

在笔者上一篇文章《内核枚举LoadImage映像回调》中 LyShark 教大家实现了枚举系统回调中的 LoadImage 通知消息，本章将实现对 Registry 注册表通知消息的枚举，与 LoadImage 消息不同 Registry 消息不需要解密只要找到 callbackListHead 消息回调链表头并解析为 _CM_NOTIFY_ENTRY 结构即可实现枚举。

Registry注册表回调是Windows操作系统提供的一种机制，它允许开发者在注册表发生变化时拦截并修改注册表的操作。Registry注册表回调是通过操作系统提供的注册表回调机制来实现的。

当应用程序或系统服务对注册表进行读写操作时，操作系统会触发注册表回调事件，然后在注册表回调事件中调用注册的Registry注册表回调函数。开发者可以在Registry注册表回调函数中执行自定义的逻辑，例如记录日志，过滤敏感数据，或者阻止某些操作。

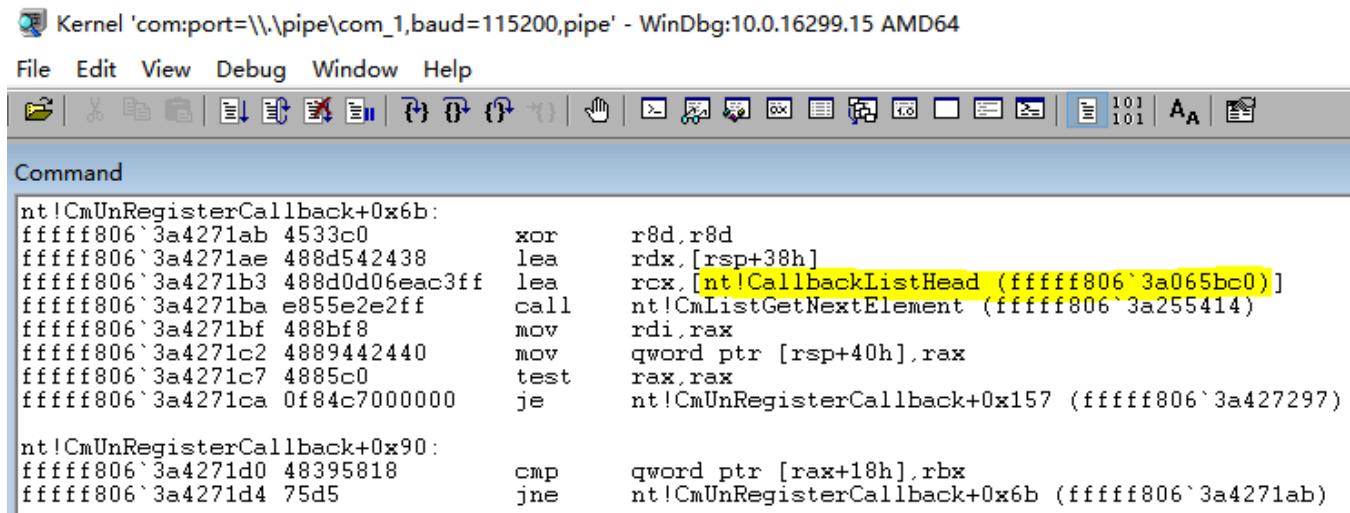
Registry注册表回调可以通过操作系统提供的注册表回调函数CmRegisterCallback和CmUnRegisterCallback来进行注册和注销。同时，Registry注册表回调函数需要遵守一定的约束条件，例如不能在回调函数中对注册表进行修改，不能调用一些内核API函数等。

Registry注册表回调在安全软件、系统监控和调试工具等领域有着广泛的应用。

我们来看一款闭源ARK工具是如何实现的：

进程	驱动模块	内核层	内核钩子	应用层钩子	设置	监控	启动信息	注册表	服务	文件	网络	调试引擎
系统回调												
			回调入口	通知类型				模块路径				
			0xFFFFF80636A2D760	ThreadObCall				C:\Windows\System32\drivers\PYArkSafe.sys				
			0xFFFFF8063C39D890	Registry				C:\Windows\system32\drivers\WdFilter.sys				
			0xFFFFF8063A065BE0	Registry				C:\Windows\system32\ntoskrnl.exe				
			0xFFFFF8063C3A8410	ProcessObCall				C:\Windows\system32\drivers\WdFilter.sys				
			0xFFFFF80636A2D420	ProcessObCall				C:\Windows\System32\drivers\PYArkSafe.sys				
			0xFFFFF80636A2C550	LoadImage				C:\Windows\System32\drivers\PYArkSafe.sys				

注册表系统回调的枚举需要通过特征码搜索来实现，首先我们可以定位到 `uf CmUnRegisterCallback` 内核函数上，在该内核函数下方存在一个 `callbackListHead` 链表节点，取出这个链表地址。



```
Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:10.0.16299.15 AMD64
File Edit View Debug Window Help
Command
nt!CmUnRegisterCallback+0x6b:
fffff806`3a4271ab 4533c0    xor    r8d,r8d
fffff806`3a4271ae 488d542438  lea    rdx,[rsp+38h]
fffff806`3a4271b3 488d0d06eac3ff  lea    rcx,[nt!CallbackListHead (fffff806`3a065bc0)]
fffff806`3a4271ba e855e2e2ff  call   nt!CmListGetNextElement (fffff806`3a255414)
fffff806`3a4271bf 488bf8    mov    rdi,rax
fffff806`3a4271c2 4889442440  mov    qword ptr [rsp+40h],rax
fffff806`3a4271c7 4885c0    test  rax,rax
fffff806`3a4271ca 0f84c7000000  je    nt!CmUnRegisterCallback+0x157 (fffff806`3a427297)

nt!CmUnRegisterCallback+0x90:
fffff806`3a4271d0 48395818  cmp    qword ptr [rax+18h],rbx
fffff806`3a4271d4 75d5    jne   nt!CmUnRegisterCallback+0x6b (fffff806`3a4271ab)
```

当得到注册表链表入口 `0xfffff8063a065bc0` 直接将其解析为 `_CM_NOTIFY_ENTRY` 即可得到数据，如果要遍历下一个链表则只需要 `ListEntryHead.Flink` 向下移动指针即可。

```

// 注册回调函数结构体定义
typedef struct _CM_NOTIFY_ENTRY
{
    LIST_ENTRY ListEntryHead;
    ULONG UnKnown1;
    ULONG UnKnown2;
    LARGE_INTEGER Cookie;
    PVOID Context;
    PVOID Function;
} CM_NOTIFY_ENTRY, *PCM_NOTIFY_ENTRY;

```

要想得到此处的链表地址，需要先通过 `MmGetSystemRoutineAddress()` 获取到 `CmUnRegisterCallback` 函数基址，然后在该函数起始位置向下搜索，找到这个链表节点，并将其后面的基址址取出来，在上一篇《内核枚举LoadImage映像回调》文章中已经介绍了定位方式此处跳过介绍，具体实现代码如下。

```

#include <ntifs.h>
#include <windef.h>

// 指定内存区域的特征码扫描
VOID SearchMemory(VOID pStartAddress, VOID pEndAddress, PUCHAR pMemoryData, ULONG ulMemoryDataSize)
{
    VOID pAddress = NULL;
    PUCHAR i = NULL;
    ULONG m = 0;

    // 扫描内存
    for (i = (PUCHAR)pStartAddress; i < (PUCHAR)pEndAddress; i++)
    {
        // 判断特征码
        for (m = 0; m < ulMemoryDataSize; m++)
        {
            if (*(PUCHAR)(i + m) != pMemoryData[m])
            {
                break;
            }
        }
        // 判断是否找到符合特征码的地址
        if (m >= ulMemoryDataSize)
        {
            // 找到特征码位置，获取紧接着特征码的下一地址
            pAddress = (VOID)(i + ulMemoryDataSize);
            break;
        }
    }

    return pAddress;
}

// 根据特征码获取 callbackListHead 链表地址
VOID SearchCallbackListHead(PUCHAR pSpecialData, ULONG ulSpecialDataSize, LONG lSpecialOffset)

```

```

{

    UNICODE_STRING ustrFuncName;
    PVOID pAddress = NULL;
    LONG lOffset = 0;
    PVOID pCmUnRegisterCallback = NULL;
    PVOID pCallbackListHead = NULL;

    // 先获取 CmUnRegisterCallback 函数地址
    RtlInitUnicodeString(&ustrFuncName, L"CmUnRegisterCallback");
    pCmUnRegisterCallback = MmGetSystemRoutineAddress(&ustrFuncName);
    if (NULL == pCmUnRegisterCallback)
    {
        return pCallbackListHead;
    }

    // 查找 ffffff806`3a4271b3 488d0d06eac3ff  lea      rcx,[nt!callbackListHead
    (fffff806`3a065bc0)]
    /*
    1yshark.com>
        nt!CmUnRegisterCallback+0x6b:
        fffff806`3a4271ab 4533c0          xor     r8d,r8d
        fffff806`3a4271ae 488d542438      lea     rdx,[rsp+38h]
        fffff806`3a4271b3 488d0d06eac3ff  lea     rcx,[nt!callbackListHead
    (fffff806`3a065bc0)]
        fffff806`3a4271ba e855e2e2ff      call   nt!CmListGetNextElement
    (fffff806`3a255414)
        fffff806`3a4271bf 488bf8          mov    rdi,rax
        fffff806`3a4271c2 4889442440      mov    qword ptr [rsp+40h],rax
        fffff806`3a4271c7 4885c0          test   rax,rax
        fffff806`3a4271ca 0f84c7000000      je    nt!CmUnRegisterCallback+0x157
    (fffff806`3a427297)  Branch
    */
    pAddress = SearchMemory(pCmUnRegisterCallback, ((PVOID)((PUCHAR)pCmUnRegisterCallback +
    0xFF), pSpecialData, ulSpecialDataSize);
    if (NULL == pAddress)
    {
        return pCallbackListHead;
    }

    // 先获取偏移再计算地址
    lOffset = *(PLONG)((PUCHAR)pAddress + 1SpecialOffset);
    pCallbackListHead = (PVOID)((PUCHAR)pAddress + 1SpecialOffset + sizeof(LONG) + lOffset);

    return pCallbackListHead;
}

VOID UnDriver(PDRIVER_OBJECT Driver)
{
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
}

```

```

PVOID pCallbackListHeadAddress = NULL;
RTL_OSVERSIONINFO osInfo = { 0 };
UCHAR pSpecialData[50] = { 0 };
ULONG ulSpecialDataSize = 0;
LONG lSpecialOffset = 0;

DbgPrint("hello lyshark.com \n");

// 查找 ffffff806`3a4271b3 488d0d06eac3ff  lea      rcx,[nt!callbackListHead
fffff806`3a065bc0)
/*
lyshark.com>
nt!CmUnRegisterCallback+0x6b:
fffff806`3a4271ab 4533c0    xor    r8d,r8d
fffff806`3a4271ae 488d542438  lea    rdx,[rsp+38h]
fffff806`3a4271b3 488d0d06eac3ff  lea    rcx,[nt!callbackListHead (fffff806`3a065bc0)]
fffff806`3a4271ba e855e2e2ff  call   nt!CmListGetNextElement (fffff806`3a255414)
fffff806`3a4271bf 488bf8    mov    rdi,rax
fffff806`3a4271c2 4889442440  mov    qword ptr [rsp+40h],rax
fffff806`3a4271c7 4885c0    test   rax,rax
fffff806`3a4271ca 0f84c7000000  je    nt!CmUnRegisterCallback+0x157
fffff806`3a427297) Branch
*/
pSpecialData[0] = 0x48;
pSpecialData[1] = 0x8D;
pSpecialData[2] = 0x0D;
ulSpecialDataSize = 3;

// 根据特征码获取地址
pCallbackListHeadAddress = SearchCallbackListHead(pSpecialData, ulSpecialDataSize,
lSpecialOffset);

DbgPrint("[Lyshark.com] callbackListHead => %p \n", pCallbackListHeadAddress);

Driver->DriverUnload = UnDriver;
return STATUS_SUCCESS;
}

```

运行这段代码，并可得到注册表回调入口地址，输出效果如下所示：

DebugView on \\DESKTOP-B53PAVI (local)		
File Edit Capture Options Computer Help		
#	Time	Debug Print
1	0.000000000	4836406250 - STORMINI: StorNVMe - POWER: IDLE
2	0.34673479	4839687500 - STORMINI: StorNVMe - POWER: ACTIVE
3	1.06057155	hello lyshark.com
4	1.06527448	[LyShark.com] CallbackListHead => FFFFF8063A065BC0
5	1.48369396	4849843750 - STORMINI: StorNVMe - POWER: IDLE
6	1.62090075	4851093750 - STORMINI: StorNVMe - POWER: ACTIVE
7	2.89499855	4861250000 - STORMINI: StorNVMe - POWER: IDLE
8	3.17070532	4863906250 - STORMINI: StorNVMe - POWER: ACTIVE

得到了注册表回调入口地址，接着直接循环遍历输出这个链表即可得到所有的注册表回调。

```
#include <ntifs.h>
#include <windef.h>

// 指定内存区域的特征码扫描
VOID SearchMemory(VOID pStartAddress, VOID pEndAddress, UCHAR pMemoryData, ULONG ulMemoryDataSize)
{
    VOID pAddress = NULL;
    UCHAR i = NULL;
    ULONG m = 0;

    // 扫描内存
    for (i = (UCHAR)pStartAddress; i < (UCHAR)pEndAddress; i++)
    {
        // 判断特征码
        for (m = 0; m < ulMemoryDataSize; m++)
        {
            if (*(PUCHAR)(i + m) != pMemoryData[m])
            {
                break;
            }
        }

        // 判断是否找到符合特征码的地址
        if (m >= ulMemoryDataSize)
        {
            // 找到特征码位置，获取紧接着特征码的下一地址
            pAddress = (VOID)(i + ulMemoryDataSize);
            break;
        }
    }

    return pAddress;
}
```

```

}

// 根据特征码获取 CallbackListHead 链表地址
VOID SearchCallbackListHead(PUCHAR pSpecialData, ULONG ulSpecialDataSize, LONG lSpecialOffset)
{
    UNICODE_STRING ustrFuncName;
    PVOID pAddress = NULL;
    LONG lOffset = 0;
    PVOID pCmUnRegisterCallback = NULL;
    PVOID pCallbackListHead = NULL;

    // 先获取 CmUnRegisterCallback 函数地址
    RtlInitUnicodeString(&ustrFuncName, L"CmUnRegisterCallback");
    pCmUnRegisterCallback = MmGetSystemRoutineAddress(&ustrFuncName);
    if (NULL == pCmUnRegisterCallback)
    {
        return pCallbackListHead;
    }

    // 查找 ffffff806`3a4271b3 488d0d06eac3ff  lea      rcx,[nt!CallbackListHead
    (fffff806`3a065bc0)]
    /*
    lshark.com>
        nt!CmUnRegisterCallback+0x6b:
        fffff806`3a4271ab 4533c0      xor     r8d,r8d
        fffff806`3a4271ae 488d542438  lea     rdx,[rsp+38h]
        fffff806`3a4271b3 488d0d06eac3ff lea     rcx,[nt!CallbackListHead
    (fffff806`3a065bc0)]
        fffff806`3a4271ba e855e2e2ff  call    nt!CmListGetNextElement
    (fffff806`3a255414)
        fffff806`3a4271bf 488bf8      mov     rdi,rax
        fffff806`3a4271c2 4889442440  mov     qword ptr [rsp+40h],rax
        fffff806`3a4271c7 4885c0      test   rax,rax
        fffff806`3a4271ca 0f84c7000000 je     nt!CmUnRegisterCallback+0x157
    (fffff806`3a427297)  Branch
    */
    pAddress = SearchMemory(pCmUnRegisterCallback, ((PVOID)((PUCHAR)pCmUnRegisterCallback + 0xFF)), pSpecialData, ulSpecialDataSize);
    if (NULL == pAddress)
    {
        return pCallbackListHead;
    }

    // 先获取偏移再计算地址
    lOffset = *(PLONG)((PUCHAR)pAddress + lSpecialOffset);
    pCallbackListHead = (PVOID)((PUCHAR)pAddress + lSpecialOffset + sizeof(LONG) + lOffset);

    return pCallbackListHead;
}

// 注册表回调函数结构体定义
typedef struct _CM_NOTIFY_ENTRY

```

```

{
    LIST_ENTRY ListEntryHead;
    ULONG Unknown1;
    ULONG Unknown2;
    LARGE_INTEGER Cookie;
    PVOID Context;
    PVOID Function;
}CM_NOTIFY_ENTRY, *PCM_NOTIFY_ENTRY;

VOID UnDriver(PDRIVER_OBJECT Driver)
{
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    PVOID pCallbackListHeadAddress = NULL;
    RTL_OSVERSIONINFO osInfo = { 0 };
    UCHAR pSpecialData[50] = { 0 };
    ULONG ulSpecialDataSize = 0;
    LONG lSpecialOffset = 0;

    DbgPrint("hello lyshark.com \n");

    // 查找 fffff806`3a4271b3 488d0d06eac3ff lea      rcx,[nt!callbackListHead
    (fffff806`3a065bc0)]
    /*
    lyshark.com>
    nt!CmUnRegisterCallback+0x6b:
    fffff806`3a4271ab 4533c0      xor      r8d,r8d
    fffff806`3a4271ae 488d542438  lea      rdx,[rsp+38h]
    fffff806`3a4271b3 488d0d06eac3ff lea      rcx,[nt!callbackListHead (fffff806`3a065bc0)]
    fffff806`3a4271ba e855e2e2ff  call    nt!CmListGetNextElement (fffff806`3a255414)
    fffff806`3a4271bf 488bf8      mov     rdi,rax
    fffff806`3a4271c2 4889442440  mov     qword ptr [rsp+40h],rax
    fffff806`3a4271c7 4885c0      test   rax,rax
    fffff806`3a4271ca 0f84c7000000 je     nt!CmUnRegisterCallback+0x157
    (fffff806`3a427297) Branch
    */
    pSpecialData[0] = 0x48;
    pSpecialData[1] = 0x8D;
    pSpecialData[2] = 0x0D;
    ulSpecialDataSize = 3;

    // 根据特征码获取地址
    pCallbackListHeadAddress = SearchCallbackListHead(pSpecialData, ulSpecialDataSize,
    lSpecialOffset);

    DbgPrint("[Lyshark.com] callbackListHead => %p \n", pCallbackListHeadAddress);

    // 遍历链表结构
    ULONG i = 0;
    PCM_NOTIFY_ENTRY pNotifyEntry = NULL;
}

```

```

if (NULL == pCallbackListHeadAddress)
{
    return FALSE;
}

// 开始遍历双向链表
pNotifyEntry = (PCM_NOTIFY_ENTRY)pCallbackListHeadAddress;
do
{
    // 判断pNotifyEntry地址是否有效
    if (FALSE == MmIsAddressValid(pNotifyEntry))
    {
        break;
    }
    // 判断回调函数地址是否有效
    if (MmIsAddressValid(pNotifyEntry->Function))
    {
        DbgPrint("[LyShark.com] 回调函数地址: 0x%p | 回调函数Cookie: 0x%I64x \n",
pNotifyEntry->Function, pNotifyEntry->Cookie.QuadPart);
    }

    // 获取下一链表
    pNotifyEntry = (PCM_NOTIFY_ENTRY)pNotifyEntry->ListEntryHead.Flink;
}

} while (pCallbackListHeadAddress != (PVOID)pNotifyEntry);

Driver->DriverUnload = UnDriver;
return STATUS_SUCCESS;
}

```

最终运行这个驱动程序，输出如下效果：

#	Debug Print
1	7703281250 - STORMINI: StorNVMe - POWER: IDLE
2	7703906250 - STORMINI: StorNVMe - POWER: ACTIVE
3	hello lyshark.com
4	[LyShark.com] CallbackListHead => FFFFFF8063A065BC0
5	[LyShark.com] 回调函数地址: 0xFFFFF8063A065BEO 回调函数Cookie: 0x1D8E4FBECBA5A18
6	[LyShark.com] 回调函数地址: 0xFFFFF8063C39D890 回调函数Cookie: 0x1D8E4FBECBA5A18
7	7714062500 - STORMINI: StorNVMe - POWER: IDLE
8	7717187500 - STORMINI: StorNVMe - POWER: ACTIVE
9	7727187500 - STORMINI: StorNVMe - POWER: IDLE
10	7727656250 - STORMINI: StorNVMe - POWER: ACTIVE
11	7737812500 - STORMINI: StorNVMe - POWER: IDLE
12	7741250000 - STORMINI: StorNVMe - POWER: ACTIVE

目前系统中有两个回调函数，这一点在第一张图片中也可以得到，枚举是正确的。

