivilegeValue, v2) )// 进程/线程有特权加载驱动程序
    {
      v3 = IoGetCurrentProcess();
      if ( Safe_InsertSystemInformationList(v3, 5) )// SCM加载调用者是services.exe
      {
        if ( dword34EE8_CurrentProcessId )    // 发起来源信息（谁加载这个驱动）Fake_ZwRequestWaitReplyPort函数进行获取
        {
          CurrentProcessId = dword34EE8_CurrentProcessId;// 替换掉发起源PID
          v4 = dword34EEC_CurrentThreadId;
        }
        else
        {
          CurrentProcessId = PsGetCurrentProcessId();// 找不到默认就说发起源是services.exe
          v4 = PsGetCurrentThreadId();
        }
        Flag = 0;
      }
      else
      {
        CurrentProcessId = PsGetCurrentProcessId();
        v4 = PsGetCurrentThreadId();
        Flag = 1;
      }
      CurrentThreadId = v4;
      ms_exc.registration.TryLevel = 0;
      ProbeForRead(*a2, 8u, 1u);
      v5 = *a2;
      if ( !**a2
        || (v10 = *&v5->Length,
            v11 = v5->Buffer,
            ProbeForRead(v5->Buffer, v5->Length, 2u),
            v6 = ExAllocatePoolWithTag(0, **a2, 0x206B6444u),
            (P = v6) == 0) )
      {
        ms_exc.registration.TryLevel = -1;
        JUMPOUT(&loc_285EE);
      }
      qmemcpy(v6, *(*a2 + 4), **a2);
      v11 = v6;
```

2、1 找到则显示真实加载驱动意图者，需要配合NtRequestWaitReplyPort or ZwAlpcSendWaitReceivePort

2、2 找不到默认提示services.exe

3、核心部分就是Safe_CheckSys

```
    //将参数In_DriverServiceName拷贝到一个临时变量存储，安全起见
    NumberOfBytes = In_DriverServiceName->Length * 2;
    TestDriverServiceName.Buffer = Safe_AllocBuff(NonPagedPool, NumberOfBytes, Tag);
    RtlCopyUnicodeString(&TestDriverServiceName, In_DriverServiceName);
    TestDriverServiceName.MaximumLength = In_DriverServiceName->MaximumLength;
    if (!TestDriverServiceName.Length)
    {
        return result;
    }
    //初始化OBJECT_ATTRIBUTES的内容
    OBJECT_ATTRIBUTES ObjectAttributes = { 0 };
    ULONG             ulAttributes =
        OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE;
    InitializeObjectAttributes(
        &ObjectAttributes,                          // 返回初始化完毕的结构体
        &TestDriverServiceName,                     // 文件对象名称
        ulAttributes,                               // 对象属性
        NULL, NULL);                                // 一般为NULL
    Status = ZwOpenKey(&KeyHandle, KEY_ALL_ACCESS, &ObjectAttributes);
    ExFreePool(TestDriverServiceName.Buffer);
    if (NT_SUCCESS(Status))
    {
        //拦截还是放行加载驱动
        result = Safe_CheckSys(KeyHandle, CurrentProcessId, CurrentThreadId, Flag);
        if (KeyHandle)
        {
            ZwClose(KeyHandle);
            KeyHandle = NULL;
        }
    }
}
```

3、1 Safe_CheckSys函数分析

1、两种方式检查驱动是否存在，存在则退出（驱动删除.sys后缀也可以加载）

2、判断该驱动加载路径前缀 == HKEY_LOCAL_MACHINE\System\ControlSet001\XXX（看不懂意义何在，有大佬能解释下吗）

个人理解驱动安装成功会在ControlSet001生成一份备份，难道还有不在ControlSet001的情况下？？？？

```
    //4、判断当前驱动加载路径是否在xxxx\ControlSet001内
    if (Safe_CheckControlSetPath(KeyHandle, KeyInformation->NameLength))
    {
        Flag = 9;                                   //Flag=9情况下R3提示也没有发生任何变化
    }
    RtlInitUnicodeString(&DestinationString, KeyInformation->Name);
    //5、判断该驱动是否已经加载
    if (Safe_AppendString_Sys(&DestinationString))
    {
        goto _FunctionRet;
    }
    //6、合成了路径,获取当前驱动路径注册表：ImagePath
    Status = Safe_SetImagePathString(&DestinationString, KeyHandle, &OutImagePathString);
    if (!NT_SUCCESS(Status))
    {
        goto _FunctionRet;
    }
    //7、全路径再次判断是否加载,以及校验驱动签名等等
    if (Safe_GetModuleBaseAddress(&OutImagePathString, NULL, NULL, NULL)
        || (Status = Safe_1D044_SendR3(CurrentProcessId, CurrentThreadId, Flag, &OutImagePathString), NT_SUCCESS(Status))
        )
    {
        //成功返回
        result = STATUS_SUCCESS;
    }
    else
    {
        //失败返回
        result = STATUS_ACCESS_DENIED;
    }
_FunctionRet:
    //释放空间
    if (OutImagePathString.Buffer)
    {
```

3、2 Safe_1D044_SendR3函数分析

1、根据Safe_CheckSys_SignatureOrHash返回值执行不同流程

```
    //2、判断该驱动的合法性例如：签名、白名单等等
    Status = Safe_CheckSys_SignatureOrHash(In_ProcessID, In_ThreadID, In_ImagePathString, &uHash, &FileSize, &Pass_Flag, Flag_v4);
    if (NT_SUCCESS(Status))
    {
        //3、必须有R3交互界面才继续执行，否则后面发拦截信息给R3就没必要了
        if (SpecialWhiteNumber)
        {
            //0错误   1放行    2正常执行流程（要检查的）
            switch (Pass_Flag)
            {
                case 0:     //拦截进程 返回错误值即可
                {
                    result = STATUS_ACCESS_DENIED;
                    break;
                }
                case 1:     //白名单进程 正常返回 STATUS_SUCCESS
                {
                    result = STATUS_SUCCESS;
                    break;
                }
                case 2:     //检查的流程
                {
```

3、3 Safe_CheckSys_SignatureOrHash函数分析

1、合法签名直接退出，返回1（白名单进程 正常退出）

2、不合法签名查询黑白名单驱动列表

```
    //4、 读取文件
    Status = ZwReadFile(
        FileHandle,     // 文件句柄
        NULL,           // 信号状态(一般为NULL)
        NULL, NULL,     // 保留
        &IoStatusBlock, // 接受函数的操作结果
        pBuff,          // 保存读取数据的缓存
        ulLength,       // 想要读取的长度
        NULL,           // 读取的起始偏移
        NULL);          // 一般为NULL
    if (!NT_SUCCESS(Status))
    {
        goto _FunctionRet;
    }
    //检查签名的代码我没分析略
    //略
    //合法签名直接退出，返回值Out_PassFlag = 1
    //不合法签名二次校验，查询是否在列表中
    //5、后面几步就是计算该PE文件的哈希值操作
    Safe_14F7C_Hash(KeyBuff);
    Safe_15846_Hash(KeyBuff, pBuff, ulLength);
    Safe_158F8_Hash(Out_Hash, KeyBuff);
    //6、查询列表中是否存在
    *(ULONG*)Out_PassFlag = Safe_QueryDrvmkDataList(Out_Hash, ulLength);
    *(ULONG*)Out_FileSize = ulLength;
_FunctionRet:
    result = Status;
    if (pBuff)
    {
        ExFreePool(pBuff);
        pBuff = NULL;
    }
    if (FileHandle)
    {
        ZwClose(FileHandle);
        FileHandle = NULL;
    }
```

3、4 Safe_QueryDrvmkDataList函数分析

```
//判断白名单个数
if (DrvmkListNumber)
{
    //循环查找是否存在
    for (Index = 0; Index < DrvmkListNumber; Index++)
    {
        if (In_FileSize == g_Drvmk_List->Pe_Hash_Data[Index].PESize &&      //判断大小
            *(ULONG*)(In_Hash) == g_Drvmk_List->Pe_Hash_Data[Index].Hash[0]&&
            *(ULONG*)(In_Hash + 4) == g_Drvmk_List->Pe_Hash_Data[Index].Hash[1] &&
            *(ULONG*)(In_Hash + 8) == g_Drvmk_List->Pe_Hash_Data[Index].Hash[2] &&
            *(ULONG*)(In_Hash + 0xC) == g_Drvmk_List->Pe_Hash_Data[Index].Hash[3]
            )
        {
            //找到跳出循环
            break;
        }
    }
    if (Index >= DrvmkListNumber)
    {
        //找不到 失败返回
        Pass_Flag = 2;
    }
    else
    {
        //根据LoadDriver_Flag标识返回对应的值
        if (g_Drvmk_List->Pe_Hash_Data[Index].LoadDriver_Flag)
        {
            //1表示 拦截，所以返回0
            Pass_Flag = 0;
        }
        else
        {
            //0表示 放行，所以返回1
            Pass_Flag = 1;
        }
```

4、假设是正常流程（2）发送给R3用户自行决定：放行（0） or 拦截（1）添加进驱动黑白名单列表中

```
pQuery_Pass->CheckWhitePID = In_ProcessID;
pQuery_Pass->Unknown_CurrentThreadId_4 = In_ThreadID;
pQuery_Pass->Unknown_CurrentThreadId_5 = PsGetCurrentThreadId();
pQuery_Pass->Hash[0] = uHash[0];
pQuery_Pass->Hash[1] = uHash[1];
pQuery_Pass->Hash[2] = uHash[2];
pQuery_Pass->Hash[3] = uHash[3];
pQuery_Pass->Unknown_Flag_6 = 1;
pQuery_Pass->Unknown_Flag_830 = In_Flag;
pQuery_Pass->FileSize = FileSize;
pQuery_Pass->Unknown_Flag_2 = 2;
//字符串最大长度应该不超过520吧，一般大于520个长度的路径直接无视算了
if (In_ImagePathString->Length < 520)
{
    RtlCopyMemory(pQuery_Pass->ImagePathBuff, In_ImagePathString->Buffer, In_ImagePathString->Length);
}
//发送消息给R3
//Flag_v16 = Safe_push_request_in_and_waitfor_finish(pQuery_Pass, 1);
//根据R3的返回值做出对应操作
if (Flag_v16 == 0)              //添加白名单进程列表，并正常返回
{
    Safe_InsertDrvmkDataList(uHash, FileSize, 0);
    result = STATUS_SUCCESS;
}
else if (Flag_v16 == 1)         //添加黑名单进程列表，并返回错误值
{
    Safe_InsertDrvmkDataList(uHash, FileSize, 1);
    result = STATUS_ACCESS_DENIED;
}
else if (Flag_v16 == 2)         //直接返回错误值
{
    result = STATUS_ACCESS_DENIED;
}
else                            //直接正常返回
{
```

总结：

1、配合NtRequestWaitReplyPort or ZwAlpcSendWaitReceivePort 获取真实的加载驱动意图者

2、判断驱动信息是否在驱动黑白名单

3、新驱动第一次加载交给用户决定放行 or 拦截

4、Fake_ZwOpenProcess

参考资料：
1、2.4.3 ACCESS_MASK
2、ACCESS_MASK
3、Process and Thread Functions
4、Ring3注入总结及编程实现【有码】

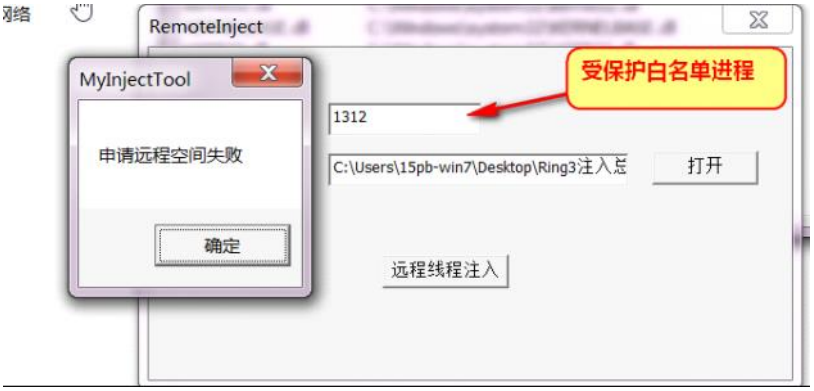
预备知识：
远程线程注入流程（打开进程是受保护进程）：
1、高权限打开进程（阉割权限）
OpenProcess
2、分配内存（GG）
VirtualAllocEx
3、在远程申请的地址当中写入DLL的路径（第二步已经凉了）
WriteProcessMemory
4、创建远程线程（第二步已经凉了）
CreateRemoteThread

远程线程注入流程（打开进程是非受保护进程）：
1、高权限打开进程（放行，不阉割权限）
OpenProcess
2、分配内存（放行）
VirtualAllocEx
3、在远程申请的地址当中写入DLL的路径（放行）
WriteProcessMemory
4、创建远程线程（拦截）
CreateRemoteThread

序号：0x2F
检查函数类型：属于第三类，重点在于返回值
设计思路：
1、对以高权限打开的白名单进程句柄进行阉割降权操作，从根源处解决问题
2、根源处解决远程线程注入OpenProcess降权后，后续一切操作都无法进行

防止误伤过滤：
1、必须是用户层调用
2、调用者是保护进程直接放行，不触发调用后检查

效果图（打开受保护白名单进程时）：
Copy句柄时候已经降权
~DesiredAccess_Flag=0x520D0BAF or 0x720D0BAF
//取反的话失去了比较重要的权限如下：
//DesiredAccess_Flag=0xADF2F450
//1、PROCESS_VM_OPERATION　　//操作进程内存空间的权限(可用VirtualProtectEx和WriteProcessMemory)
//2、PROCESS_VM_WRITE　　//读取进程内存空间的权限，可使用WriteProcessMemory
//3、创建线程、进程之类的
PROCESS_VM_OPERATION权限没了自然VirtualProtectEx就失败了



函数执行前判断：

1、敏感权限触发基本都会触发调用后检查（除了调用者是保护进程）

2、打开某进程包含PROCESS_TERMINATE（结束进程权限）继续判断

3、普通进程直接放行

```
if (!g_Win2k_AF_2000_Flag)
{
    DesiredAccess_Flag |= GENERIC_EXECUTE;  //0x720D0BAF
}
//3、判断权限
if (DesiredAccess_Flag & In_DesiredAccess)
{
    //判断是不是保护进程调用ZwOpenProcess，如果是直接返回，不是继续判断
    if (!Safe_QueryWhitePID(PsGetCurrentProcessId()))
    {
        if (In_DesiredAccess & PROCESS_TERMINATE)
        {
            if (In_ClientId)
            {
                //判断参数合法性
                if (myProbeRead(In_ClientId, sizeof(CLIENT_ID), sizeof(ULONG)))
                {
                    KdPrint(("ProbeRead(Fake_ZwOpenProcess：In_ClientId) error \r\n"));
                    return result;
                }
                //获取ETHREAD / EPROCESS
                if (In_ClientId->UniqueThread)
                {
                    //千万别忘记释放ObfDereferenceObject(ClientProcess)和ObfDereferenceObject(ClientThread)
                    Status = PsLookupProcessThreadByCid(&In_ClientId, &ClientProcess, &ClientThread);
                }
                else
                {
                    Status = PsLookupProcessByProcessId(In_ClientId->UniqueProcess, &ClientProcess);
                }
                //获取失败直接退出
                if (!NT_SUCCESS(Status))
                {
                    *(ULONG*)ret_func = After_ZwOpenProcess_Func;
                    return result;
                }
```

3、当你打开受保护进程直接修改DesiredAccess权限去掉PROCESS_TERMINATE属性，并且对几个特殊进程直接放行

设置单独PROCESS_TERMINATE属性的直接替换成默认属性

```
}
//受保护进程有几个需要单独处理权限(这几个进程不在白名单列表中)
if (ClientThread)
{
    ObfDereferenceObject(ClientThread);
}
if (!Safe_InsertSystemInformationList(IoGetCurrentProcess(), SYSTEMROOT_SYSTEM32_CSRSS_EXE, g_VersionFlag))
{
    //获取要打开句柄的路径，判断是不是大数字的
    Safe_PsGetProcessImageFileName(ClientProcess, &ImageFileNameBuff, sizeof(ImageFileNameBuff));
    //遇到[redacted]的进程直接放行,其他保护进程设置对应的权限
    for (ULONG i = 0; i < WHILEPROCESSNAMENUMBER_ZWOPENPROCESS; i++)
    {
        if (_stricmp(&ImageFileNameBuff, g_WhiteProcessName_ZwOpenProcess[i]) == 0)
        {
            //特殊进程直接放行
            break;
        }
        else
        {
            //保护进程进行阉割权限

            //判断是不是想结束OpenPorcess(PROCESS_TERMINATE,xx,PID)然后调用TerminateProcess结束进程
            ULONG Flag = *(ULONG*)((ULONG)ArgArray + 4) & PROCESS_TERMINATE == 0;
            //取消掉结束进程的权限
            *(ULONG*)((ULONG)ArgArray + 4) &= PROCESS_TERMINATE;
            //仅有PROCESS_TERMINATE权限，就给设置个默认权限
            if (Flag)
            {
                *(ULONG*)((ULONG)ArgArray + 4) = PROCESS_QUERY_INFORMATION;
            }
            Protection_Flag = TRUE;
        }
    }
}
ObfDereferenceObject(ClientProcess);
```

函数执行前检查总结:

1、普通进程、特殊进程权限不进行阉割，并且标志位置0（触发调用后检查）

2、受保护白名单进程权限进行阉割权限，并且标志位置1（触发调用后检查）

3、调用者是保护进程直接放行（不触发调用后检查）

函数执行后判断:

1、原始权限取反后基本分配内存、创建进程线程、远程写内存就和你说拜拜了

```
PCLIENT_ID      In_ClientId = *(ULONG*)((ULONG)ArgArray + 0x8);
//1、判断上次调用原始函数返回值
if (!NT_SUCCESS(InResult))
{
    return InResult;
}
//2、低版本权限
if (!g_Win2K_XP_2003_Flag)
{
    DesiredAccess_Flag |= GENERIC_EXECUTE;  //0x720D0BAF;
}
DesiredAccess_Flag = ~DesiredAccess_Flag;
//取反的话失去了比较重要的权限如下：
//DesiredAccess_Flag=0xADF2F450
//1、PROCESS_VM_OPERATION       //操作进程内存空间的权限(可用VirtualProtectEx和WriteProcessMemory)
//2、PROCESS_VM_WRITE           //读取进程内存空间的权限，可使用WriteProcessMemory
//3、创建线程、进程之类的
```

2、非保护进程直接放行（不阉割权限）

3、受保护进程重新Copy一份低权限句柄（原始句柄删除）

4、任务管理器结束特殊进程权限阉割后保留PROCESS_TERMINATE属性（原始句柄删除）

```
}
else if ( a4 == 1 || a2->DesiredAccess & 1 )// 保护进程 or 带有结束进程标志
{
  if ( Safe_CmpImageFileName(dword_262C0 + 2) )
  {
    v9 = ObReferenceObjectByHandle(*a2->ProcessHandle, 0, PsProcessType, 1, &pPeprocess, 0);
    if ( v9 >= 0 )
    {
      Safe_PsGetProcessImageFileName(pPeprocess, &v15);
      if ( !stricmp(&v15, &dword_262C0[3] + 2)
        || !stricmp(&v15, &dword_262C0[7] + 2)
        || !stricmp(&v15, &dword_262C0[11] + 2)
        || !stricmp(&v15, &dword_262C0[15] + 2) )
      {
        v9 = ZwDuplicateObject(
               0xFFFFFFFF,
               *a2->ProcessHandle,
               0xFFFFFFFF,
               &v11,
               a2->DesiredAccess & ~v4 | 1,
               0,
               0);                          // 将要打开的进程降权并加上结束进程的标识
        if ( v9 >= 0 )
        {
          ObfDereferenceObject(pPeprocess);
          Safe_ZwNtClose(*a2->ProcessHandle);
          v6 = v11;
          goto LABEL_9;
        }
      }
      ObfDereferenceObject(pPeprocess);
    }
  }
  if ( ZwDuplicateObject(0xFFFFFFFF, *a2->ProcessHandle, 0xFFFFFFFF, &v10, a2->DesiredAccess & ~v4, 0, 0) >= 0 )// 将要打开的进程降权
  {
    Safe_ZwNtClose(*a2->ProcessHandle);
    v6 = v10;
    goto LABEL_9;
  }
}
Safe_ZwNtClose(*a2->ProcessHandle);
*a2->ProcessHandle = 0;
v14 = 0xC0000022;
```

判断父进程是不是特殊进程：任务管理器

那几个特殊进程阉割权限后保留PROCESS_TERMINATE属性

受保护白名单进程走这里

释放原始高权限句柄，替换成低权限（阉割）

函数执行后检查总结:

1、受保护白名单进程权限被阉割（远程线程第二步分配内存就已经GG了）

2、非保护进程权限不影响（大数字这里并没有处理，后续CreateThread拦截）

5、Fake_ZwAllocateVirtualMemory

参考资料：

1、总结一把，较为精确判断SCM加载

2、如何判断一个线程的状态

3、R0挂钩NtResumeThread如何判断是否为新进程

4、科锐三阶段项目---CreateProcess流程分析

5、《Windows 内核情景分析 上》5.6 Windows的进程创建和映像装入

序号：0x4E

检查函数类型：

设计思路：

1、非保护进程直接放行

2、句柄是受保护进程直接返回错误值

3、将受保护进程初次创建进程的内存信息保存到列表中

防止误伤过滤：

1、必须是用户层调用

2、调用者是保护进程直接放行，不触发调用后检查

3、进程创建也会调用为了防止误伤，判断该句柄线程!=1（因为每个进程都会有1个初始线程）

图1（运行中的进程，线程个数不可能是1）：

```
   +0x160 PageDirectoryPte : _HARDWARE_PTE_X86
   +0x160 Filler           : 0
   +0x168 Session          : 0x9b449000 Void
   +0x16c ImageFileName    : [15] "ZhuDongFangYu."
   +0x17b PriorityClass    : 0x2 ''
   +0x17c JobLinks         : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x184 LockedPagesList  : (null)
   +0x188 ThreadListHead   : _LIST_ENTRY [ 0x89106ce0 - 0x87e77e18 ]
   +0x190 SecurityPort     : (null)
   +0x194 PaeTop           : 0x884db240 Void
   +0x198 ActiveThreads    : 0x15
   +0x19c ImagePathHash    : 0xd05972c2
   +0x1a0 DefaultHardErrorProcessing : 0
   +0x1a4 LastThreadExitStatus : 0n0
   +0x1a8 Peb              : 0x7ffd9000 _PEB
```

图2（刚创建进程，线程个数是1）：

```
   +0x168 Session          : 0x9b449000 Void
   +0x16c ImageFileName    : [15] "ZhuDongFangYu."
   +0x17b PriorityClass    : 0x2 ''
   +0x17c JobLinks         : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x184 LockedPagesList  : (null)
   +0x188 ThreadListHead   : _LIST_ENTRY [ 0x89106ce0 - 0x89106ce0 ]
   +0x190 SecurityPort     : (null)
   +0x194 PaeTop           : 0x884db240 Void
   +0x198 ActiveThreads    : 1
   +0x19c ImagePathHash    : 0
   +0x1a0 DefaultHardErrorProcessing : 0
   +0x1a4 LastThreadExitStatus : 0n0
```

函数执行前判断：

1、被操作句柄是受保护进程进行判断，否则无视

```
// 函数说明:
// 1、ZwOpenProcess已经将句柄降权阉割处理过一遍了
unsigned int __stdcall Fake_ZwAllocateVirtualMemory(int a1, ULONG *a2, signed int (__cdecl **a3)(int a1, int a2, int a3), _DWORD *a4)
{
  HANDLE v4; // eax@3
  PEPROCESS v5; // eax@5
  PEPROCESS v6; // eax@6
  PEPROCESS v7; // eax@7
  PEPROCESS v8; // eax@8
  int v9; // eax@10

  if ( ExGetPreviousMode() )                    // 必须是应用层调用
  {
    if ( *a2 != 0xFFFFFFFF )
    {
      v4 = PsGetCurrentProcessId();
      if ( !Safe_QueryWhitePID(v4) )            // 调用者非白名单进程
      {
        if ( Safe_QueryWintePID_ProcessHandle(*a2) )// 判断被操作的句柄是不是保护进程的:是继续判断 不是直接无视
        {
          v5 = IoGetCurrentProcess();
          if ( !Safe_QuerySystemInformationList(v5, 0) )
          {
            v6 = IoGetCurrentProcess();
            if ( !Safe_QuerySystemInformationList(v6, 0xE) )
            {
              v7 = IoGetCurrentProcess();
              if ( !Safe_InsertSystemInformationList(v7, 0) )
              {
                v8 = IoGetCurrentProcess();
                if ( !Safe_InsertSystemInformationList(v8, 0xE) )
                {
                  if ( g_Win2K_XP_2003_Flag )
                    v9 = Safe_FindEprocessThreadCount(*a2, 1);// 进程线程个数等于1时候（刚创建时候就满足）,所以那些注入保护进程的在这里就已经GG了，因为跑起来的进
                  else
                    v9 = Safe_FindEprocessThreadCount(*a2, 0);
                  if ( !v9 )
                    return 0xC0000022;
                  *a3 = After_ZwAllocateVirtualMemory_Func;// 满足条件：创建新进程 and 受保护进程
                  *a4 = 0;
                }
              }
            }
          }
        }
      }
    }
  }
  return 0;
}
```

2、Safe_FindEprocessThreadCount函数分析

进程线程个数等于1时候（刚创建时候就满足）

注入保护进程的在这里就已经GG了，因为跑起来的进程的线程个数不可能等于1

```
//查询线程信息的
BOOLEAN NTAPI Safe_FindEprocessThreadCount(IN HANDLE In_ProcessHandle, IN BOOLEAN In_Flag)
{
	NTSTATUS Status; // eax@5
	BOOLEAN result = TRUE;
	PROCESS_BASIC_INFORMATION PBI = { 0 };
	ULONG ReturnLength = NULL;
	PSYSTEM_PROCESS_INFORMATION pInfo = NULL;
	size_t	BufLen = 4096;
	//1、获取进程PID
	Status = Safe_ZwQueryInformationProcess(In_ProcessHandle, ProcessBasicInformation, &PBI, sizeof(PROCESS_BASIC_INFORMATION), &ReturnLength);
	if (!NT_SUCCESS(Status) || !PBI.UniqueProcessId)
	{
		return result;
	}
	//2、调用API获取信息
	do
	{
		if (pInfo)
		{
			ExFreePool(pInfo);
		}
		pInfo = Safe_AllocBuff(NonPagedPool, BufLen, SELFPROTECTION_POOLTAG);
		if (!pInfo)
		{

	return result;

		}
	Status = ZwQuerySystemInformation(SystemProcessesAndThreadsInformation, pInfo, BufLen, &ReturnLength);
	if (!NT_SUCCESS(Status) && Status != STATUS_INFO_LENGTH_MISMATCH)
	{
		ExFreePool(pInfo);
		return result;
	}
	BufLen += 4096;
	} while (!NT_SUCCESS(Status));
	//3、找到进程并遍历所有线程
	while (1)
	{
		//判断是否还有下一个进程
		if (pInfo->NextEntryDelta == 0)
		{
```

```
                break;
            }
            //判断是否找到了进程ID
            if (pInfo->ProcessId == PBI.UniqueProcessId)
            {
                //线程个数
                if (pInfo->ThreadCount)
                {
                    //进程线程个数等于1时候（刚创建时候就满足）,所以那些注入保护进程的在这里就已经GG了，因为跑起来的进程的线程个数不可能等于1
                    //这里为什么会加个!In_Flag,假设In_Flag输入时0，那么就直接运算符短路不判断后续的？？？
                    if (!In_Flag || (pInfo->ThreadCount != 1) || pInfo->Threads->UserTime.QuadPart)
                    {
                        result = FALSE;
                        break;
                    }
                }
            }
            //换下一个节点
            pInfo = (PSYSTEM_PROCESS_INFORMATION)((PUCHAR)pInfo + pInfo->NextEntryDelta);
        }
        if (pInfo)
        {
            ExFreePool(pInfo);
        }
        return result;
}
```

函数执行后判断：

1、保存内存BaseAddress、Size、UniqueProcessId和父进程ID到列表中

```
  if ( a3 < 0 )
    return 0;
  ms_exc.registration.TryLevel = 0;
  Status = Safe_ZwQueryInformationProcess(*a2, 0, ProcessInformation, 0x18u, 0);
  if ( Status >= 0 )
  {
    ProbeForRead(*(a2 + 4), 4u, 1u);
    ProbeForRead(*(a2 + 0xC), 4u, 1u);
    BaseAddress = **(a2 + 4);
    RegionSize = **(a2 + 0xC);
    v3 = PsGetCurrentProcessId();
    if ( !Safe_InsertVirtualMemoryDataList(BaseAddress, RegionSize, ProcessInformation[0].UniqueProcessId, v3) )
    {
      if ( HighVersionFlag )
      {
        ZwFreeVirtualMemory(*a2, &BaseAddress, &RegionSize, 0x8000u);
      }
      else
      {
        **(a2 + 0xC) = 0;
        NtFreeVirtualMemory(*a2, *(a2 + 4), *(a2 + 0xC), 0x8000u);
      }
      **(a2 + 4) = 0;
      **(a2 + 0xC) = 0;
      result = 0xC0000022;
      goto LABEL_12;
    }
    ms_exc.registration.TryLevel = 0xFFFFFFFF;
    return 0;
  }
  result = 0;
LABEL_12:
  ms_exc.registration.TryLevel = 0xFFFFFFFF;
  return result;
```

代码疑惑的地方：

Safe_FindEprocessThreadCount的第二个参数问题

```
  {
    if ( g_Win2K_XP_2003_Flag )
      v9 = Safe_FindEprocessThreadCount(*a2, 1);// 进程线程个数等于1时候（刚创建时候就满足）,所以那些注入保护进程的在这里就已经GG了，因为跑起来的进程的线程个
    else
      v9 = Safe_FindEprocessThreadCount(*a2, 0);
    if ( !v9 )
```

假设它参数2是0，后面判断线程个数就没意义了。那么说只有参数为1才能正常使用

所以感觉很奇怪，搞不明白迷糊（旧版本代码同理）

```
          if ( !v5 )
            goto LABEL_8;
        }
        ThreadCount = v5->ThreadCount;
        if ( ThreadCount )
        {
          if ( !In_Flag || ThreadCount != 1 || *&v5->Threads[0].UserTime.QuadPart )// 进程刚创建的时候？？？？？？
          {
            v6 = 0;
            goto LABEL_9;
          }
        }
      }
      goto LABEL_8;
    }
  }
  v4 = ExAllocatePoolWithTag(0, ReturnLength, 0x206B6444u);
  P = v4;
  if ( !v4 )
    return 1;
  v5 = v4;
  if ( ZwQuerySystemInformation(SystemProcessesAndThreadsInformation, v4, ReturnLength, &ReturnLength) >= 0 )
    goto LABEL_16;
LABEL_8:
  v6 = 1;
LABEL_9:
```

!In_Flag直接成立运算符短路，就不判断后续ThreadCount线程个数？？？

6、Fake_ZwWriteVirtualMemory

参考资料

1、关于x64/Win10中Shim机制的一次调试分析过程

2、Windows Shim Engine 初探即利用

3、Windows 8的用户模式Shim Engine小探及利用

4、伪装进程过DXF读写

序号：0x1A

检查函数类型：属于第一类，函数执行前检查

设计思路：

1、这个函数看的迷迷糊糊。。。

防止误伤过滤：

1、必须是用户层调用

2、调用者是保护进程直接放行，不触发调用后检查

3、对象类型必须是Process

函数执行前判断：

1、Fake_ZwWriteVirtualMemory整理框架

```
if ( ExGetPreviousMode() )
{
  if ( a2->ProcessHandle != 0xFFFFFFFF )      // 若句柄值!=当前进程的句柄（-1），特殊处理
  {
    v4 = PsGetCurrentProcessId();
    if ( !Safe_QueryWhitePID(v4) )            // 调用者非保护进程（要检查）
    {
      if ( Safe_QueryObjectType(a2->ProcessHandle, L"Process") )// 查找指定的类型
      {
        if ( !Safe_QueryWintePID_ProcessHandle(a2->ProcessHandle)
          || (v5 = IoGetCurrentProcess(), Safe_QuerySystemInformationList(v5, 0))
          || (v6 = IoGetCurrentProcess(), Safe_QuerySystemInformationList(v6, 0xE))
          || (v7 = IoGetCurrentProcess(), Safe_InsertSystemInformationList(v7, 0))
          || (v8 = IoGetCurrentProcess(), Safe_InsertSystemInformationList(v8, 0xE)) )
        {                                     // 非白名单进程走这里
          if ( Safe_CheckSysProcess_Csrss_Lsass(a2->ProcessHandle)// 过滤掉csrss.exe和lsass.exe
            && !Safe_FindEprocessThreadCount(a2->ProcessHandle, 0) )// 判断进程句柄的线程个数
          {                                   // 执行到这里就说明（进程线程个数！=1 启动后的进程了）
            return Safe_CheckSysProcess_Coherence() != 0 ? 0 : 0xC0000022;// 调用者为coherence.exe返回1，否则0
          }
        }
        else if ( g_HookPort_Version < 0x3EB )// 无视，没有低版本的demo分析
        {
          if ( !g_Win2K_XP_2003_Flag )
            return Safe_FindEprocessThreadCount(a2->ProcessHandle, 0) != 0 ? 0 : 0xC0000503;
        }                                     // 白名单进程走这里
        else if ( !Safe_CheckWriteMemory_PEB(a2->ProcessHandle, a2->BaseAddress, a2->BufferLength) )// 写保护进程内存 or 修改PEB数据的放行？？？？？？？？
                                              // ProcessParameters（包含进程名等重要信息，可以隐藏进程之类的）
                                              // pShimData（利用Shim Engine来Dll劫持）
                                              // pContextData
        {
          v9 = PsGetCurrentThreadId();
          v10 = PsGetCurrentProcessId();
          Safe_18A72_SendR3(v10, v9, HANDLE_FLAG_PROTECT_FROM_CLOSE);
          return 0xC0000503;
        }
```

2、Safe_CheckSysProcess_Csrss_Lsass函数分析（迷糊，不知道意义何在）

1、csrss.exe和lsass.exe进程有特殊意义吗？还是经常给劫持的？

```
BOOLEAN     Result = TRUE;
NTSTATUS    Status = STATUS_SUCCESS;
PEPROCESS   pPeprocess = NULL;
UCHAR       ImageFileNameBuff[0x356] = { 0 };
Status = ObReferenceObjectByHandle(In_Handle, NULL, PsProcessType, UserMode, &pPeprocess, 0);
if (NT_SUCCESS(Status))
{
    //获取要打开句柄的路径
    Safe_PsGetProcessImageFileName(pPeprocess, &ImageFileNameBuff, sizeof(ImageFileNameBuff));
    //过滤掉csrss.exe和lsass.exe
    if (_stricmp(&ImageFileNameBuff,"csrss.exe"))           //打开句柄进程名是"csrss.exe"返回TRUE
    {
        if (_stricmp(&ImageFileNameBuff, "lsass.exe")       //打开句柄进程名非"lsass.exe"返回FALSE
            || !g_dynData->SystemInformation.Userinit_Flag  //打开句柄进程名是"lsass.exe" 并且 userinit.exe进程未启动
            && g_Win2K_XP_2003_Flag                         //低版本
            && (Safe_QuerySystemInformationList(IoGetCurrentProcess(), SYSTEMROOT_SYSTEM32_WININIT_EXE, g_VersionFlag))
            || (Safe_QuerySystemInformationList(IoGetCurrentProcess(), SYSTEMROOT_SYSTEM32_DLLHOST_EXE, g_VersionFlag))
            || (Safe_QuerySystemInformationList(IoGetCurrentProcess(), SYSTEMROOT_SYSTEM32_CSRSS_EXE, g_VersionFlag))
            || !g_dynData->SystemInformation.Userinit_Flag
            && g_Win2K_XP_2003_Flag
            && (Safe_InsertSystemInformationList(IoGetCurrentProcess(), SYSTEMROOT_SYSTEM32_WININIT_EXE, g_VersionFlag))
            || (Safe_InsertSystemInformationList(IoGetCurrentProcess(), SYSTEMROOT_SYSTEM32_DLLHOST_EXE, g_VersionFlag))
            || (Safe_InsertSystemInformationList(IoGetCurrentProcess(), SYSTEMROOT_SYSTEM32_CSRSS_EXE, g_VersionFlag)))
        {
            Result = FALSE;
        }
    }
    ObfDereferenceObject(pPeprocess);
}
else
{
    Result = FALSE;
}
```

3、Safe_CheckWriteMemory_PEB函数分析（迷惑行为？？？？？？？？？）

1、不应该是写Peb->pShimData和Peb->ProcessParameters属于高危操作应该拦截吗？我没看明白有什么玄机。。。

```
int result; // eax@2
HANDLE v4; // eax@4
int Peb_Offset; // esi@6
PROCESS_BASIC_INFORMATION ProcessInformation; // [sp+0h] [bp-18h]@1

if ( Safe_ZwQueryInformationProcess(ProcessHandle, 0, &ProcessInformation, 0x18u, 0) >= 0 )
{
  result = 0;
  if ( In_BaseAddress + In_BufferLength >= In_BaseAddress )
  {
    v4 = PsGetCurrentProcessId();
    if ( Safe_QueryVirtualMemoryDataList(In_BaseAddress, In_BufferLength, ProcessInformation.UniqueProcessId, v4)// 判断写的空间是否在列表空间之间
      || In_BaseAddress > ProcessInformation.PebBaseAddress
      && ((Peb_Offset = In_BaseAddress - ProcessInformation.PebBaseAddress,
           In_BaseAddress - ProcessInformation.PebBaseAddress == 0x10)// Peb->ProcessParameters（进程参数块包含进程路径等信息）
       || Peb_Offset == 0x1E8                // Peb->pShimData（Shim Engine这个我没仔细看，感兴趣读者自行参阅）
       || Peb_Offset == 0x238)               // Peb->pContextData（知识盲区，有知道的老大告诉下）
      && In_BufferLength == 4 )
    {
      result = 1;
    }
  }
}
else
{
  result = 1;
}
return result;
```

触发报警条件：
1、写地址非保护
2、写地址非PebBase
3、写地址非Peb执行部分
4、写缓冲区大小!=4

7、Fake_ZwCreateThread

参考资料：

1、总结一把，较为精确判断SCM加载

2、如何判断一个线程的状态

3、《Windows 内核情景分析 上》5.6 Windows的进程创建和映像装入

序号：0x1A

检查函数类型：属于第一类，函数执行前检查

设计思路：

1、注入的进程必须是已启动的，那么启动进程线程必然>1

2、进程创建会创建一个初始化线程

3、注入都是A 注入 B，不可能自己注入自己没意义

4、判断pPeprocess != IoGetCurrentProcess()，结合上面线程个数就可以基本拦截了

ObReferenceObjectByHandle(In_ProcessHandle, 2u, PsProcessType, UserMode, &pPeprocess, 0)

5、***SelfProtection.sys只拦截注入保护进程，你注入非保护进程它不管你（本地）

6、***mdrv.sys注入非保护进程都拦截（联网）

防止误伤过滤：

1、必须是用户层调用

2、调用者是保护进程直接放行

3、对象类型必须是Process

函数执行前判断：

1、注入非保护进程基本无视

2、注入保护进程拦截（前面基本拦截干净了，基本轮不到ZwCreateThread来处理）

```
  v3 = PsGetCurrentProcessId();
  if ( Safe_QueryWhitePID(v3)
    || !ExGetPreviousMode()
    || !Safe_QueryObjectType(a2->ProcessHandle, L"Process")
    || ObReferenceObjectByHandle(a2->ProcessHandle, 2u, PsProcessType, 1, &Object, 0) < 0 )
  {
    return 0;
  }
  if ( Object != IoGetCurrentProcess() )        // 判断调用者的Eprocess和被调用的Eprocess是不是同一个,可以进一步判断远程线程调用
    v2 = 1;
  ObfDereferenceObject(Object);
  if ( !v2 )
    return 0;
  if ( !Safe_CheckSysProcess_Csrss_Lsass(a2->ProcessHandle) || Safe_FindEprocessThreadCount(a2->ProcessHandle, 0) )// 判断进程的线程个数
  {
    if ( Safe_QueryWintePID_ProcessHandle(a2->ProcessHandle) )// 访问句柄为白名单进程
    {
      if ( !g_Win2K_XP_2003_Flag && Safe_FindEprocessThreadCount(a2->ProcessHandle, 0) )
      {
        ms_exc.registration.TryLevel = 0;
        ProbeForRead(a2->ThreadContext, 0x2CCu, 1u);
        In_ThreadContext = a2->ThreadContext;
        In_Eip = In_ThreadContext->Eip;
        v15 = In_ThreadContext->Eip;
        In_Eax = In_ThreadContext->Eax;
        In_Esp = In_ThreadContext->Esp;
        ProbeForRead(a2->UserStack, 0x14u, 4u);
        stack = a2->UserStack;
        ExpandableStackBottom = stack->ExpandableStackBottom;
        v14 = ExpandableStackBottom;
        ExpandableStackSize = (stack->ExpandableStackBase - ExpandableStackBottom);
        v13 = ExpandableStackSize;
        ms_exc.registration.TryLevel = 0xFFFFFFFF;
        if ( !g_Thread_Information.ThreadContext_Eip )
        {
          g_Thread_Information.ThreadContext_Eip = *(a2->ThreadContext + 0x88);
          return 0;
        }
        if ( Safe_19AFC(a2->ProcessHandle, In_Eip, In_Eax, In_Esp, ExpandableStackBottom, ExpandableStackSize) )// Win2K生效懒得逆向了
          return 0;
      }
      v10 = PsGetCurrentThreadId();
      v11 = PsGetCurrentProcessId();
      Safe_18A72_SendR3(v11, v10, HANDLE_FLAG_PROTECT_FROM_CLOSE);
      return 0xC0000022;
    }
    return 0;
  }
  if ( Safe_CheckSysProcess_Coherence() )
```

低版本的无视，没去分析

访问保护进程句柄错误返回
基本不可能走到这里，前面Open之类的已经拦截完了
1、OpenProcess（降权）
2、ZwAllocateVirtualMemory（检查线程个数）

非保护进程这里返回

8、Fake_ZwAlpcSendWaitReceivePort

参考资料：
1、总结一把，较为精确判断SCM加载
2、监视windows服务创建的问题
3、拦截LPC监控服务加载和启动
4、如何得到与services.exe的通信数据
5、编写驱动加载程序
6、Service Security and Access Rights
7、Windows服务管家婆之Service Control Manager
8、关于个人防火墙的真相
9、LPC 简单介绍

预备知识：
SSDT上hook NtSetSystemInformation NtLoadDriver，可以拦截应用层装载驱动的操作.如果怕不保险，还可以PsSetLoadImageNotifyRoutine或者hook NtCreateSection。但是常规通过服务API加载驱动时却不能找到加载驱动的源程序，PsGetCurrentProcess程序已经是services.exe了.
原因是StartService这个API将服务的相关信息发送给了services.exe，services.exe内部统一管理注册表中的服务信息数据，其中就包括启动一个驱动服务。想要拦截这个操作需要拦截发送消息的Native API，就是进程间通信机制LPC相关的api。

序号：0x44
检查函数类型：属于第一类，函数执行前检查
设计思路：
1、PORT_MESSAGE后面紧跟一个通信附加数据（核心部分）
2、NtAlpcSendWaitReceivePort（Win7版本）调用非常频繁，需要自行过滤感兴趣的RPC拦截
3、例如我想拦截创建服务、启动服务相关的就services建立的服务端口名是\\RPC Control\\ntsvcs
4、拦截到启动服务（StartServiceW || StartServiceA）记录进程PID和线程ID,为LoadDriver做准备
5、OpenService打开敏感驱动服务直接修改权限去掉SERVICE_START，后续就无法使用StartService
6、其他不感兴趣的略，读者感兴趣自行逆向

防止误伤过滤：
1、必须是用户层调用

疑感的地方：
PORT_MESSAGE后面紧跟一个通信附加数据，这个结构体是如何定义的？懂得老哥告诉下

```
typedef struct _PORT_MESSAGE {
    union {
        struct {
            CSHORT DataLength;
            CSHORT TotalLength;
        } s1;
        ULONG Length;
    } u1;
    union {
        struct {
            CSHORT Type;
            CSHORT DataInfoOffset;
        } s2;
        ULONG ZeroInit;
    } u2;
    union {
        LPC_CLIENT_ID ClientId;
        double DoNotUseThisField;       // Force quadword alignment
    };
    ULONG MessageId;
    union {
        LPC_SIZE_T ClientViewSize;          // Only valid on LPC_CONNECTION_REQUEST message
        ULONG CallbackId;                   // Only valid on LPC_REQUEST message
    };
//  UCHAR Data[];
} PORT_MESSAGE, *PPORT_MESSAGE;
```

重要公式：
根据大数字源码得到一组公式：

服务类型	InSendMessage（参数3） + MessageType_Offset
驱动长度	InSendMessage（参数3） + ServiceName_Offset + 0x4C（固定值）
驱动名字	InSendMessage（参数3） + ServiceName_Offset + 0x58（固定值）
OpenServiceA的dwDesiredAccess：	(InSendMessage（参数3） + ServiceName_Offset + 0x5B（固定值） + 驱动长度(A)) & 0xFFFFFFFC
OpenServiceW的dwDesiredAccess：	InSendMessage（参数3） + ServiceName_Offset + 0x5B（固定值） + 驱动长度(W * 2) & 0xFFFFFFFC

服务类型：

API	Type（通用）	Type（A版本）	Type（2A版本）	Type（W版本）	Type（2A版本）	Type（Ex版本）	Type（ExA版本）	Type（ExW版本）
CloseServiceHandle	0x0							
ControlService	0x1						0x32	0x33
DeleteService	0x2							
QueryServiceStatus	0x6					0x28		
ChangeServiceConfig		0x17		0xB				
CreateService		0x18		0xC				
OpenSCManager		0x1B		0xF				
OpenService		0x1C		0x10				
StartService		0x1F		0x13				

函数执行前判断：

1、根本版本获取不同的偏移量

2、根据MessageType区分启动服务、创建服务、打开服务等操作

```
ServiceName_Offset = 0;
if ( VersionFlag == 2 )                         // WinXp
{
  AppendData_Offset = 0x20;
  MessageType_Offset = 0x1E;
}
else
{
  if ( !g_Win2K_XP_2003_Flag )
    return 0;
  AppendData_Offset = 0x2E;
  MessageType_Offset = 0x2C;
  ServiceName_Offset = 0x20;
}
MessageType_Offset_1 = MessageType_Offset;
if ( Safe_CmpPortName(PortHandle, &RPCControl_ntsvcs) )// services建立的服务端口名是 \\RPC Control\\ntsvcs
{
  ms_exc.registration.TryLevel = 0;
  ProbeForRead(SendMessage, AppendData_Offset, 1u);
  MessageType = *(SendMessage + MessageType_Offset_1);
  MessageType_1 = MessageType;
  v5 = *(SendMessage + MessageType_Offset_1 - 2);
  v28 = *(SendMessage + MessageType_Offset_1 - 2);
  if ( !g_Win2K_XP_2003_Flag && (v5 & 0x240) != 0x240 )// 低版本，就额外判断
    goto LABEL_86;
  if ( MessageType == 0x13 || MessageType == 0x1F )// StartServiceW(13) || StartServiceA(1F)
  {
    SourceDrivenLoad_CurrentProcessId = PsGetCurrentProcessId();
    SourceDrivenLoad_CurrentThreadId = PsGetCurrentThreadId();
    goto CheckSys;
  }
  if ( MessageType == 0x10 )                    // OpenServiceW
  {
    Safe_RemoveDesiredAccess_OpenService(SendMessage, ServiceName_Offset, 0);// 访问敏感驱动服务路径进行降权处理
    goto CheckSys;
  }
  if ( MessageType == 0x1C )                    // OpenServiceA
  {
    Safe_RemoveDesiredAccess_OpenService(SendMessage, ServiceName_Offset, 1);
    goto CheckSys;
  }
  if ( MessageType == 1 )                       // ControlService
```

3、Safe_RemoveDesiredAccess_OpenService函数

调用者非白名单OpenService打开敏感驱动路径，修改对应权限

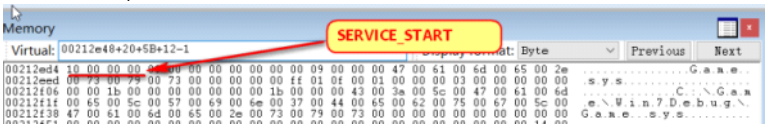
取反失去了非常重要的属性SERVICE_START，后面StartService就无法执行

```
v12 = PsGetCurrentProcessId();
result = Safe_QueryWhitePID(v12);
if ( !result )                                  // 调用者非白名单
{
  result = a2 + a1;
  if ( *(a2 + a1 + 0x4C) == v11 )               // 比较驱动名称长度
  {
    v15 = (result + 0x58);                      // 驱动名称地址
    result = a3 ? strnicmp(v15, v9, v11) : wcsnicmp(v15, v9, v11);
    if ( !result )
    {
      v14 = ((a1 + v11 * ((a3 == 0) + 1) + a2 + 0x5B) & 0xFFFFFFFC);// 去掉末尾4个字节内容
      ProbeForWrite(v14, 4u, 1u);
      result = *v14;                            // 0x60000000, GENERIC_WRITE | GENERIC_EXECUTE
      if ( *v14 & 0x610D0122 )                  // 0x01000000, ACCESS_SYSTEM_SECURITY
                                                // 0x000D0000, WRITE_OWNER | WRITE_DAC | DELETE
                                                // 0x00000100, SERVICE_USER_DEFINED_CONTROL
                                                // 0x00000020, SERVICE_STOP
                                                // 0x00000002, SERVICE_CHANGE_CONFIG
      {
        result &= 0x9EF2FEDD;                   // 取反失去了非常重要的属性SERVICE_START，后面StartService就无法执行
        *v14 = result;
      }
      if ( !*v14 )
        *v14 = 4;                               // SERVICE_QUERY_STATUS
    }
  }
}
return result;
```

对照

```
Msg(_T("启动中"));
m_hServiceDDK = OpenService(sh, DriverName.c_str(), SERVICE_START);
```

套用公式定位到OpenService的参数dwDesiredAccess



4、低版本，MessageType==0x1D是哪个API没有仔细分析过

这个函数跟LoadDriver基本一样，加载驱动相关的

```
if ( !g_Win2K_XP_2003_Flag && MessageType_1 == 0x10 )
{
  ProbeForRead(SendMessage, 0x50u, 1u);
  v17 = *(SendMessage + 0x13);
  v39 = v17;
  if ( v17 & 1 )
    v39 = ++v17;
  if ( v17 >= 0x104 )
    goto LABEL_86;
  ProbeForRead(SendMessage, 2 * v17 + 0x50, 1u);
  v18 = v39;
  v19 = SendMessage + 0x50;
  v20 = &SourceString;
  while ( v18 )
  {
    *v20 = *v19;
    v19 += 2;
    ++v20;
    --v18;
  }
  v45 = 0;
  RtlInitUnicodeString(&DestinationString, &SourceString);
  v41 = Safe_ZwOpenKey(&Handle, 0, &unk_3311C, &loc_20018 + 1);
  if ( v41 < 0 )
    goto LABEL_86;
  v41 = Safe_ZwOpenKey(&KeyHandle, Handle, &DestinationString, &loc_20018 + 1);
  ZwClose(Handle);
  if ( v41 < 0 )
    goto LABEL_86;
  v41 = Safe_GetKeyValueFullInformation_0(KeyHandle, L"Service", &P);
  ZwClose(KeyHandle);
  if ( v41 < 0 )
    goto LABEL_86;
  if ( !*(P + 3) || (v41 = Safe_ZwOpenKey(&v33, 0, &unk_33F48, &loc_20018 + 1), v41 < 0) )
  {
    v26 = P;
    goto LABEL_69;
  }
  RtlInitUnicodeString(&v27, (P + *(P + 2)));
  v41 = Safe_ZwOpenKey(&v30, v33, &v27, &loc_20018 + 1);
  ZwClose(v33);
  ExFreePool(P);
  if ( v41 >= 0 )
  {
    v21 = PsGetCurrentThreadId();
    v22 = PsGetCurrentProcessId();
    result = Safe_CheckSys(v30, v22, v21, 4);
```

5、其他不感兴趣的略

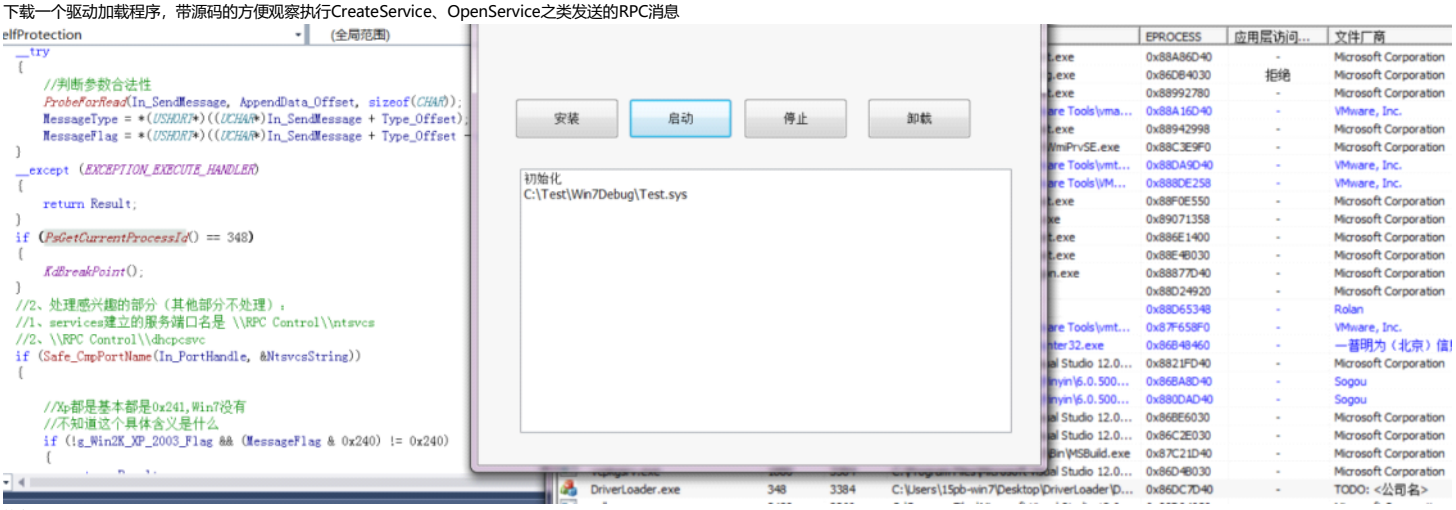
略

调试笔记：

测试部分Win7（网上资料是XP的，需要微调）：

VS动态调试驱动加载程序，执行到特定API函数就会触发RPC消息（因为这个API触发太频繁了）

下载一个驱动加载程序，带源码的方便观察执行CreateService、OpenService之类发送的RPC消息



执行OpenService

```
#02 ff
ChildEBP RetAddr
b222fad8 cff1640f 360SelfProtection!Safe_RPCDispatcher(void * In_SendMessage = 0x
b222fat8 cff0945f 360SelfProtection!Fake_ZwAlpcSendWaitReceivePort(unsigned long
b222fb34 8720a998 HookPort!HookPort_DoFilter(unsigned long CallIndex = 0x44, void
WARNING: Frame IP not in any known module. Following frames may be wrong.
b222fc0c 840791ea 0x8720a998
b222fc0c 77b170b4 nt!KiFastCallEntry+0x12a
002fe3dc 77b15424 ntdll!KiFastSystemCallRet
002fe3e0 7622f499 ntdll!ZwAlpcSendWaitReceivePort+0xc
002fe414 7622fd37 RPCRT4!LRPC_CASSOCIATION::AlpcSendWaitReceivePort+0x50
002fe460 7622fc35 RPCRT4!LRPC_BASE_CCALL::DoSendReceive+0xab
002fe484 76217a0d RPCRT4!LRPC_BASE_CCALL::SendReceive+0x36
002fe494 7622fbbb RPCRT4!LRPC_CCALL::SendReceive+0x25
002fe4ac 7622fb68 RPCRT4!I_RpcSendReceive+0x29
002fe4c0 7622fb36 RPCRT4!NdrSendReceive+0x31
002fe4d0 76285753 RPCRT4!NdrpSendReceive+0xc
002fe8e8 7648a37d RPCRT4!NdrClientCall2+0x1a6
002fe900 76487171 sechost!ROpenServiceW+0x19
002fe94c 006fd534 sechost!OpenServiceW+0x26
002fea70 00726d89 DriverLoader+0x31d534
```

服务类型：InSendMessage（参数3） + MessageType_Offset

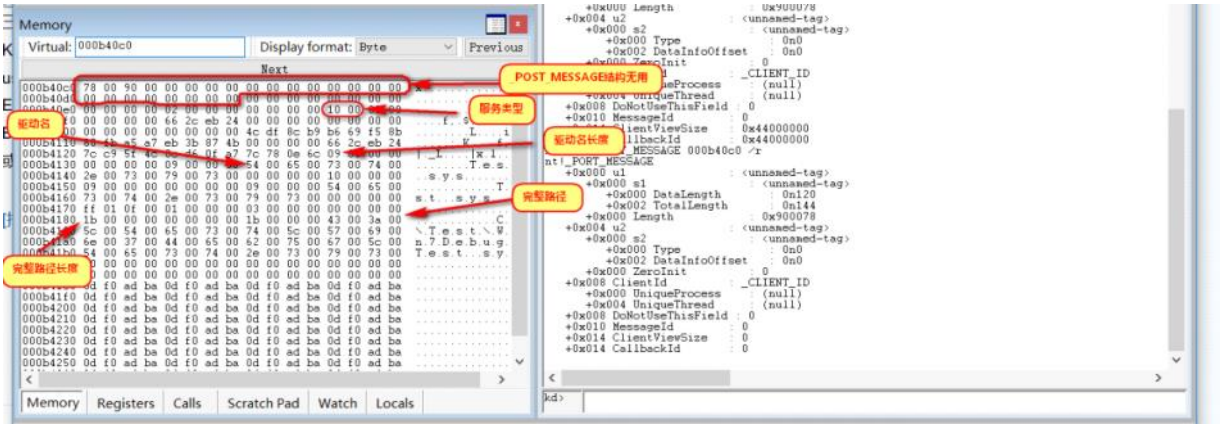
000b40c0+0x2C = 0x10 （ OpenServiceW）

驱动长度：InSendMessage（参数3） + ServiceName_Offset + 0x4C（固定值）

000b40c0+20+4C= 0x9

驱动名字：InSendMessage（参数3） + ServiceName_Offset + 0x58（固定值）

000b40c0+20+58= 00650054 00740073 0073002e 00730079



这样就产生了一个疑问：

第1个疑问：

PORT_MESSAGE后面紧跟一个通信附加数据，这个结构体是如何定义的?

第2个疑问：

驱动名称长度xxx+xxx+0x4C是一个64位类型的吧?

驱动名称xxx+xxx+0x58是一个UNICODE_STRING类型吧，前面那个0x9应该也是一起的？？？

合成一个UNICODE_STRING Len=0x9 buff=Test.sys？？？

我看大数字就是直接+0x58，并不是+0x54很疑惑的地方？？？

```
if ( !result )
{
  result = a2 + a1;
  if ( *(a2 + a1 + 0x4C) == v11 )            // 比较驱动名称长度
  {
    v15 = (result + 0x58);                   // 驱动名称地址
    result = a3 ? strnicmp(v15, v9, v11) : wcsnicmp(v15, v9, v11);
    if ( !result )
    {
```

第3个疑问：

TotalLength =0x90; //消息完整大小

0xB40C0+0X90=0xB4150，那么0xB4180那些完整路径长度又是其他的结构？？？？？

希望懂得大佬能告诉我下

测试部分XP：

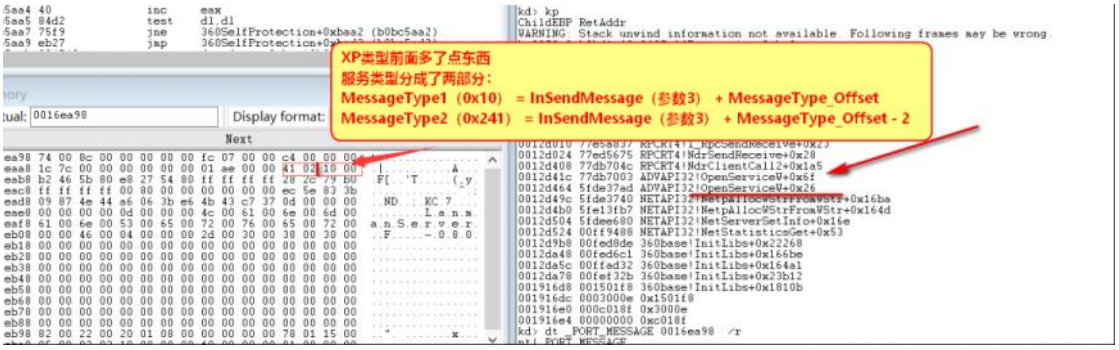
服务类型拆成两个部分判断，这个我也不明白是为什么，懂得老哥告诉下

服务类型：InSendMessage（参数3） + MessageType_Offset

16EA98+0x1E= 0x10（ OpenServiceW）

InSendMessage（参数3） + MessageType_Offset - 2

16EA98+0x1C= 0x241



9、Fake_ClientLoadLibrary

参考资料：
1、一种绕过全局钩子安装拦截的思路
2、处女贴ring3改KernelCallbackTable防键盘钩子
3、windows全局消息钩子的一个BUG
4、重读老文章：DLL注入的又一个梗
5、KeUserModeCallback用法详解
6、加密解密4

预备知识（来自MJ0011帖子）：
绕过安全软件的全局钩子拦截了:

(1).length 传入0，直接绕过检测

(2).buffer为c:\windows\system32\msctf.dllx

length传递为该字符串字节长度-sizeof(wchar)

也就是让安全软件得到错误的系统DLL名，实际注册的是其他的文件

重要结构体：

```
typedef struct _CLientLoadLibraryParam（V校贴出的）
{
	ULONG dwSize;//+0		== InputLength
	ULONG dwStringLength; //+4
	ULONG ReservedZero1;//+8
	ULONG ReservedZero2;//+C
	ULONG ReservedZero3;//+10
	ULONG ReservedZero4;//+14
	USHORT Length;		//0x018
	USHORT MaximumLength;	//0x01A
	ULONG ptrDllString;//+1C	//偏移需要+基地址
	ULONG ReservedZero6;//+20
	ULONG ptrApiString;//+24	//偏移需要+基地址
	WCHAR szDllName[MAX_PATH];
	WCHAR szApiName[MAX_PATH];
}CLientLoadLibraryParam, *PCLientLoadLibraryParam;
```

序号：0x4B
检查函数类型：属于第一类，函数执行前检查

设计思路：
1、Length重新计算长度
2、重点检查msctf.dll防止恶意修改做校验

函数执行前判断：
1、重新计算长度，检查buff为msctf.dll的

```
        }
        KdBreakPoint();
        //设置要比较的DLL字符串信息
        pCLientLoadLibraryParam = (PCLientLoadLibraryParam)In_InputBuffer;
        ptrDllString.MaximumLength = pCLientLoadLibraryParam->MaximumLength;
        ptrDllString.Buffer = ((ULONG)In_InputBuffer + (ULONG)pCLientLoadLibraryParam->ptrDllString);        //基地+偏移
        ptrDllString.Length = 2 * wcslen(ptrDllString.Buffer);
        if (ptrDllString.Length >= MsctfDllPathMinSize)
        {
            //参考MJ0011
            //防止修改msctf绕过安全软件的全局钩子拦截
            if (_wcsnicmp((PWSTR)((CHAR *)ptrDllString.Buffer + (ptrDllString.Length - MsctfDllPathMinSize)), L"msctf.dll", MsctfDllSize) == 0)
            {
                if (wcschr(ptrDllString.Buffer, '\\'))
                {
                    //构造 //??//通配符
                    RtlCopyMemory(szDllName, Wildcard, wcslen(Wildcard) * 2);
                    RtlCopyMemory((PVOID)(szDllName + 4), ptrDllString.Buffer, ptrDllString.Length);
                    RtlInitUnicodeString(&DestinationString, (PCWSTR)szDllName);
                }
                else
                {
                    //构造\\SystemRoot\\system32通配符
                    RtlCopyMemory(szDllName, SystemRootWildcard, wcslen(SystemRootWildcard) * 2);
                    RtlCopyMemory((PVOID)(szDllName + 0x14), ptrDllString.Buffer, ptrDllString.Length);
                    RtlInitUnicodeString(&DestinationString, (PCWSTR)szDllName);
                }
                //以上算防止恶意修改做校验
```

2、校验文件信息防止被修改

```
            }
            //检查运行过程中DLL是否被修改(msctf.dll)
            if ((System_InformationFile.IndexNumber_LowPart == g_System_InformationFile_Data[SYSTEMROOT_SYSTEM32_MSCTF_DLL].IndexNumber_LowPart) &&
                (System_InformationFile.VolumeSerialNumber == g_System_InformationFile_Data[SYSTEMROOT_SYSTEM32_MSCTF_DLL].VolumeSerialNumber) &&
                (System_InformationFile.u.IndexNumber_HighPart == g_System_InformationFile_Data[SYSTEMROOT_SYSTEM32_MSCTF_DLL].u.IndexNumber_HighPart)
                )
            {
                //成功返回
                return STATUS_SUCCESS;
            }
            else
            {
                //失败返回，msctf.dll被恶意修改
                return STATUS_ACCESS_DENIED;
            }
        }
    }
```

10、Fake_ZwUnmapViewOfSection

参考资料：

1、傀儡进程技术实现

前言：

傀儡进程调用的函数，用来释放掉原始函数空间

序号：0x46

检查函数类型：属于第一类，函数执行前检查

代码逻辑：通用框架（调用者检查，以后不感兴趣Fake函数都可以这样处理 省事）

```
//1、必须是应用层调用
	if (ExGetPreviousMode())
	{
		//假设是取消映射的是白名单进程直接错误返回，并通知用户拦截还是放行
		if ((In_ProcessHandle != NtCurrentProcess()) &&
			(!Safe_QueryWhitePID(PsGetCurrentProcessId())) &&	//判断是不是白名单调用，如果是放行 不是继续判断
			(NT_SUCCESS(Safe_ZwQueryInformationProcess(In_ProcessHandle, ProcessBasicInformation, &PBI,
			sizeof(PROCESS_BASIC_INFORMATION), &ReturnLength))) &&	//获取进程PID
			Safe_QueryWhitePID(PBI.UniqueProcessId)			//判断要操作的PID是不是白名单，如果是拦截 不是放行
			)
		{
			//触发拦截还是放行
			略
			//失败返回
			result = STATUS_ACCESS_DENIED;
		}
		else
		{
			result = STATUS_SUCCESS;
		}
	}
	return result;
```

设计思路：

1、白名单调用放行，否则检查

2、要操作的是白名单直接拦截

3、发送给用户层处理

函数执行前判断：

```
if ( ExGetPreviousMode()
  && *a2 != 0xFFFFFFFF
  && (v4 = PsGetCurrentProcessId(), !Safe_QueryWhitePID(v4))// 判断是不是保护进程调用
  && Safe_ZwQueryInformationProcess(*a2, 0, ProcessInformation, 0x18u, &ReturnLength) >= 0
  && Safe_QueryWhitePID(ProcessInformation[0].UniqueProcessId) )// 判断要访问的是不是保护进程
{
  v5 = PsGetCurrentThreadId();
  v6 = PsGetCurrentProcessId();
  Safe_18A72_SendR3(v6, v5, HANDLE_FLAG_PROTECT_FROM_CLOSE);
  result = 0xC0000022;
}
else
{
  result = 0;
}
return result;
}
```

11、Fake_ZwOpenSection

参考资料：

1、突破HIPS的防御思路之duplicate physical memory

2、续PhysicalMemory攻击

前言：

应用程序用ZwOpenSection打开"\Device\PhysicalMemory"访问物理，所以函数执行前检查是非常困难的（MJ文章就解释过了），但是通过调用后检查就简单多了。根据得到的句柄查询对象名称，如果是"\Device\PhysicalMemory"直接结束句柄并返回错误值。

序号：0x1F

检查函数类型：属于第三类，函数执行后检查

设计思路：

1、高权限的触发检查机制

2、调用者是白名单放行，否则触发调用后检查

3、调用后检查\\Device\\PhysicalMemory和\\KnownDlls\\

4、访问\\Device\\PhysicalMemory直接句柄清零，错误返回

5、访问\\KnownDlls\\

句柄copy成功，降权去除DELETE权限并与R3通讯，成功返回

句柄copy失败，句柄清零，错误返回

函数执行前判断：

1、判断DesiredAccess是否包含SECTION_MAP_WRITE

2、调用者非白名单触发调用后检查

```
    //Section Access Rights
    ACCESS_MASK    DesiredAccess_Flag =                                          //0x52010002
        (GENERIC_WRITE | GENERIC_ALL) |                                          //0x50000000 = GENERIC_W
        (MAXIMUM_ALLOWED) |                                                      //0x02000000 = MAXIMUM_A
        (DELETE) |                                                               //0x00010000 = DELETE
        (SECTION_MAP_WRITE);                                                     //0x00000002 = SECTION_M

    //0、获取ZwOpenSection原始参数
    ACCESS_MASK    In_DesiredAccess = *(ULONG*)((ULONG)ArgArray + 4);
    //1、必须是应用层调用
    if (ExGetPreviousMode())
    {
        //检查高权限操作的
        if (In_DesiredAccess & DesiredAccess_Flag)
        {
            //调用者非保护进程，需要二次判断
            if (!Safe_QueryWhitePID(PsGetCurrentProcessId()))
            {
                //触发调用后检查
                *(ULONG*)ret_func = After_ZwOpenSection_Func;
            }
        }
    }
    return result;
}
```

函数执行后判断：

1、访问敏感路径直接结束句柄并返回错误值。

```
    }
    //2、1 PhysicalMemory检查
    if (!_wcsnicmp(pPubObjTypeInfo->TypeName.Buffer, L"\\Device\\PhysicalMemory", PhysicalMemorySize))
    {
        //句柄清零，禁止访问
        Safe_ZwNtClose(*(HANDLE*)SectionHandle, g_VersionFlag);
        *(HANDLE*)SectionHandle = 0;
        //Safe_18A72_SendR3(PsGetCurrentProcessId(), PsGetCurrentThreadId(),0x8);
        result = STATUS_ACCESS_DENIED;
    }
    //2、2 KnownDlls检查
    else if (!_wcsnicmp(pPubObjTypeInfo->TypeName.Buffer, L"\\KnownDlls\\", KnownDllsSize))
    {
        //2、3 获取该句柄原始权限
        Status  = Safe_GetGrantedAccess(*(HANDLE*)SectionHandle, &Out_GrantedAccess);
        if (NT_SUCCESS(Status))
        {
            //2、4 拷贝句柄并且降权,删掉DELETE权限
            //0xFFFEFFFF = 16  DELETE  Delete access.
            Status = ZwDuplicateObject(
                NtCurrentProcess(),                    //__in HANDLE SourceProcessHandle,
                *(HANDLE*)SectionHandle                //__in HANDLE SourceHandle,
```

12、Fake_ZwCreateSymbolicLinkObject

参考资料：

1、突破HIPS的防御思路之duplicate physical memory

2、续PhysicalMemory攻击

前言：

CreateSymbolicLink,这也是很古老的方法，很多安全软件也已经防御。因为一些安全软件只防御了NtOpenSection,并根据打开的对象名是否是\Device\PhysicalMemory来进行拦截，但是只要对\Device\PhysicalMemory创建符号链接，那么一样可以使用NtOpenSection打开

序号：0x20

检查函数类型：属于第一类，函数执行前检查

设计思路：

1、拦截创建\Device\PhysicalMemory符号链接

函数执行前判断：

```
//2、必须是应用层调用
if (ExGetPreviousMode())
{
    //判断参数合法性
    if (myProbeRead(In_TargetName, sizeof(UNICODE_STRING), sizeof(CHAR)))
    {
        KdPrint(("ProbeRead(Fake_ZwCreateSymbolicLinkObject: In_TargetName) error \r\n"));
        return result;
    }
    //访问敏感符号路径：\\Device\\PhysicalMemory，只有白名单调用者才有资格访问
    if (RtlEqualUnicodeString(In_TargetName, &PhysicalMemoryString, TRUE) && !Safe_QueryWhitePID(PsGetCurrentProcessId()))
    {
        result = STATUS_ACCESS_DENIED;
    }
```

13、Fake_ZwCreateSection

参考资料：

1、老树开新花：DLL劫持漏洞新玩法

2、通过挂钩NtCreateSection监控可执行模块

3、hook方式进程创建监控，进程保护

略

14、Fake_ZwDuplicateObject

参考资料：
1、突破HIPS的防御思路之duplicate physical memory
2、续PhysicalMemory攻击
3、Windows内核安全与驱动开发
4、内核对象HOOK

前言：
　获取内核对象的句柄不仅仅可以通过Open打开获取，还可以通过句柄拷贝的操作。通过句柄复制可以把一个进程句柄表中的句柄复制到另一个进程的句柄表中。并且还可以提升复制后的句柄权限。

函数介绍：

序号：0x33
检查函数类型：属于第一类，函数执行前检查

设计思路：
1、禁止操作受保护源目标进程句柄
2、目标进程非自身都触发检查
3、拷贝后权限不变的直接无视
4、检查部分就将File、Thread、Section、Process判断代码全部融到一起
//里面细化处理各种类型：File、Process、Section、Thread敏感操作
//File：　　违规操作：查文件信息表，存在则报错
//Process：违规操作：访问句柄是指定的白名单、自身进程是IE
//Section：违规操作：路径是\\Device\\PhysicalMemory和\\KnownDlls\\
//Thread：违规操作：访问句柄是指定的白名单、自身进程是IE

防止误伤过滤：
1、必须是用户层调用
2、调用者白名单直接放行
3、句柄类型必须是Process

函数执行前判断：
略

15、永恒之蓝

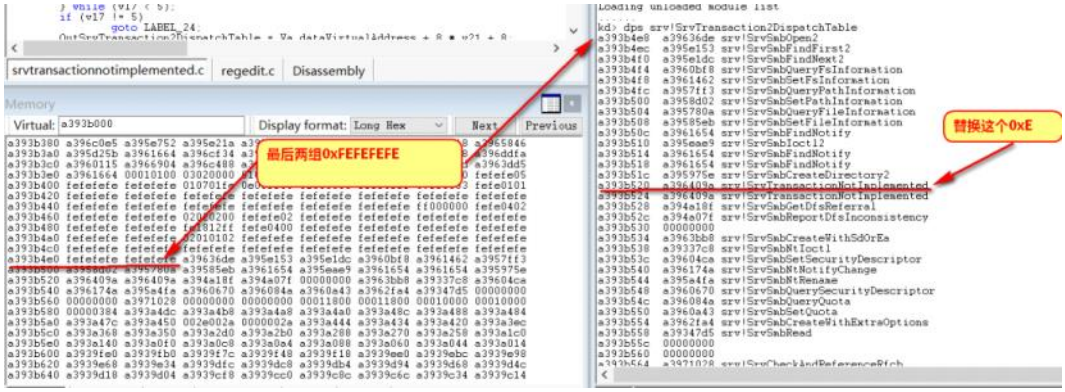
参考资料：
1、在看永恒之蓝
2、EternalBlue Shellcode详细分析
3、NSA Eternalblue SMB 漏洞分析

前言：
永恒之蓝漏洞(CVE-2017-0144),替换srv!SrvTransaction2DispatchTable的0x0E

设计思路：
1、支持XP~Win8_9200

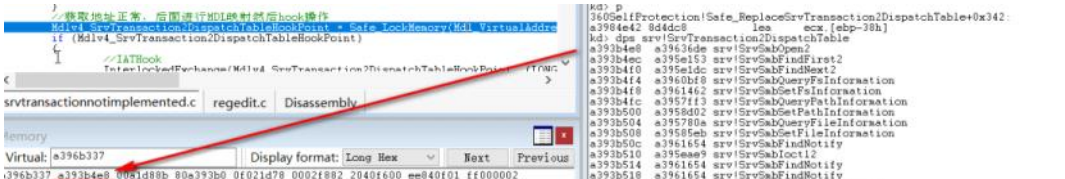
2、获取Srv.sys驱动基地址和大小

3、获取SrvTransaction2DispatchTable函数地址
通过遍历PE结构；定位到.data区段，找到.data节后，最后进行内存搜索，根据SrvTransaction2DispatchTable的结构特征 0xFEFEFEFE，0xFEFEFEFE，SrvTransaction2DispatchTable
Windbg输入：dps srv!SrvTransaction2DispatchTable

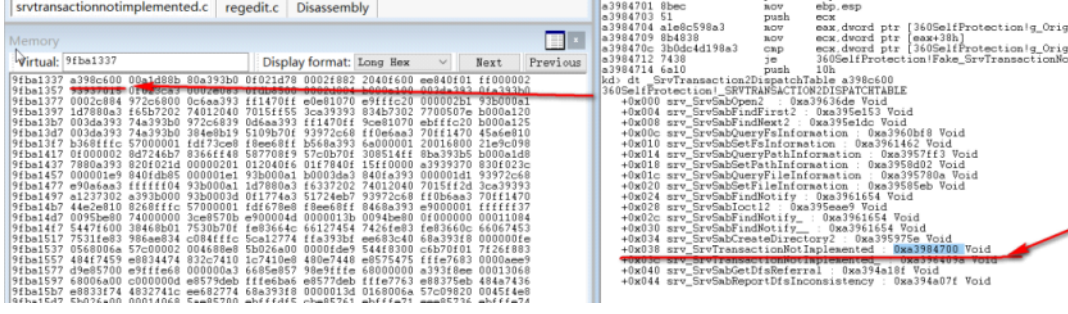


4、为SrvTransaction2DispatchTable结构重新分配一份空间，并修改0xE的项函数地址（后面替换原始重定位表地址）
RtlCopyMemory((VOID*)&g_MdlSrvTransaction2DispatchTable, (CONST VOID*)g_OriginalSrvTransaction2DispatchTable, sizeof(SRVTRANSACTION2DISPATCHTABLE));
//保存原始的第0x0E项的函数地址
g_OriginalSrvTransactionNotImplementedPtr = g_MdlSrvTransaction2DispatchTable.srv_SrvTransactionNotImplemented;
//替换第0x0E项的函数地址为fake函数
g_MdlSrvTransaction2DispatchTable.srv_SrvTransactionNotImplemented = Fake_SrvTransactionNotImplemented_0xE;

5、MDL方式映射，再替换掉重定位表的原始地址
替换前



替换后（g_MdlSrvTransaction2DispatchTable.srv_SrvTransactionNotImplemented的地址）



6、Fake_SrvTransactionNotImplemented_0xE函数实现


v1 = *(g_OriginalSrvTransaction2DispatchTable + 0x38);
if (v1 != g_OriginalSrvTransactionNotImplementedPtr)// 判断是否
{
if (*&qword_38480.QuadPart)
{
KeQuerySystemTime(&CurrentTime);
v2 = (*&CurrentTime.QuadPart - *&qword_38480.QuadPart) >> 0x20;
if (v2 <= 0 && (v2 < 0 || (LODWORD(CurrentTime.QuadPart) - LODWORD(qword_38480.QuadPart)) < 1000000000))
goto LABEL_9;
KeQuerySystemTime(&qword_38480);
}
else
{
KeQuerySystemTime(&qword_38480);
}
ThreadId = PsGetCurrentThreadId();
ProcessId = PsGetCurrentProcessId();
Safe_18A72_SendR3(ProcessId, ThreadId, 0x10);// 提示用户拦截蠕动远程攻击
LABEL_9:
*(g_OriginalSrvTransaction2DispatchTable + 0x38) = g_OriginalSrvTransactionNotImplementedPtr;
return v1(a1); // 调用原始的
}
return g_OriginalSrvTransactionNotImplementedPtr(a1);
}
跟OD执行时间反调试一样，必须再规定时间内执行完毕？？

16、额外学习到的知识

参考资料：
1、空指针漏洞防护技术 提高篇
2、圣诞礼物：妙用0地址~
3、空指针漏洞防护技术-初级篇

1、NULL不是一个无效的参数（Win7）

理论：

空指针指向了内存的什么地方呢？标准中并没有对空指针指向内存中的什么地方这一个问题作出规定，也就是说用哪个具体的地址值（0x0 地址还是某一特定地址）表示空指针取决于系统的实现。我们常见的 **空指针一般指向 0 地址，即空指针的内部用全 0 来表示** （zero null pointer，零空指针）。

对于NULL能表示空指针，是否还有其他值来表示空指针呢？在windows核心编程第五版的windows内存结构一章中，表13-1有提到NULL指针分配的分区，其范围是从0x00000000到0x0000FFFF。这段空间是空闲的，对于空闲的空间而言，没有相应的物理存储器与之相对应，所以对这段空间来说，任何读写操作都是会引起异常的。

表13-1 进程的地址空间如何分区

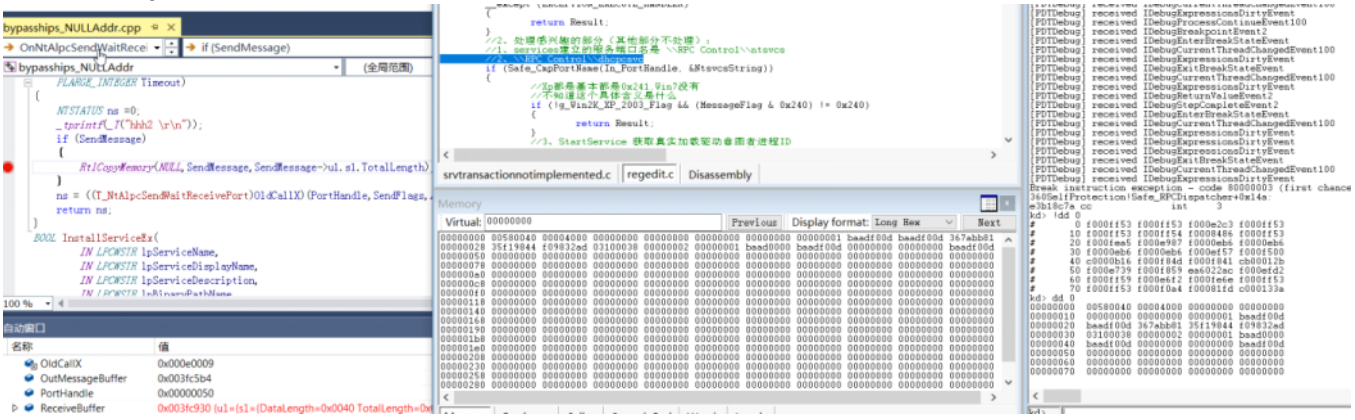
分区	32位Windows（x86和Alpha处理器）	32位Windows 2000(x86 w/3 GB用户方式)	64位Windows 2000（Alpha和IA-64处理器）	Windows 98
NULL指针分配的分区	0x00000000	0x00000000	0x00000000 00000000	0X00000000
	0x0000FFFF	0x0000FFFF	0x00000000 0000FFFF	0x0000FFFF
DOS 16位	无	无	无	0x00001000
Windows应用程序兼容分区				0x003FFFFF
用户方式	0x00010000	0x00010000	0x00000000 00010000	0x00400000
	0x7FFEFFFF	0xBFFEFFFF	0x000003FF FFFEFFFF	0x7FFFFFFF
64-KB禁止进入	0x7FFF0000	0xBFFF0000	0x000003FF FFFF0000	无
	0x7FFFFFFF	0xBFFFFFFF	0x000003FF FFFFFFFF	
共享内存映射文件（MMF）内核方式	无	无	无	0x80000000
	0x80000000	0xC0000000	0x00000400 00000000	0xC0000000
	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF FFFFFFFF	0xFFFFFFFF

从图中看出，对NULL指针分配的区域有0x10000之多，为什么分配如此大的空间？在定义NULL的时候，只使用了 0x00000000这么一个值，而在表中有提到NULL指针分配的分区包含了0x00000000-0x0000FFFF,是不是有点浪费空间了；这是和操作系统地址空间的分配粒度相关的，windows X86的默认分配粒度是64KB，为了达到对齐，空间地址需要从0x00010000开始分配，故空指针的区间范围有那么大。

代码测试：

刚好学习到ZwAlpcSendWaitReceivePort,配合v校的代码测试了一波NULL地址

V校测试代码SendMessage是NULL



17、基础知识

1、KeuserModeCallback防全局消息钩子注入

函数介绍	API	防范措施	使用途径
加载DLL	ClientLoadlibrary		
加载DLL	ClientImmLoadLayout		未知知识盲区
拦截键盘消息	fnHkINLPKBDLLHOOKSTRUCT		未知知识盲区
未知	fnHkINLPKBDLLHOOKSTRUCT		未知知识盲区

2、拦截进程打开、读、写，以及创建远程线程、发送APC等操作拦截如下API：

函数介绍	API	防范措施	使用途径
打开进程	NtOpenProcess	降权	万恶之源，根源处抹杀
打开线程	NtOpenThread	白名单保护	
复制句柄	NtDuplicateObject	略	可以拷贝一份高权限的句柄
申请内存	NtAllocateVirtualMemory	白名单保护	
写入内存	NtWriteVirtualMemory	白名单保护	
创建线程	NtCreateThread(Ex)	略	创建远程线程
更改线程上下文	NtSetContextThread	略	
插入APC	NtQueueApcThread	略	APC注入
清空目标进程	NtUnmapViewOfSection	白名单保护	傀儡进程

3、常见黄金搭档组合

NtDuplicateObject + NtTerminateProcess = 结束进程

ZwCreateSymbolicLinkObject + ZwOpenSection = PhysicalMemory攻击

4、句柄方面的防护

获取句柄的方式通过以下几种：

1、Create\Open方式打开

2、通过DuplicateObject复制

3、通过句柄继承的方式获取句柄

解决方法：

第一种：访问敏感进程时直接句柄权限降权

第二种：Craete\Open时候权限去掉PROCESS_DUP_HANDLE 权限，那么就无法使用DuplicateObject复制

第三种：通过挂钩OpenProceduer方式拦截（Windows内核安全与驱动开发）