

在笔者之前的文章《内核特征码搜索函数封装》中我们封装实现了特征码定位功能，本章将继续使用该功能，本次我们需要枚举内核 LoadImage 映像回调，在Win64环境下我们可以设置一个 LoadImage 映像加载通告回调，当有新驱动或者DLL被加载时，回调函数就会被调用从而执行我们自己的回调例程，映像回调也存储在数组里，枚举时从数组中读取值之后，需要进行位运算解密得到地址。

LoadImage映像回调是Windows操作系统提供的一种机制，它允许开发者在加载映像文件（如DLL、EXE等）时拦截并修改映像的加载过程。LoadImage映像回调是通过操作系统提供的ImageLoad事件机制来实现的。

当操作系统加载映像文件时，它会调用LoadImage函数。在LoadImage函数内部，操作系统会触发ImageLoad事件，然后在ImageLoad事件中调用注册的LoadImage映像回调函数。开发者可以在LoadImage映像回调函数中执行自定义的逻辑，例如修改映像文件的内容，或者阻止映像文件的加载。

LoadImage映像回调可以通过Win32 API函数SetImageLoadCallback或者操作系统提供的驱动程序回调函数PsSetLoadImageNotifyRoutine来进行注册。同时，LoadImage映像回调函数需要遵守一定的约束条件，例如必须是非分页代码，不能调用一些内核API函数等。

我们来看一款闭源ARK工具是如何实现的：

回调入口	通知类型	模块路径
0xFFFFF80180D1D760	ThreadObCall	C:\Windows\System32\drivers\PYArkSafe.sys
0xFFFFF8018519D890	Registry	C:\Windows\system32\drivers\WdFilter.sys
0xFFFFF80180465BE0	Registry	C:\Windows\system32\ntoskrnl.exe
0xFFFFF801851A8410	ProcessObCall	C:\Windows\system32\drivers\WdFilter.sys
0xFFFFF80180D1D420	ProcessObCall	C:\Windows\System32\drivers\PYArkSafe.sys
0xFFFFF80180D1C550	LoadImage	C:\Windows\System32\drivers\PYArkSafe.sys
0xFFFFF801869FB210	LoadImage	C:\Windows\system32\DRIVERS\ahcache.sys
0xFFFFF801851AABA0	LoadImage	C:\Windows\system32\drivers\WdFilter.sys

如上所述，如果我们要拿到回调数组那么首先要得到该数组，数组的符号名是 `PspLoadImageNotifyRoutine` 我们可以在 `PsSetLoadImageNotifyRoutineEx` 中找到。

第一步使用WinDBG输入 `uf PsSetLoadImageNotifyRoutineEx` 首先定位到，能够找到 `PsSetLoadImageNotifyRoutineEx` 这里的两个位置都可以被引用，当然了这个函数可以直接通过 `PsSetLoadImageNotifyRoutineEx` 函数动态拿到此处不需要我们动态定位。

```
Command

nt!PsSetLoadImageNotifyRoutineEx+0x41:
fffff801`80748a81 488d0dd8d3dbff  lea    rcx,[nt!PspLoadImageNotifyRoutine (fffff801`80505e60)]
fffff801`80748a88 4533c0          xor   r8d,r8d
fffff801`80748a8b 488d0cd9          lea    rcx,[rcx+rbx*8]
fffff801`80748a8f 488bd7          mov   rdx,rdi
fffff801`80748a92 e80584a3ff        call  nt!ExCompareExchangeCallBack (fffff801`80180e9c)
fffff801`80748a97 84c0            test  al,al
fffff801`80748a99 0f849f0000000 je    nt!PsSetLoadImageNotifyRoutineEx+0xfe (fffff801`80748b3e) Branch
nt!PsSetLoadImageNotifyRoutineEx+0x5f:
fffff801`80748a9f f0ff05966e2600 lock inc dword ptr [nt!PspLoadImageNotifyRoutineCount (fffff801`809af93c)]
fffff801`80748aa6 8b0546b2600      mov   eax,dword ptr [nt!PspNotifyEnableMask (fffff801`809af5f0)]
fffff801`80748aac a801            test  al,i
fffff801`80748aaa 7509            jne   nt!PsSetLoadImageNotifyRoutineEx+0x79 (fffff801`80748ab9) Branch
```

我们通过获取到 `PsSetLoadImageNotifyRoutineEx` 函数的内存首地址，然后向下匹配特征码搜索找到 `488d0d88e8dbff` 并取出 `PspLoadImageNotifyRoutine` 内存地址，该内存地址就是 `LoadImage` 映像模块的基址。

## Command

```
nt!PsSetLoadImageNotifyRoutineEx+0x41:
fffff801`80748a81 488d0dd8d3dbff lea    rcx,[nt!PspLoadImageNotifyRoutine (fffff801`80505e60)]
fffff801`80748a88 4533c0 xor   r8d,r8d
fffff801`80748a8b 488d0cd9 lea    rcx,[rcx+rbx*8]
fffff801`80748a8f 488bd7 mov   rdx,rdi
fffff801`80748a92 e80584a3ff call  nt!ExCompareExchangeCallBack (fffff801`80180e9c)
fffff801`80748a97 84c0 test  al,al
fffff801`80748a99 0f849f000000 je    nt!PsSetLoadImageNotifyRoutineEx+0xfe (fffff801`80748b3e) Branch
```

如果使用代码去定位这段空间，则你可以这样写，这样即可得到具体特征地址。

```
#include <ntddk.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 (*(UCHAR)(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 PspLoadImageNotifyRoutine 数组地址
VOID SearchPspLoadImageNotifyRoutine(UCHAR pSpecialData, ULONG ulSpecialDataSize)
{
    UNICODE_STRING usrfuncName;
    VOID pAddress = NULL;
    LONG lOffset = 0;
    VOID pPsSetLoadImageNotifyRoutine = NULL;
    VOID pPspLoadImageNotifyRoutine = NULL;
```

```

// 先获取 PsSetLoadImageNotifyRoutineEx 函数地址
RtlInitUnicodeString(&ustrFuncName, L"PsSetLoadImageNotifyRoutineEx");
pPsSetLoadImageNotifyRoutine = MmGetSystemRoutineAddress(&ustrFuncName);
if (NULL == pPsSetLoadImageNotifyRoutine)
{
    return pPspLoadImageNotifyRoutine;
}

// 查找 PspLoadImageNotifyRoutine 函数地址
pAddress = SearchMemory(pPsSetLoadImageNotifyRoutine, (PVOID)
((PUCHAR)pPsSetLoadImageNotifyRoutine + 0xFF), pSpecialData, ulSpecialDataSize);
if (NULL == pAddress)
{
    return pPspLoadImageNotifyRoutine;
}

// 先获取偏移，再计算地址
loffset = *(PLONG)pAddress;
pPspLoadImageNotifyRoutine = (PVOID)((PUCHAR)pAddress + sizeof(LONG) + loffset);

return pPspLoadImageNotifyRoutine;
}

VOID UnDriver(PDRIVER_OBJECT Driver)
{
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

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

    // 获取系统版本信息，判断系统版本
    RtlGetVersion(&osInfo);
    if (10 == osInfo.dwMajorVersion)
    {
        // 48 8d 0d 88 e8 db ff
        // 查找指令 lea rcx,[nt!PspLoadImageNotifyRoutine (fffff801`44313ce0)]
        /*
        nt!PsSetLoadImageNotifyRoutineEx+0x41:
        fffff801`80748a81 488d0dd8d3dbff  lea      rcx, [nt!PspLoadImageNotifyRoutine
(fffff801`80505e60)]
        fffff801`80748a88 4533c0          xor     r8d,r8d
        fffff801`80748a8b 488d0cd9          lea     rcx, [rcx+rbx*8]
        fffff801`80748a8f 488bd7          mov     rdx,rdi
        fffff801`80748a92 e80584a3ff         call   nt!ExCompareExchangeCallBack
(fffff801`80180e9c)
        fffff801`80748a97 84c0            test   al,al

```

```

fffff801`80748a99 0f849f000000    je      nt!PsSetLoadImageNotifyRoutineEx+0xfe
(fffff801`80748b3e)  Branch
/*
pSpecialData[0] = 0x48;
pSpecialData[1] = 0x8D;
pSpecialData[2] = 0x0D;
ulSpecialDatasize = 3;
}

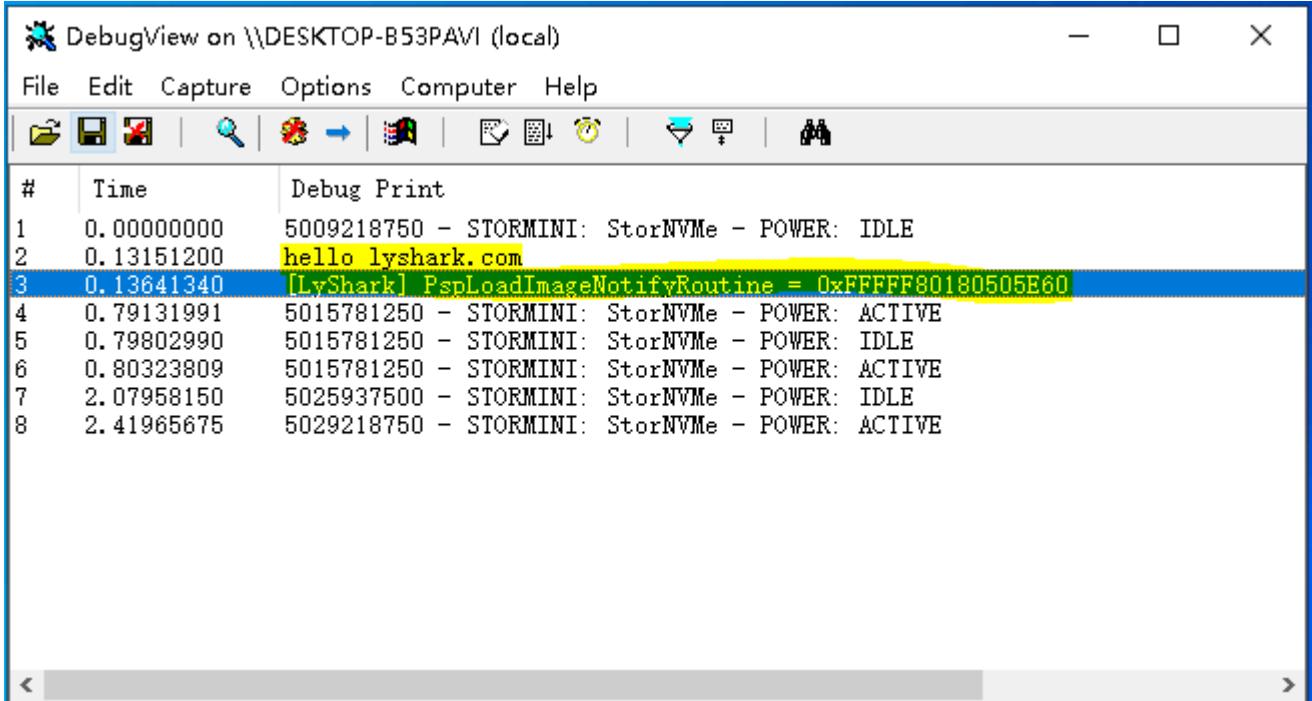
// 根据特征码获取地址 获取 PspLoadImageNotifyRoutine 数组地址
pPspLoadImageNotifyRoutineAddress = SearchPspLoadImageNotifyRoutine(pSpecialData,
ulSpecialDatasize);

DbgPrint("[LyShark] PspLoadImageNotifyRoutine = 0x%p \n",
pPspLoadImageNotifyRoutineAddress);

Driver->DriverUnload = UnDriver;
return STATUS_SUCCESS;
}

```

将这个驱动拖入到虚拟机中并运行，输出结果如下：



#	Time	Debug Print
1	0.000000000	5009218750 - STORMINI: StorNVMe - POWER: IDLE
2	0.13151200	hello lyshark.com
3	0.13641340	[LyShark] PspLoadImageNotifyRoutine = 0xFFFFF80180505E60
4	0.79131991	5015781250 - STORMINI: StorNVMe - POWER: ACTIVE
5	0.79802990	5015781250 - STORMINI: StorNVMe - POWER: IDLE
6	0.80323809	5015781250 - STORMINI: StorNVMe - POWER: ACTIVE
7	2.07958150	5025937500 - STORMINI: StorNVMe - POWER: IDLE
8	2.41965675	5029218750 - STORMINI: StorNVMe - POWER: ACTIVE

有了数组地址接下来就是要对数组进行解密，如何解密？

- 1.首先拿到数组指针 pPspLoadImageNotifyRoutineAddress + sizeof(PVOID) \* i 此处的i也就是下标。
- 2.得到的新地址在与 pNotifyRoutineAddress & 0xfffffffffffffff8 进行与运算。
- 3.最后 \*(PVOID \*)pNotifyRoutineAddress 取出里面的参数。

增加解密代码以后，这段程序的完整代码也就可以被写出来了，如下所示。

```

#include <ntddk.h>
#include <windef.h>

// 指定内存区域的特征码扫描

```

```
PVOID SearchMemory(PVOID pStartAddress, PVOID pEndAddress, PUCHAR pMemoryData, ULONG ulMemoryDataSize)
{
    PVOID 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 = (PVOID)(i + ulMemoryDataSize);
            break;
        }
    }

    return pAddress;
}

// 根据特征码获取 PspLoadImageNotifyRoutine 数组地址
PVOID SearchPspLoadImageNotifyRoutine(PUCHAR pSpecialData, ULONG ulSpecialDataSize)
{
    UNICODE_STRING ustrFuncName;
    PVOID pAddress = NULL;
    LONG lOffset = 0;
    PVOID pPsSetLoadImageNotifyRoutine = NULL;
    PVOID pPspLoadImageNotifyRoutine = NULL;

    // 先获取 PsSetLoadImageNotifyRoutineEx 函数地址
    RtlInitUnicodeString(&ustrFuncName, L"PsSetLoadImageNotifyRoutineEx");
    pPsSetLoadImageNotifyRoutine = MmGetSystemRoutineAddress(&ustrFuncName);
    if (NULL == pPsSetLoadImageNotifyRoutine)
    {
        return pPspLoadImageNotifyRoutine;
    }

    // 查找 PspLoadImageNotifyRoutine 函数地址
    pAddress = SearchMemory(pPsSetLoadImageNotifyRoutine, (PVOID)((PUCHAR)pPsSetLoadImageNotifyRoutine + 0xFF), pSpecialData, ulSpecialDataSize);
    if (NULL == pAddress)
    {
        return pPspLoadImageNotifyRoutine;
    }
}
```

```

}

// 先获取偏移，再计算地址
loffset = *(PLONG)pAddress;
pPspLoadImageNotifyRoutine = (PVOID)((PUCHAR)pAddress + sizeof(LONG) + loffset);

return pPspLoadImageNotifyRoutine;
}

// 移除回调
NTSTATUS RemoveNotifyRoutine(PVOID pNotifyRoutineAddress)
{
    NTSTATUS status =
PsRemoveLoadImageNotifyRoutine((PLOAD_IMAGE_NOTIFY_ROUTINE)pNotifyRoutineAddress);
    return status;
}

VOID UnDriver(PDRIVER_OBJECT Driver)
{
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

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

    // 获取系统版本信息，判断系统版本
    RtlGetVersion(&osInfo);
    if (10 == osInfo.dwMajorVersion)
    {
        // 48 8d 0d 88 e8 db ff
        // 查找指令 lea rcx,[nt!PspLoadImageNotifyRoutine (fffff804`44313ce0)]
        /*
        nt!PsSetLoadImageNotifyRoutineEx+0x41:
        fffff801`80748a81 488d0dd8d3dbff  lea      rcx, [nt!PspLoadImageNotifyRoutine
(fffff801`80505e60)]
        fffff801`80748a88 4533c0          xor      r8d,r8d
        fffff801`80748a8b 488d0cd9          lea      rcx, [rcx+rbx*8]
        fffff801`80748a8f 488bd7          mov      rdx,rdi
        fffff801`80748a92 e80584a3ff         call     nt!ExCompareExchangeCallBack
(fffff801`80180e9c)
        fffff801`80748a97 84c0          test     al,al
        fffff801`80748a99 0f849f000000         je      nt!PsSetLoadImageNotifyRoutineEx+0xfe
(fffff801`80748b3e)  Branch
        */
        pSpecialData[0] = 0x48;
        pSpecialData[1] = 0x8D;
        pSpecialData[2] = 0x0D;
        ulSpecialDataSize = 3;
    }
}

```

```

}

// 根据特征码获取地址 获取 PspLoadImageNotifyRoutine 数组地址
pPspLoadImageNotifyRoutineAddress = SearchPspLoadImageNotifyRoutine(pSpecialData,
ulSpecialDatasize);
DbgPrint("[LyShark] PspLoadImageNotifyRoutine = 0x%p \n",
pPspLoadImageNotifyRoutineAddress);

// 遍历回调
ULONG i = 0;
PVOID pNotifyRoutineAddress = NULL;

// 获取 PspLoadImageNotifyRoutine 数组地址
if (NULL == pPspLoadImageNotifyRoutineAddress)
{
    return FALSE;
}

// 获取回调地址并解密
for (i = 0; i < 64; i++)
{
    pNotifyRoutineAddress = *(PVOID *)((PUCHAR)pPspLoadImageNotifyRoutineAddress +
sizeof(PVOID) * i);
    pNotifyRoutineAddress = (PVOID)((ULONG64)pNotifyRoutineAddress &
0xfffffffffffffff8);
    if (MmIsAddressValid(pNotifyRoutineAddress))
    {
        pNotifyRoutineAddress = *(PVOID *)pNotifyRoutineAddress;
        DbgPrint("[LyShark] 序号: %d | 回调地址: 0x%p \n", i, pNotifyRoutineAddress);
    }
}

Driver->DriverUnload = UnDriver;
return STATUS_SUCCESS;
}

```

运行这段完整的程序代码，输出如下效果：

DebugView on \\DESKTOP-B53PAVI (local)

File Edit Capture Options Computer Help

#	Time	Debug Print
1	0.000000000	5580781250 - STORMINI: StorNVMe - POWER: ACTIVE
2	0.00778480	5580781250 - STORMINI: StorNVMe - POWER: IDLE
3	0.01418650	5580781250 - STORMINI: StorNVMe - POWER: ACTIVE
4	1.14490056	hello lyshark.com
5	1.14906919	[LyShark] PspLoadImageNotifyRoutine = 0xFFFFF80180505E60
6	1.15259695	[LyShark] 序号: 0   回调地址: 0xFFFFF801851AABA0
7	1.15770435	[LyShark] 序号: 1   回调地址: 0xFFFFF801869FB210
8	1.16953981	5590937500 - STORMINI: StorNVMe - POWER: IDLE
9	1.55468082	5594687500 - STORMINI: StorNVMe - POWER: ACTIVE
10	2.57838035	5604843750 - STORMINI: StorNVMe - POWER: IDLE
11	2.97423005	5608750000 - STORMINI: StorNVMe - POWER: ACTIVE
12	3.98607922	5618750000 - STORMINI: StorNVMe - POWER: IDLE
13	6.41761017	5640468750 - STORMINI: StorNVMe - POWER: ACTIVE
14	6.42514610	5640468750 - STORMINI: StorNVMe - POWER: IDLE
15	6.43382788	5640468750 - STORMINI: StorNVMe - POWER: ACTIVE

目前系统中只有两个回调，所以枚举出来的只有两条，打开ARK验证一下会发现完全正确，忽略 pyark 这是后期打开的。

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		
0xFFFFF8018118D760										ThreadObCall		
0xFFFFF8018519D890										Registry		
0xFFFFF80180465BE0										Registry		
0xFFFFF801851A8410										ProcessObCall		
0xFFFFF8018118D420										ProcessObCall		
0xFFFFF8018118C550										LoadImage		
0xFFFFF801869FB210										LoadImage		
0xFFFFF801851AABA0										LoadImage		