

DKOM – Now with Symbolic Links!

 windows-internals.com/dkom-now-with-symbolic-links

By Yarden Shafir & Alex Ionescu

You might think “What can ANYONE still say about kernel callbacks? We’ve already seen every callback possible – there are process creation callbacks, object type callbacks, image load notifications, callback objects, object type callbacks, host extensions... there can’t be any more kinds of callbacks. Right? Right...?”

Nope.

In Microsoft’s never-ending attempt to close one door for kernel hooking and open two more, Windows 10 Creators Update (RS2) added a new type of callback – this time for symbolic links.

Notice [these recent changes](#) to the `OBJECT_SYMBOLIC_LINK` structure:

```
typedef struct _OBJECT_SYMBOLIC_LINK
{
    LARGE_INTEGER CreationTime;
+   union
+   {
        UNICODE_STRING LinkTarget;
+       struct
+       {
+           POBJECT_SYMBOLIC_LINK_CALLBACK Callback;
+           PVOID CallbackContext;
+       };
+   }
    ULONG DosDeviceDriveIndex;
    ULONG Flags;
    ULONG AccessMask;
} OBJECT_SYMBOLIC_LINK, *POBJECT_SYMBOLIC_LINK;
```

What used to be a Unicode String containing the target of the symbolic link is now a union that contains one of our favorite keywords to see when looking at the kernel – *callback*.

These callbacks were added in RS2 to support Memory Partitions, which are a new type of object used to segment physical address ranges into their own instance of the memory manager. Without going into too many details, the key point is that some of the event objects in `\KernelObjects`, such as `LowMemoryCondition` are no longer global – but rather refer to the specific conditions in the Memory Partition of the current caller. However, in order not to break compatibility, their naming and location could not be changed (such as `\KernelObjects\Partition2\`

`LowMemoryCondition`). As a result, they were turned into symbolic links attached to a dynamic callback, which will look at the current Memory Partition in `EPROCESS` and return the appropriate `KEVENT` Object for the caller's partition.

Now, whenever bit `5` in the symbolic link flags is set (`Flags & OBJECT_SYMBOLIC_LINK_USE_CALLBACK`), the `LinkTarget` will not be treated as a string, but instead will be treated as a function with this prototype:

```
typedef
NTSTATUS
(POBJECT_SYMBOLIC_LINK_CALLBACK*) (
    _In_ POBJECT_SYMBOLIC_LINK Symlink,
    _In_ PVOID SymlinkContext,
    _Out_ PUNICODE_STRING SymlinkPath,
    _Outptr_ PVOID* Object
);
```

This function will be called whenever the symbolic link is reparsed, and has to either set the `SymlinkPath` parameter to the target path to be parsed by the object manager, or set the `Object` parameter to the correct object that will be used as the target for this symbolic link.

This callback is set (or not set) by the `ObCreateSymbolicLink` function, based on an input structure that contains flags and the target string or callback function:

```
#define OB_SYMLINK_TARGET_DYNAMIC 0x01
typedef struct _OB_SYMLINK_TARGET
{
    ULONG Flags;
    union
    {
        UNICODE_STRING LinkTarget;
        struct
        {
            POBJECT_SYMBOLIC_LINK_CALLBACK Callback;
            PVOID CallbackContext;
        };
    };
};
} OB_SYMLINK_TARGET, *POB_SYMLINK_TARGET;
```

Based on this parameter, the function creates a symbolic link object and sets its target:

```
NTSTATUS
ObCreateSymbolicLink (
    _Out_ PHANDLE LinkHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
```

```

    _In_ POB_SYMLINK_TARGET TargetInfo,
    _In_ KPROCESSOR_MODE AccessMode
)
{
    NTSTATUS status;
    PWCHAR linkString;
    POBJECT_SYMBOLIC_LINK symlinkObject;
    HANDLE linkHandle;

    //
    // Create the symlink object
    //
    symlinkObject = NULL;
    status = ObCreateObjectEx(AccessMode,
                             ObpSymbolicLinkObjectType,
                             ObjectAttributes,
                             AccessMode,
                             0,
                             sizeof(*symlinkObject),
                             0,
                             0,
                             &symlinkObject,
                             NULL);

    if (NT_SUCCESS(status))
    {
        KeQuerySystemTime(&symlinkObject->CreationTime);
        symlinkObject->DosDeviceDriveIndex = 0;
        symlinkObject->Flags = 0;

        //
        // If the symlink has a dynamic target, set the flags accordingly
        // and populate the callback field
        //
        if (TargetInfo->Flags & OB_SYMLINK_TARGET_DYNAMIC)
        {
            symlinkObject->Flags = OBJECT_SYMBOLIC_LINK_USE_CALLBACK;
            symlinkObject->Callback = TargetInfo->Callback;
        }
        else
        {
            //
            // If the symlink doesn't have a dynamic target, set the
            LinkTarget to the string
            //
            symlinkObject->LinkTarget.MaximumLength = TargetInfo-
            >LinkTarget.MaximumLength;

```

```

        symlinkObject->LinkTarget.Length = TargetInfo-
>LinkTarget.Length;
        linkString = (PWCHAR)ExAllocatePoolWithTag(PagedPool,
                                                    TargetInfo-
>LinkTarget.MaximumLength,
                                                    'tmyS');
        symlinkObject->LinkTarget.Buffer = linkString;
        if (linkString == NULL)
        {
            status = STATUS_NO_MEMORY;
            goto Exit;
        }

```

```

        RtlCopyMemory(linkString,
                    TargetInfo->LinkTarget.Buffer,
                    TargetInfo->LinkTarget.MaximumLength);
    }

```

```

    if (RtlIsSandboxedToken(NULL, AccessMode) != FALSE)
    {
        symlinkObject->Flags |= OBJECT_SYMBOLIC_IS_SANDBOXED;
    }

```

```

    status = ObInsertObjectEx(symlinkObject,
                            NULL,
                            DesiredAccess,
                            0,
                            0,
                            NULL,
                            &linkHandle);

    symlinkObject = NULL;

```

```

    if (NT_SUCCESS(status))
    {
        *LinkHandle = linkHandle;
        status = STATUS_SUCCESS;
    }
}

```

```

Exit:
    if (symlinkObject != NULL)
    {
        ObDereferenceObject(symlinkObject);
    }

```

```

return status;
}

```

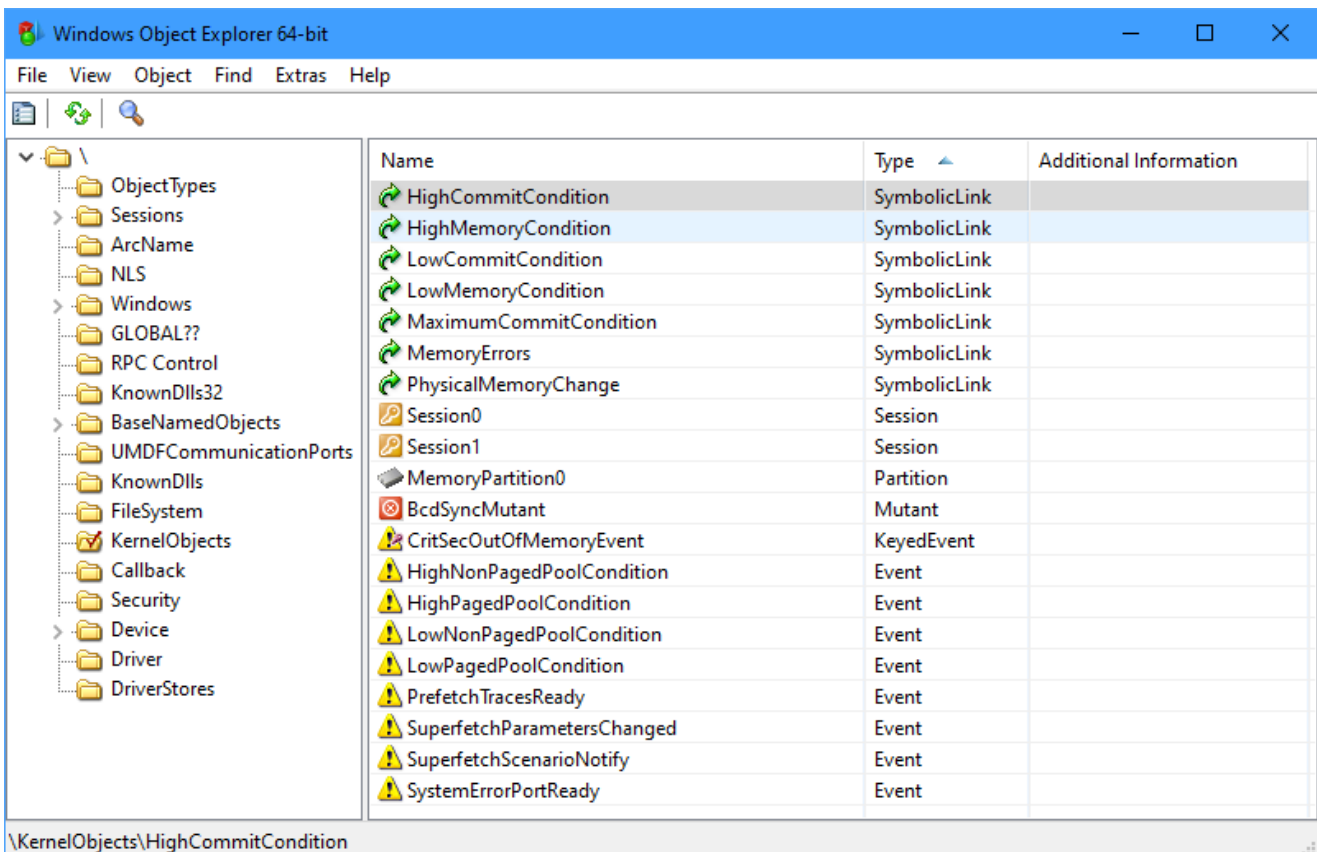
Unfortunately, `ObCreateSymbolicLink` is not exported, so we can't call it ourselves and create a symbolic link with a callback function. It's never that simple. The function has 2 callers – `NtCreateSymbolicLinkObject` and `MiCreateMemoryEvent`. The latter function handles the Memory Partition functionality we described earlier. It creates the various memory events as symbolic links with no target strings, and sets their callback to `MiResolveMemoryEvent`:

```

ObjectAttributes.RootDirectory = 0i64;
ObjectAttributes.SecurityQualityOfService = 0i64;
*(amp;CreateInfo.Flags + 1) = 0;
CreateInfo.Callback = MiResolveMemoryEvent;
ObjectAttributes.Length = sizeof(_OBJECT_ATTRIBUTES);
ObjectAttributes.Attributes = OBJ_CASE_INSENSITIVE|OBJ_KERNEL_HANDLE;
ObjectAttributes.ObjectName = EventName;
ObjectAttributes.SecurityDescriptor = SecurityDescriptor;
CreateInfo.Flags = OB_SYMLINK_TARGET_DYNAMIC;
CreateInfo.CallbackContext = (void *)eventType;
status = ObCreateSymbolicLink(&LinkHandle, SYMBOLIC_LINK_ALL_ACCESS, &ObjectAttributes, &CreateInfo, 0);
if ( status >= 0 )
{
    ObCloseHandle(LinkHandle, 0);
}

```

You can see these symbolic links in `WinObjEx`. They can be recognized by having no target string:



But `MiCreateMemoryEvent` is an internal function that is not very useful for us in this case. So we turn to look at `NtCreateSymbolicLinkObject` which gives us very little to work with:

```
*( _QWORD *)&CreateInfo.Flags = 0i64;
CreateInfo.LinkTarget = linkTarget;
status = ObCreateSymbolicLink(
    LinkHandle,
    DesiredAccess,
    ObjectAttributes,
    &CreateInfo,
    PreviousMode);
```

It always sets the `Flags` for the `OB_SYMLINK_TARGET` structure to `0`, meaning the target is always a string, not a function pointer. This is unfortunate, since it means we can't create symbolic link objects containing callbacks from user mode. But we didn't really expect that to be possible, so we weren't devastated. Instead, we decided to try and modify an existing symbolic link – we can use this feature to hook some frequently used symlink and register our own function to be called whenever it's used.

We chose the symbolic link for the `C:` volume as our target. To achieve our goal, we first needed to open the symbolic link and get its object so we could modify it:

```
NTSTATUS status;
HANDLE symLinkHandle = NULL;
POBJECT_SYMBOLIC_LINK symLinkObject;
UNICODE_STRING symLinkName = RTL_CONSTANT_STRING(L"\\GLOBAL??\\c:");
OBJECT_ATTRIBUTES objectAttributes =
    RTL_CONSTANT_OBJECT_ATTRIBUTES(&symLinkName,
                                   OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE);
//
// Open a handle to the symbolic link object for C: directory,
// so we can hook it
//
status = ZwOpenSymbolicLinkObject(&symLinkHandle,
                                  SYMBOLIC_LINK_ALL_ACCESS,
                                  &objectAttributes);
if (!NT_SUCCESS(status))
{
    goto Cleanup;
}
//
// Get the symbolic link object
//
status = ObReferenceObjectByHandle(symLinkHandle,
```

```

        SYMBOLIC_LINK_ALL_ACCESS,
        NULL,
        KernelMode,
        (PVOID*)&symlinkObject,
        NULL);

if (!NT_SUCCESS(status))
{
    goto Cleanup;
}
//
// Save the original string that the symlink points to
// so we can change the object back when we unload
//
origStr = symlinkObj->LinkTarget;

```

After we got our requested symbolic link object, we needed to save either the target string or the device it would point to, in order to return it from our callback function. Retrieving the device is messy and can have some issues, while the target string is right there in the object itself. We stored it in a global variable, and then we had everything we needed to modify the symbolic link. We just needed to create our callback function:

```

NTSTATUS
SymLinkCallback (
    _In_ POBJECT_SYMBOLIC_LINK Symlink,
    _In_ PVOID SymlinkContext,
    _Out_ PUNICODE_STRING SymlinkPath,
    _Outptr_ PVOID* Object
)
{
    UNREFERENCED_PARAMETER(Symlink);

    //
    // We need to either return the right object for this symlink
    // or the correct target string.
    // It's a lot easier to get the string, so we can set Object to Null.
    //
    *Object = NULL;
    *SymlinkPath = *(PUNICODE_STRING)(SymlinkContext); // OrigStr

    return STATUS_SUCCESS;
}

```

The `symlinkCallback` function receives the symbolic link object, 2 output parameters (only one of which must be set by the function) and a `SymlinkContext` parameter, which is controlled by whoever is registering the function. We chose to use this context to store the

original `LinkTarget` string, so we can set the output `SymlinkPath` parameter to it and send the symlink to its correct destination.

After we defined our callback function, we could go back to our main function and edit the symlink object:

```
//  
// Modify the symlink to point to our callback instead of the string  
// and change the flags so the union will be treated as a callback.  
// Set CallbackContext to the original string so we can  
// return it from the callback and allow the system to run normally.  
//  
symlinkObj->Callback = SymLinkCallback;  
symlinkObj->CallbackContext = &symlinkObj->LinkTarget;  
_MemoryBarrier();  
symlinkObj->Flags |= OBJECT_SYMBOLIC_LINK_USE_CALLBACK;
```

Theoretically, we were done. We could load our driver and every access to the `C:` volume should reach our callback. But as some of you might notice, there is actually a race condition here. Since the callback function and context are part of a union which could also be a Unicode String, the data placed there can be interpreted as the wrong type, causing a type confusion and the inevitable crash.

```
union  
{  
    UNICODE_STRING LinkTarget;  
    struct  
    {  
        PVOID Callback;  
        PVOID CallbackContext;  
    };  
};
```

The type of this data is determined by `OBJECT_SYMBOLIC_LINK.Flags`, but the `Callback` field is too far from the `Flags` field in the `OBJECT_SYMBOLIC_LINK` structure. This means that we can't change both with a single CPU instruction, unless we go into the realm of Intel's Transactional Synchronization eXtensions (TSX), which would allow us to perform all these accesses as a single memory transaction without any races. However, outside of side-channel bugs, there doesn't seem to be any real-world practical uses of TSX, and we'd hate to be the first ones to suggest any, lest this feature be actually uncancelled by Intel.

This means if we change the flags first, someone might try to access this symlink before we changed the `LinkTarget` string and the kernel will try to call a Unicode String as if it was executable memory, leading to a crash. Or if we change the string first and someone tries to use the symlink, the kernel will interpret the lower 2 bytes of our callback address as the

string length and will try to read that many bytes as a string. That can come up to a huge number that will lead to unexpected results, but most likely to reading invalid memory and again, a crash.

We have found a way to get around this issue, and we think it was a pretty clever idea. It did require about 45 minutes of fighting the linker settings, but that's just a price you have to pay sometimes. We also realized that a "simpler" solution is possible as well: creating our own `OBJECT_SYMBOLIC_LINK` with the right settings and then modifying the `OBJECT_DIRECTORY_ENTRY` to swap the pointer of the original object with ours. Because it's a simple pointer, we can use `InterlockedCompareExchangePointer`. Additionally, the `OBJECT_DIRECTORY` has a lock (`EX_PUSH_LOCK`) we could use to make the operation totally safe. But we liked our clever way better (and wanted to show off).

As we mentioned – if we change the string pointer to our function pointer and someone tries to use the symlink before we change the flags, they are going to treat the first 2 bytes of the callback address as a string length and try to parse the "string" based on that. Therefore, we decided to just make sure the lower 2 bytes of the callback address are `0000`, so the length is treated as `0` and no string parsing is attempted. This means we need to align our callback function to `64KB`. Doing that required a lot of attempts and some linker magic, but what eventually worked was this:

- Create a section named `.call$0` and place in it a buffer sized `0xB000`.
- Fill this buffer with zeroes so it won't be optimized out by the compiler. We picked `0xB000` because we noticed the `.text` segment was at `0x5000`, which got our section to therefore be at `0xB000+0x5000` (`0x10000 - 64KB`) bytes.
- Immediately after this, create the normal `.text` segment, where most of our code will be.
- After all the other functions, create an executable section named `.call$1` and place our callback function there.

This is what our driver is going to look like after making these changes:

```
EXTERN_C_START
```

```
__declspec(code_seg(".call$1"))  
NTSTATUS  
SymLinkCallback (  
    _In_ POBJECT_SYMBOLIC_LINK Symlink,  
    _In_ PVOID SymlinkContext,  
    _Out_ PUNICODE_STRING SymlinkPath,  
    _Outptr_ PVOID* Object  
);
```

```
EXTERN_C_END
```

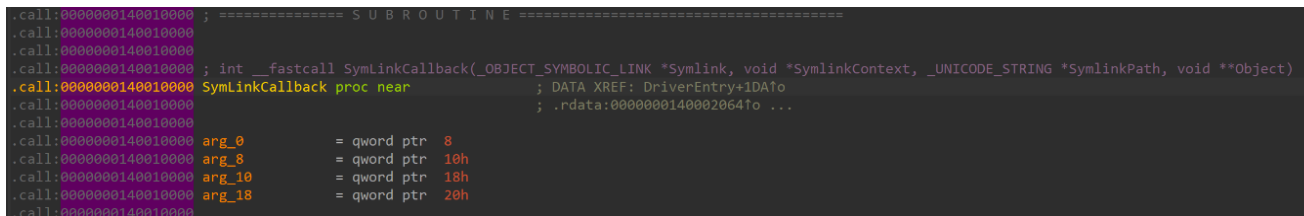
```
#pragma section(".call$0", write)
__declspec(allocate(".call$0")) UCHAR buffer[0xB000] = { 0 };
```

```
#pragma code_seg(".text")
_Use_decl_annotations_
NTSTATUS
DriverEntry (
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath
)
{
    ...
}
```

```
#pragma section(".call$1", execute)
__declspec(code_seg(".call$1"))
```

```
NTSTATUS
SymLinkCallback (
    _In_ POBJECT_SYMBOLIC_LINK Symlink,
    _In_ PVOID SymlinkContext,
    _Out_ PUNICODE_STRING SymlinkPath,
    _Outptr_ PVOID* Object
)
{
    ...
}
```

We compiled our driver and opened it in IDA to see the address of our callback function:



```
.call:00000000140010000 ; ===== S U B R O U T I N E =====
.call:00000000140010000
.call:00000000140010000
.call:00000000140010000 ; int __fastcall SymLinkCallback(OBJECT_SYMBOLIC_LINK *Symlink, void *SymlinkContext, UNICODE_STRING *SymlinkPath, void **Object)
.call:00000000140010000 SymLinkCallback proc near ; DATA XREF: DriverEntry+1DAfo
.call:00000000140010000 ; .rdata:0000000014002064fo ...
.call:00000000140010000
.call:00000000140010000 arg_0 = qword ptr 8
.call:00000000140010000 arg_8 = qword ptr 10h
.call:00000000140010000 arg_10 = qword ptr 18h
.call:00000000140010000 arg_18 = qword ptr 20h
.call:00000000140010000
```

Let's load our driver and see what happens if we try to treat the symbolic link target as a string...

```

5: kd> !object \GLOBAL??\c:
Object: fffffe500139d78e0 Type: (ffff948d14278d10) SymbolicLink
ObjectHeader: fffffe500139d78b0 (new version)
HandleCount: 1 PointerCount: 32770
Directory Object: fffffe5001342b8f0 Name: C:
Flags: 0x000010 ( Local )
Target String is ''
Drive Letter Index is 3 (C:)
5: kd> dx -r0 (nt!_OBJECT_SYMBOLIC_LINK*)(0xfffffe500139d78e0)
(nt!_OBJECT_SYMBOLIC_LINK*)(0xfffffe500139d78e0) : 0xfffffe500139d78e0 [Type: _OBJECT_SYMBOLIC_LINK *]
5: kd> dx @$symlink->LinkTarget
@$symlink->LinkTarget : "" [Type: _UNICODE_STRING]
[<Raw View>] [Type: _UNICODE_STRING]
5: kd> dx @$symlink->LinkTarget.Length
@$symlink->LinkTarget.Length : 0x0 [Type: unsigned short]

```

We dump the symbolic link that we modified, and we can see that when trying to treat our callback address as a Unicode String we get a string with `Length == 0`, which fixes our race condition.

Of course, if we ever want to be able to unload our driver we also need to implement an unload routine that will change the symbolic link back to its original target. We saved the object in a global `symlinkObj` variable, and saved the original `LinkTarget` in a global `origStr` variable, so we can change everything back when we unload:

```

_Use_decl_annotations_
VOID
DriverUnload (
    _In_ PDRIVER_OBJECT DriverObject
)
{
    UNREFERENCED_PARAMETER(DriverObject);

    symlinkObj->Flags &= ~OBJECT_SYMBOLIC_LINK_SANDBOXED;
    _MemoryBarrier();
    symlinkObj->LinkTarget = origStr;

    ObDereferenceObject(symlinkObj);
}

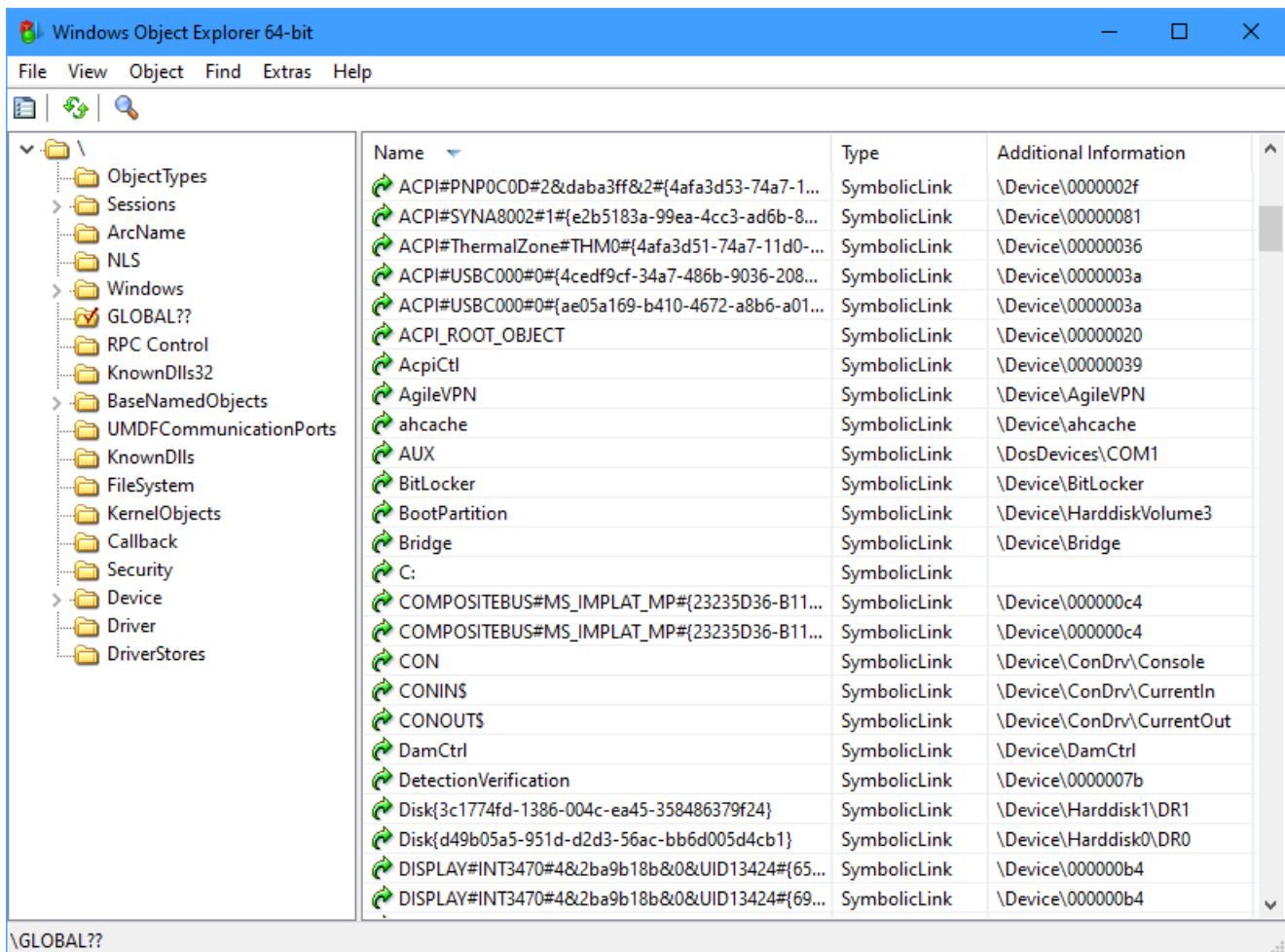
```

It's important to first change Flags and only then the `LinkTarget` to avoid the same race as before. That being said, you probably noticed a second interesting line of code (and unusual in security PoC code) – the call to `_MemoryBarrier()`. You probably already know that compilers reserve the right to re-order any memory operation performed on non-volatile variables (or members), meaning that there's no guarantee that the way we wrote these two lines of C would actually end up matching in assembly code.

To solve this, Visual C/C++ includes inline functions such as `_ReadWriteBarrier()`. However, modern processors themselves can also choose to re-order memory operations at a hardware level, meaning that these two writes could also happen in a different sequence (the

same thing can sometimes happen for reads too). To solve the hardware re-ordering issue, we need to use a fence instruction, which is what `_MemoryBarrier()` does.

Now we have a working callback function that gets called whenever anyone tries to access the `C:` volume. This method is not visible and will not be detected unless specifically searched for. `WinObjEx` will show this symbolic link as having no target, which will only look suspicious to someone looking for this technique (although some existing legitimate symbolic links, such as the ones in `\KernelObjects`, already look like this).



You can get the source code for this simple rootkit driver from our GitHub repo [here](#).

But there is one more thing we want to achieve – getting the full path requested by the caller every time the symbolic link is being accessed. This would let us, for example, return different files or directories depending on who the caller is, as well as monitor accesses. Our callback does not receive this information, and none of us wanted to implement stack parsing to find it. As such, we started looking for a different option... unfortunately, it seemed our last blog post was over 40 pages, and while we heard this was useful for people taking entire afternoons off at certain organizations, we'll break things up this time and see you in a future Part 2 !

Read our other blog posts:

