# Symbolic Hooks Part 2 : Getting the Target Name

windows-internals.com/symhooks-part-two

By Yarden Shafir & Alex Ionescu

In our last blog part, we concluded with a working callback, but no information about the path being opened. Of course, we could get it from the stack since it should be saved there somewhere, but we thought there must be a more elegant way. We also wanted to avoid writing a book on *Unwind Opcodes* and how they can be used to recover stack parameters efficiently.

And so, we to go a different path, and come up with a way to force our own sort of parse routine to execute, in which we could get the original path, and take a decision as to whether or not to redirect the caller. Two options came to mind:

- We could create a new object type with `ObCreateObjectTypeEx`, implement our own `ParseRoutine`, and have the symlink redirect to an object of our type so that we can have our routine return `STATUS_REPARSE`, with the name of the original target Device Object.
- We could create a new Device Object with `IoCreateDevice`, implement our own `IRP_MJ_CREATE` handler, and have it use the I/O Manager's existing reparsing logic (which it calls *transmogrification*) so that we can return `STATUS_REPARSE` and a new name for any File Object it creates, which would re-direct it to the original target Device Object.

Ultimately, creating a new object type is undocumented, monitored by Patch Guard if we make any wrong moves, and, most importantly, does not have a matching API to undo/destroy the operation. Yep, there is no way to delete an object type, thus our driver would never be able to unload.

Therefore, we decided to have our symlink callback redirect the symbolic link to a Device Object we will create, instead of returning the original string. Then, when our Device Object's `IRP_MJ_CREATE` handler is called, the I/O Manager has already created a File Object, and we can get it from the `IRP`, retrieve its name, plus any other information about it and the creator/caller.

Thus, we first create our device – `\Device\HarddiskVolume0`. Next, we get the symbolic link to the `C:` volume the same way we showed in Part 1, and modify it to point to our callback as the `LinkTarget`. Then, we only must make one change: instead of passing the original link target string as a parameter in `SymlinkContext`, we pass in the path of our new device:

```
_Use_decl_annotations_
NTSTATUS
DriverEntry (
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
    )
{
    NTSTATUS status;
    HANDLE symLinkHandle;
    DECLARE_CONST_UNICODE_STRING(symlinkName, L"\\GLOBAL??\\c:");
    OBJECT_ATTRIBUTES objAttr =
RTL_CONSTANT_OBJECT_ATTRIBUTES(&symlinkName,

 OBJ_KERNEL_HANDLE |

 OBJ_CASE_INSENSITIVE);
    UNREFERENCED_PARAMETER(RegistryPath);

    //
    // Make sure our alignment trick worked out
    //
    if (((ULONG_PTR)SymLinkCallback & 0xFFFF) != 0)
    {
        status = STATUS_CONFLICTING_ADDRESSES;
        DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                   DPFLTR_ERROR_LEVEL,
                   "Callback function not aligned correctly!\n");
        goto Exit;

    //
    // Set an unload routine so we can update during testing
    //
    DriverObject->DriverUnload = DriverUnload;

    //
    // Open a handle to the symbolic link object for C: directory,
    // so we can hook it
    //
    status = ZwOpenSymbolicLinkObject(&symLinkHandle,
                                      SYMBOLIC_LINK_ALL_ACCESS,
                                      &objAttr);
    if (!NT_SUCCESS(status))
    {
        DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                   DPFLTR_ERROR_LEVEL,
```

```
                      "Failed opening symbolic link with error: %lx\n",
                      status);
         goto Exit;

     //
     // Get the symbolic link object and close the handle since we
     // no longer need it
     //
     status = ObReferenceObjectByHandle(symLinkHandle,
                                        SYMBOLIC_LINK_ALL_ACCESS,
                                        NULL,
                                        KernelMode,
                                        (PVOID*)&g_SymLinkObject,
                                        NULL);
     ObCloseHandle(symLinkHandle, KernelMode);
     if (!NT_SUCCESS(status))
     {
         DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                    DPFLTR_ERROR_LEVEL,
                    "Failed referencing symbolic link with error: %lx\n",
                    status);
         goto Exit;

     //
     // Create our device object hook
     //
     RtlAppendUnicodeToString(&g_DeviceName, L"\\Device\\HarddiskVolume0");
     status = IoCreateDevice(DriverObject,
                             0,
                             &g_DeviceName,
                             FILE_DEVICE_UNKNOWN,
                             0,
                             FALSE,
                             &g_DeviceObject);
     if (!NT_SUCCESS(status))
     {
         //
         // Fail, and drop the symlink object reference
         //
         ObDereferenceObject(g_SymLinkObject);
         DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                    DPFLTR_ERROR_LEVEL,
                    "Failed create devobj with error: %lx\n",
                    status);
         goto Exit;
```

```
    //
    // Attach our create handler
    //
    DriverObject->MajorFunction[IRP_MJ_CREATE] = SymHookCreate;

    //
    // Save the original string that the symlink points to
    // so we can change the object back when we unload
    //
    g_LinkPath = g_SymLinkObject->LinkTarget;

    //
    // 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.
    //
    g_SymLinkObject->Callback = SymLinkCallback;
    RtlAppendUnicodeStringToString(&g_DeviceName, &g_TailName);
    g_SymLinkObject->CallbackContext = &g_DeviceName;
    MemoryBarrier();
    g_SymLinkObject->Flags |= OBJECT_SYMBOLIC_LINK_USE_CALLBACK;

Exit:
    //
    // Return the result back to the system
    //
    return status;
}
```

This code means that when someone tries to access the symlink they will reach our callback and will receive the path to our Device Object path ( `\Device\HarddiskVolume0` ) instead of `\Device\HarddiskVolume<N>` , where `N` is the real `C:` partition.

Then, when this path will be opened, the I/O manager will create a File Object for the *remaining path*, such as `\Windows\Notepad.exe` , and will then call our Driver Object's `IRP_MJ_CREATE` handler, where we will get this name from the `FILE_OBJECT` structure, and replace it with a new, fully qualified path, including both the original Device Object path and the remaining path.

Replacing a `FILE_OBJECT` name is trickier than it sounds – the original path, allocated by the I/O Manager, has a specific pool tag, and us freeing it and allocating our own would look like a leak to various testing tools such as Driver Verifier, unless we mimic the original tag.

To fix this issue, Microsoft implemented a special API: `IoReplaceFileObjectName`. Not only does it use the correct internal kernel pool tag, but it also implements certain optimizations such that the length of the file name string buffer will always be "aligned" to `56`, `120`, or `248` bytes (unless the name is bigger, in which case the precise size is used). This avoids having to free/re-allocate the buffer in many situations, as the new name can simply override the old.

Here's how creating this new name ends up looking like:

```
//
// Get the FILE_OBJECT from the I/O Stack Location
//
ioStack = IoGetCurrentIrpStackLocation(Irp);
fileObject = ioStack->FileObject;

//
// Allocate space for the original device name, plus the size of the
// file name, and adding space for the terminating NUL.
//
bufferLength = fileObject->FileName.Length +
               g_LinkPath.Length +
               sizeof(UNICODE_NULL);
buffer = (PWCHAR)ExAllocatePoolWithTag(PagedPool, bufferLength, 'maNF');
if (buffer == NULL)
{
    status = STATUS_INSUFFICIENT_RESOURCES;
    goto Exit;

//
// Append the original device name first
//
buffer[0] = UNICODE_NULL;
NT_VERIFY(NT_SUCCESS(RtlStringCbCatNW(buffer,
                                      bufferLength,
                                      g_LinkPath.Buffer,
                                      g_LinkPath.Length)));

//
// Then add the name of the file name
//
NT_VERIFY(NT_SUCCESS(RtlStringCbCatNW(buffer,
                                      bufferLength,
                                      fileObject->FileName.Buffer,
                                      fileObject->FileName.Length)));

//
```

```
// Ask the I/O manager to free the original file name and use ours instead
//
status = IoReplaceFileObjectName(fileObject,
                                 buffer,
                                 bufferLength - sizeof(UNICODE_NULL));
if (!NT_SUCCESS(status))
{
    DbgPrintEx(DPFLTR_IHVDRIVER_ID,
               DPFLTR_ERROR_LEVEL,
               "Failed to swap file object name: %lx\n",
               status);
    ExFreePool(buffer);
    goto Exit;
}
```

Once we're replaced the File Object's name, this code still has a problem – we can't return `STATUS_SUCCESS` , since that would make us the owner Device Object for this new file, and not actually point to the original target Device Object of the partition. All future I/O will flow through our driver as `IRP` s, and we must now essentially implement forwarders for every operation.

We *could* get the correct Device Object for `\Device\HarddiskVolume<N>` and manually forward all `IRP` s to it, but then all requests will still be attached to our device. Not only does this make us a lot more visible, but it essentially turns is into a file system filter driver. We just want to get the creation request and then pass it on to the correct device and not have to handle it ever again.

To make this work correctly, we have to exercise the I/O Manager's transmogrification logic, which is a two-step process:

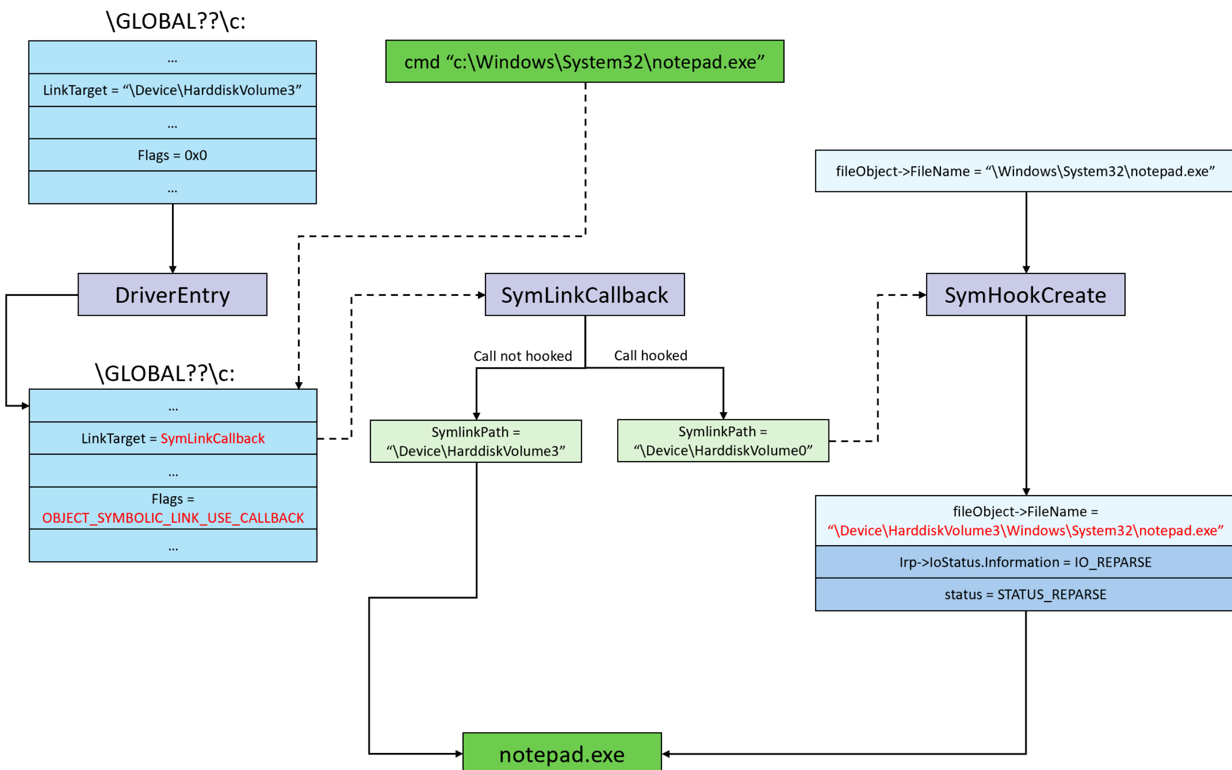
1. Return `STATUS_REPARSE` , to indicate that a reparse operation is needed. This causes `IopParseDevice` to look at the new name string in the File Object, and begin the name lookup logic all over again, based on this new name, freeing the old object and previous work done. This code is highly complex, but you can see a simpler version of it in the ReactOS sources here.
2. Set the IRP's Information field to `IO_REPARSE` , which indicates the type of reparsing operation that we are attempting. This is normally where a true hard link or symlink would be indicated by using a special *reparse tag* and a matching structure documented by Microsoft, such as REPARSE_DATA_BUFFER. However, `IO_REPARSE` is a magic/reserved value which indicates just a plain replacement of the name, and not a true *reparse point*.

Taking these points into consideration, our `IRP_MJ_CREATE` handler completes with the following logic:

```
    //
    // Return a reparse operation so that the I/O manager uses the new
file
    // object name for its lookup, and starts over
    //
    Irp->IoStatus.Information = IO_REPARSE;
    status = STATUS_REPARSE;

Exit:
    //
    // Complete the IRP with the relevant status code
    //
    Irp->IoStatus.Status = status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
```

So now we have a mechanism that looks something like this:




Some of you might point out that this method will work just as well without using a symlink callback at all – we could have just replaced the `LinkTarget` of the symbolic link with the path of our device, and would have gotten all the requests anyway – in fact, we'd only have had to change the last digit of the path. However, we felt that doing this makes us a lot more visible, as anyone inspecting the symlink object will easily see this path, as well as a change in the structure.

Another reason is that with the callback we can dynamically decide what to do. For example, if we know we are being inspected by an AV driver, we can use the callback to return the original string and not redirect the request to our device. We could even redirect to a completely different device if we wanted to, without having to constantly keep changing the path (which would result in a race condition anyway).

Excited to try things out, we load our new and improved driver and look at the results:

```
841 1.38116026   Opening file \
842 1.38163102   Opening file \Windows\
843 1.38216186   Opening file \Windows\system32\WindowsPowerShell\
844 1.38399053   Opening file \Windows\system32\WindowsPowerShell\v1.0\powershell.exe
845 1.38526225   Opening file \Windows\System32\WindowsPowerShell\v1.0\powershell.exe
846 1.38635075   Opening file \Windows\System32\WindowsPowerShell\v1.0\powershell.exe
847 1.38718069   Opening file \Windows\System32\WindowsPowerShell\v1.0\powershell.exe.Config
848 1.38824213   Opening file \Windows\System32\WindowsPowerShell\v1.0\powershell.exe
849 1.38996887   Opening file \Windows\system32\WindowsPowerShell\v1.0\powershell.exe
850 1.39035892   Opening file \Windows\system32\WindowsPowerShell\v1.0\powershell.exe
851 1.39116681   Opening file \Windows\system32\WindowsPowerShell\v1.0\powershell.exe
852 1.39219761   Opening file \Windows\system32\WindowsPowerShell\v1.0\powershell.exe
853 1.39626694   Opening file \Users\YardenShafir\AppData\Local\Microsoft\Windows\Explorer
854 1.39868104   Opening file \Users\YardenShafir\AppData\Local\Microsoft\Windows\Explorer
855 1.40071106   Opening file \Windows\system32\windowspowershell\v1.0\powershell.exe
856 1.40153337   Opening file \Windows\system32\windowspowershell\v1.0\powershell.exe
857 1.40248466   Opening file \Windows\system32\windowspowershell\v1.0\powershell.exe
858 1.40345633   Opening file \Windows\system32\windowspowershell\v1.0\powershell.exe
859 1.40825105   Opening file \Users\YardenShafir\AppData\Local\Microsoft\Windows\Explorer
860 1.41926789   Opening file \Users\YardenShafir\AppData\Local\Microsoft\Windows\Explorer
```

And get super happy, until, about 10 seconds later, we get a crash:

Shit. When our device receives an open request for the root of the C: volume itself, it can't return `STATUS_REPARSE` , it's in the rules.

So what do we do now? All will be revealed in part 3 (and 4… and possibly 5).

We have created a new branch on GitHub which implements the improved hooking mechanism introduced in this part.

Read our other blog posts: