Symbolic Hooks Part 3: The Remainder Theorem

windows-internals.com/symhooks-part-three

By Yarden Shafir & Alex Ionescu

We ended the second part with, unsurprisingly, a bugcheck. We tried to redirect all access to the `C:` volume to our device in order to get information about all the paths that are being accessed, but the first time anyone tried opening the `C:` volume itself, the I/O manager threw a `DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN` blue screen at us.

Unfortunately, we can't return any other status code than `STATUS_REPARSE` or the path will not be parsed properly and a lot of things will break in the system as our fake device now becomes the "file system" of this poor path. But what if we could find a way to never have to return `STATUS_REPARSE` for volume opens, because we never see a volume open to begin with?

First, we should probably understand what it means to have a *volume open*. While based on the ancient Windows Server 2003 code base, ReactOS can offer a clue here — as it contains the exact same behavior in IopParseDevice:

```
//
// In case we override checks, but got this on volume open, fail hard
//
if (OpenPacket->Override != FALSE)
{
    KeBugCheckEx(DRIVER_RETURNED_STATUS_REPARSE_FOR_VOLUME_OPEN,
                 (ULONG_PTR)OriginalDeviceObject,
                 (ULONG_PTR)DeviceObject,
                 (ULONG_PTR)CompleteName,
                 OpenPacket->Information);
}
```

We can see that `Override` is set at this line underneath the following if statement:

```
//
// Now check if we need access checks
//
if (((AccessMode != KernelMode) || (OpenPacket->Options &
IO_FORCE_ACCESS_CHECK)) &&
    ((OpenPacket->RelatedFileObject == NULL) || (VolumeOpen != FALSE)) &&
    (OpenPacket->Override == FALSE))
{
```

This leaves us with the final question — how does `VolumeOpen` become `TRUE`? This line provides the answer:

```
//
// Check if this is a volume open
//
if ((OpenPacket->RelatedFileObject != NULL) &&
    (OpenPacket->RelatedFileObject->Flags & FO_VOLUME_OPEN) &&
    (RemainingName->Length == 0))
{
    //
    // It is
    //
    VolumeOpen = TRUE;
}
```

In other words, if a file object is being opened on top of an existing file object that represents a volume, and this new file object doesn't have a `RemainingName`, then we are directly opening the volume represented by `RelatedFileObject` itself. This is exactly what happens when we open `C:`.

James Forshaw provided us with an interesting idea – what if we could make it so that our device never receives a path that's seen as a volume open by the I/O manager? In other words, what if `RemainingName` would never be `0`?

James' suggestion was pretty simple. Instead of redirecting the symlink through the callback to `\Device\HarddiskVolume0` (the name of our device), we'll redirect it to `\Device\HarddiskVolume0\Foo`. That way, all paths reaching our device will start with `\Foo`, and none of them will be treated by the I/O Manager as a volume open, so returning a `STATUS_REPARSE` should not present any issues. We'll just need to remove this suffix from the path and set the file name to the correct string before returning.

First, we define our suffix:

```
DECLARE_GLOBAL_CONST_UNICODE_STRING(g_TailName, L"\\Foo");
```

And when defining the Device Object name that we want the symbolic link callback to return, we now append this string in the `DriverEntry`:

```
RtlAppendUnicodeStringToString(&g_DeviceName, &g_TailName);
```

Finally, we make some changes to our `IRP_MJ_CREATE` handler. First, the final name buffer must remove the space of the `\Foo` suffix in the original file name:

```
//
// Allocate space for the original device name, plus the size of the
// file name, minus "\Foo", and adding space for the terminating NUL.
//
bufferLength = fileObject->FileName.Length -
               g_TailName.Length +
               g_LinkPath.Length +
               sizeof(UNICODE_NULL);
```

And then, we must skip past the suffix when concatenating the file name:

```
//
// Then add the name of the file name, minus "\Foo"
//
NT_VERIFY(NT_SUCCESS(RtlStringCbCatNW(buffer,
                                      bufferLength,
                                      fileObject->FileName.Buffer +
                                      (g_TailName.Length /
sizeof(g_TailName.Buffer[0])),
                                      fileObject->FileName.Length -
                                      g_TailName.Length)));
```

That’s pretty much it! With these simple changes, the driver should no longer crash. However, there’s still a subtle bug here: while our symbolic link callback will guarantee that there’s always a `\Foo` present, there’s other ways that our `IRP_MJ_CREATE` handler could be reached: if someone directly attempts to open `\Device\HarddiskVolume0` from the kernel or with a native API. One such example is the `WinObjEx64` tool from `hfiref0x` — when double-clicking on our device object, we immediately crashed. So let’s be safe, and simply prohibit direct opens of our device, which would not have the required `\Foo` suffix, by adding one last block:

```
//
// If this is someone directly trying to access our device object,
// fail them, so that we do not crash the system (since we should
// not reparse direct opens).
//
if (fileObject->FileName.Length < g_TailName.Length)
{
    status = STATUS_ACCESS_DENIED;
    goto Exit;
}
```

We loaded our new driver, which you can now find on our GitHub repository here, and this time, we got all the paths that were accessed in the `C:` volume, and no machine crashes! We celebrated our victory with a drink, then another, and another. And then we noticed things on the machine didn’t work too well. Processes such as Calculator wouldn’t run, the Start Menu refused to show up, and pretty soon the machine was basically unusable. So we had another drink to handle this additional failure, and passed out.

We eventually did figure out this issue during a long flight, but that story will be told in part `4`. We promise that’s the last part. It all worked afterward, which is why none of your machines are showing any symptoms.

Read our other blog posts: