Get Social with OSR 
Real -Time Updates

Just in case you’re not already following us on Twitter, Facebook, LinkedIn, or via our own “osrhints” distribution list, below are a few of the more recent contributions that are getting attention in the Windows driver development community:

Unexpected FltGetFileNameInformation Behavior for Network Renames
A couple of weeks ago I was teaching our Developing file System Minifilters for Windows seminar here in Manchester, NH. A student asked a question about a behavior they were

How L1 Terminal Fault (L1TF) Mitigation and WinDbg Wasted My Morning
I’ve been doing some research into the Windows Filtering Platform and the information available at each of the various filtering layers. In particular, I’ve been focusing on the information available in Windows 7 as that predates some ETW trace points...

Three-Plus Years Later...Driver Signing Still Baffles
It was back in 2015 that I wrote my first set of blog posts on Windows driver signing. Then I wrote some more in 2016. And then in 2017...
https://www.osr.com/blog/2019/06/03/three-plus-years-later-driver-signing-still-baffles/

Withdrawning From the Microsoft MVP Program
When I was first named a Microsoft Most Valuable Professional, back in the early 2000’s, I was very proud. Really, I was. There was a cohort of smart, generous, and engaged engineers who were named “DDK MVPs” at the same time that I was appointed. And I am pleased to say that I personally considered a good number of them. Engagement between Microsoft and the third-party driver development community was at a high.
https://www.osr.com/blog/2019/04/08/withdrawing-from-the-microsoft-mvp-program/

PSA: FsRtlIsNameInExpression Can Raise an Exception
Well, THIS one was a surprise...After triggering a memory leak in a driver, the system surprisingly crashed due to a call to FsRtlIsNameInExpression...
https://www.osr.com/blog/2019/03/04/psa-fsrtlisnameinexpression-can-raise-exception/

NTFS Status Debugging
As a file system filter developer, one of the great pains in life is when a file system operation fails deep in the bowels of the file system. For example, say I’m trying to rename a file with FltSetInformationFile for FileRenameInformation and I get back STATUS_ACCESS_DENIED. How do I track that down? Sure, I could try single stepping through the function until I see a STATUS_ACCESS_DENIED, but that could take quite a while. Even worse, if the file system is NTFS I will undoubtedly end up stepping into some other thread and losing the thread context of my failing rename...
https://www.osr.com/blog/2018/10/17/ntfs-status-debugging/

Follow us!  

The NT Insider  
Sep - Oct 2019  
© OSR Open Systems Resources, Inc.
For those of you who have been asking for years about an online attendance option for OSR’s seminars, the wait is over. This summer, our very first online audience joined us at our FS Minifilter seminar presented at OSR’s Seminar Space in Manchester, NH. The good news is that it went very well. The better news is that it went so well that we immediately began plans to permit online attendance at future offerings.

So what’s it like to attend online? During your week-long seminar presentation, you will have a live-stream feed of audio and video of your instructor presenting content of the seminar. The instructor will use multiple camera views (e.g., whiteboard view, close-up code view) to highlight specific portions of the presentation, while using “classroom” views as the standard to provide a sense of participation for online participants. Live, in-person attendees, and you, can ask questions. It’s like being at the seminar, or rather…as close an approximation as we can reasonably manage. But hey…you don’t have to spend $$$ on travel, or haul yourself across the country/globe to attend! So what’s your next opportunity to join us in-person or online?

OSR live-streams seminars to an online audience!

Now Available Online!
Live-Streamed via the Web

Need To Know WDF?
Tip: You can read all the articles ever published in The NT Insider and still not come close to what you will learn in one week in our WDF seminar. So why not join us?

Both Scott and Peter have in-depth knowledge and extensive hands-on experience writing device drivers. Their discussions about mistakes to avoid was as valuable as explaining Windows. This was the best training class that I have ever taken.

- Feedback from an attendee of THIS seminar


Next presentation:
Manchester, NH (OSR) & ONLINE
21-25 October
Custom Adapts Itself to Expediency†
Creating Custom WDF Objects

Windows 8 introduced many changes to the driver ecosystem: A new OS for Windows Phone, a heightened focus on power efficiency, and support for Simple Peripheral Bus devices being some of the most notable. Less noticed, tucked-in as part of KMDF 1.11 (and conspicuously missing from the summary of changes in the WDK documentation), was the addition of the ability for WDF drivers to define their own custom Object types.

A Quiet Introduction
The fact that this new feature came without a great deal of fanfare from our friends in Redmond wasn’t much of a surprise. Custom Object types are primarily intended for use by Class Extension developers, and several new such Extensions were introduced as part of Windows 8 (among them SPBCx, SerCx, GpioClx). The idea behind these Class Extensions (or “Framework Extensions” as the WDK docs sometimes label them) is that they extend the scope of WDF beyond just its native Objects and provide support for additional devices types and architectures. A Class Extension is implemented as a DLL, and when you build a driver that uses a Class Extension, you link it both with the WDF Framework Library and the Class Extension Library into what the WDK docs somewhat oddly refer to as a “driver triple.”

While we’ve had the ability to create custom WDF Objects for years now, there hasn’t been any discussion about their use or example of how to create them. Until now.

The Need to Extend the Framework
I think we all agree that WDF has taken hold and finally supplanted WDM as the primary interface for writing Windows drivers. As new or unique device architectures are supported by Windows, it makes sense to extend the capabilities of WDF to support these new or specialized device types. This shortens the required “ramp-up” time for driver devs by allowing them to apply already familiar WDF design patterns to the development of drivers for additional types of devices.

The initial way extensions to the Framework were implemented was by changing the core code of the Framework itself. USB support in WDF is the primary example of this approach. USB support in WDF gives us a big pile of USB-specific APIs, and even a unique pair of WDF I/O Target types.

However, adding specific extensions to KMDF and UMDF for every additional device architecture to be supported wasn’t a reasonable long-term plan. First, there are only a limited number of WDF developers, and their job is to keep the Framework running smoothly from release to release. Second, adding support for a new device type typically requires deep knowledge of that device architecture. That means the right group of folks to add that support are the folks who are experts in the device technology, not folks who are experts in KMDF or UMDF internals. Clearly, a way to add support for additional device types without having to alter the mainline WDF code was required. And thus, Class Extensions were born.

Today we have all manner of Class Extensions, including some very clever ones from the network group. But that’s a subject for another article. Let’s turn our attention back to the topic of creating custom Objects.

You Can Create Your Own Object
Regardless of why the facility was created, WDF (both KMDF and UMDF) now allows developers to define their own custom Object types. More precisely, it allows developers to create custom Object types that are based on other, preexisting, WDF Objects types (including the generic WDFOBJECT). It’s almost like deriving a child class from a base class. But in C and not C++.

The beauty of custom WDF Objects is that, regardless of what you call them or how you use them, they are actual native WDF Objects. That means, when properly implemented, they can be used anyplace a WDF Object can be used. For example, you can store a custom Object in a WDFCOLLECTION. A user of your custom Object can allocate one (or more) WDF Contexts that are associated with your Object and, just like native WDF Objects, the Framework will manage the lifetime of that Context.

The differences between your custom Object and the underlying native WDF Object on which its based are:

- Your custom Object can (and almost always will) have its own internal data area. This is like a WDF Context, but it is not accessible by users of your custom Object.

† Tacitus

(CONTINUED ON PAGE 5)
WDF Objects... (Cont.)

(CONTINUED FROM PAGE 4)

- Your custom Object can support custom methods that you devise, including custom Event Processing Callbacks.

In implementing a custom WDF Object, you choose the native WDF Object that best matches the characteristics that you want your Object to have or the way that you want your Object to be used. For example, if you want your Object to be able to represent an I/O operation and to be able to be stored on a WDFQUEUE, you would base your Object on the native WDFREQUEST. Note that you optionally can expose these capabilities of your custom Object to the eventual user of your custom Object. And, of course, because this is WDF, your custom Object could even represent a group of multiple WDF Objects, with one parented on the other. Bottom line, the system is extremely flexible.

No question, custom WDF Objects can be both useful and fun. Later, I’ll talk more about when I think it’s most appropriate to create custom WDF Objects. But, for now, suffice it to say that properly implementing a custom WDF Object takes a reasonable amount of effort and is probably something you only want to do if you plan to create a custom Object type that you’d want to re-use across multiple projects. Sort of like a Class Extension.

How To “Roll Your Own”
I tend to think of the process of defining custom WDF Objects as comprising two parts:

1) Declaring the Object and its handle — This provides all the necessary naming and glue for the custom Object.
2) Providing the Custom Infrastructure -- These are the macros and functions that you create, following the established WDF design patterns, to support the use and operation of your Object.

(CONTINUED ON PAGE 6)
To illustrate the process, we’re going to create our own custom WDF Object. By the way, an important best practice to follow is to
never name custom Objects starting with WDF. Names starting with WDF are indicative of Object supported directly by the
Framework. Thus, we have very unimaginatively named the example custom Object type that we’ll be using in this article
OSRCUSTOM.

### Step 1: Declaring the Object and Its Handle

Recall that, in WDF, the data type that contains a handle to a WDF Object is the name of that Object type. So, the datatype
WDFDEVICE is the handle to a WDF Device Object and the datatype WDFTIMER is the handle to a WDF Timer Object.

Given this naming convention, we implement Step 1 (above) by declaring the handle and data type of our new custom Object:

```c
// OSR Custom Declaration
DECLARE_HANDLE(OSRCUSTOM);
WDF_DECLARE_CUSTOM_TYPE(OSRCUSTOM);
```

In this code, we declare a WDF Data Type and handle type named OSRCUSTOM. WDF very cleverly takes care of all the necessary
declarations and mess for you. And with that, Step 1 “Declaring the Object and its handle”, is complete.

The WDF_DECLARE_CUSTOM_TYPE macro is pretty interesting, if you take the time to look at its definition. While it’s very simple
to use, it’s the piece of code that defines a clever little function that you’ll use to instantiate your custom Object later on when you
call WdfObjectAddCustomTypeWithData. But, more about that later. Let’s not get too far ahead of ourselves.

The DECLARE_HANDLE directive allows users who are going to use our Object to declare and use Object handle variables in the
usual WDF way (shown here as part of an EvtIoDeviceControl Event Processing Callback):

```c
VOID NothingEvtDeviceControl(WDFQUEUE Queue,
    WDFREQUEST Request,
    size_t OutputBufferLength,
    size_t InputBufferLength,
    ULONG IoControlCode)
{
    PNOTHING_DEVICE_CONTEXT devContext;
    WDFDEVICE device;
    NTSTATUS status;
    OSRCUSTOM custom;
    device = wdfIoQueueGetDevice(Queue);
}
```

Easy, right? Two macro invocations and we’re done! Well, yes, at least so far. But Step 2 is where the real work takes place.

### Step 2: Adding the Infrastructure

In Step 2 we add the infrastructure, or what I like to refer to as the “WDF Gloss”, to your custom Object type. As part of this step,
you create macros and functions that implement and facilitate common WDF design patterns, and that allow your Object to do
useful things.

Let’s continue with the development of our OSRCUSTOM Object. Because it’s just an example, the OSRCUSTOM Object doesn’t
implement anything that’s actually useful. It implements a counter. Using the Object you can:

- Provide an initial value of the “Counter” property when you instantiate the Object.
- Specify an Event Processing Callback (when you instantiate the Object) that is invoked whenever the
  Object’s Counter property is incremented to an integer multiple of ten. We call this the Object’s
  EvtOsrCustomInformCount Event Processing Callback.
- Increment the Object’s Counter property (by invoking a method)
- Get the current value of the Object’s Counter property (by invoking a method)

(CONTINUED ON PAGE 7)
We base our OSRCUSTOM Object on the generic WDFOBJECT, because there are no other Object-type “features” that we need. But, as mentioned previously, you can base your custom Object on any native WDF Object type your wish. If you wanted to implement some sort of special timer, for example, you could base your custom Object on WDFTIMER.

When crafting your new Object, it’s important to follow as much of the established WDF design pattern as possible. For example, in creating our OSRCUSTOM Object, the user should be able to follow the standard 3 step pattern for WDF Object Instantiation: (1) Specify WDF Object Attributes, (2) Specify Object-specific configuration using a XXX_CONFIG structure, (3) Instantiate the Object using a XxxxCreate function (where Xxxx is the Object name). Of course (or, perhaps I should say, unfortunately), there’s nothing in the process of defining a custom WDF Object that forces you to follow these conventions. But one of the major features of WDF is its consistency. You can leverage that feature by making your custom Object “work” as much like a native WDF Object as possible.

The steps necessary for a user to instantiate our OSRCUSTOM Object should look pretty familiar to any WDF driver developer (See Figure 1, below).

The code in Figure 1, checks to see if an OSRCUSTOM Object has already been created, and if it has not it creates one using the standard WDF Object creation pattern. There’s an OSR_CUSTOM_CONFIG structure that’s used as the Object’s configurator. That structure is initialized by a macro named OSR_CUSTOM_CONFIG_INIT. The _INIT macro takes both a pointer to the configurator and a value that will be used to initialize the OSRCUSTOM’s Counter property.

The OSRCUSTOM Object is then instantiated by calling OsrCustomCreate, passing:

- The handle to an associated WDFDEVICE (which will be the parent of the OSRCUSTOM)
- A pointer to the initialized OSR_CUSTOM_CONFIG structure
- An optional pointer to a WDF_OBJECT_ATTRIBUTES structure (which is not supplied in the example)
- A pointer of type OSRCUSTOM into which to return the handle of the newly instantiated Object, if the call to OsrCustomCreate is successful.

```c
devContext = NothingGetContextFromDevice(device);
if (devContext->OsrCustom == nullptr) {
    OSR_CUSTOM_CONFIG customConfig;
    OSRCUSTOM newCustom;
    DbgPrint("Creating new 'Custom'!\n");
    OSR_CUSTOM_THING_CONFIG_INIT(&customConfig,22); // Config pointer and Initial Counter value
    status = OsrCustomCreate(device,
                              customConfig,
                              WDF_NO_OBJECT_ATTRIBUTES,
                              &newCustom);
    if (!NT_SUCCESS(status)) {
        DbgPrint("OsrCustomCreate failed 0x%0x\n", status);
        goto done;
    }
    devContext->OsrCustom = newCustom;
    goto done;
}
```

Figure 1 – Instantiating an OSRCUSTOM Object
WDF Objects... (Cont.)

(CONTINUED FROM PAGE 7)

The code to support this is simple, but it requires some thought. We can start by looking at how the configurator is handled (Figure 2, below).

In Figure 2, we provide the declaration of the OSR_CUSTOM_CONFIG structure, which includes a field that contains the starting value for the Counter, as well as a pointer to an optional EvtOsrInformCount Event Processing Callback. The OSR_CUSTOM_CONFIG_INIT function does the expected initialization of the structure (zeroing it) and fills-in the CounterStartValue field.

The code that implements OsrCustomCreate is a bit more involved, but still... not difficult. You can see this code is Figure 3 (next page).

In Figure 3, you can see that we create a WDF_OBJECT_ATTRIBUTES structure if one is not provided by the caller. Because it’s key to make your Object creation process as much like that of a native WDF Object as possible, it’s important for you to follow the usual WDF procedures with respect to Object creation. This means allowing the caller to specify _OBJECT_ATTRIBUTES that will accompany the instance of your Object. This does not, however, mean that you must allow any combination of _OBJECT_ATTRIBUTES that the caller provides. For example, we force the ParentObject field to the handle of the WDFDEVICE that’s passed-in to the creator. It is not unusual in WDF for an Object to require a specific parent. As a side note, it’s interesting that there does not appear to be any function that allows you to check the type of a passed-in WDF Object handle (or to even check if the handle is valid at all). It would be nice to have this, to allow us to validate that the user did indeed pass us a valid handle to a WDFDEVICE.

Using the WDF_OBJECT_ATTRIBUTES structure, we instantiate a generic WDFOBJECT by calling WdfObjectCreate. Again, we could have based our custom Object on any native WDF Object type. And, by so doing, our custom Object would effectively adopt the characteristics of that native Object.

```c
typedef _OSR_CUSTOM_CONFIG {
    LONG    CounterStartValue;
    PEVT_OSR_CUSTOM_INFORM_COUNT    EvtOsrInformCountTens;
} OSR_CUSTOM_CONFIG, *POSR_CUSTOM_CONFIG;

VOID
FORCEINLINE
OSR_CUSTOM_THING_CONFIG_INIT(_Out_ POSR_CUSTOM_CONFIG Config,
    _In_    LONG CounterStart)
{
    RtlZeroMemory(Config, sizeof(OSR_CUSTOM_CONFIG));
    Config->CounterStartValue = CounterStart;
}
```

Figure 2 – Implementing OSRCUSTOM’s Configurator

(CONTINUED ON PAGE 9)
The NT Insider

WDF Objects... (Cont.)

```c
NTSTATUS OsrCustomCreate(_In_ WDFDEVICE Device,
_In_ POSR_CUSTOM_CONFIG Config,
_In_opt_ PWDF_OBJECT_ATTRIBUTES ObjectAttributes,
_Out_ POSRCUSTOM Object)
{
    WDFOBJECT object;
    NTSTATUS status;
    PVOID data;
    POSRCUSTOM_INTERNAL_DATA customData;
    WDF_OBJECT_ATTRIBUTES objAtts;

    // If no Object Attributes provided by the user, supply the default
    if (ObjectAttributes == WDF_NO_OBJECT_ATTRIBUTES) {
        WDF_OBJECT_ATTRIBUTES_INIT(&objAtts);
        ObjectAttributes = &objAtts;
    }

    // Override whatever the user specifies for Parent
    ObjectAttributes->ParentObject = Device;
    status = WdfObjectCreate(ObjectAttributes,&object);
    if (!NT_SUCCESS(status)) {
        DbgPrint("WdfObjectCreate for object failed 0x%0x\n", status);
        goto done;
    }

data = ExAllocatePoolWithTag(NonPagedPoolNx,sizeof(OSRCUSTOM_INTERNAL_DATA),'crso');
    if (data == nullptr) {
        DbgPrint("ExAllocatePoolWithTag for object data failed\n");
        status = STATUS_INSUFFICIENT_RESOURCES;
            goto done;
    }

customData = (POSRCUSTOM_INTERNAL_DATA)data;
    RtlZeroMemory(customData, sizeof(OSRCUSTOM_INTERNAL_DATA));
    customData->Config = Config->CounterStartValue;
    customData->EvtOsrCustomInformCount = Config->EvtOsrCustomInformCount;
    status = WdfObjectAddCustomTypeWithData(object,
            OSRCUSTOM,
            (ULONG_PTR)data,
            NULL,
            EvtWdfDestroyCustom);

    if (!NT_SUCCESS(status)) {
        DbgPrint("WdfObjectAddCustomTypeWithData for object failed 0x%0x\n", status);
        goto done;
    }

    DbgPrint("New thing created, with Counter Starting value = %u\n", customData->Counter);
    status = STATUS_SUCCESS;
    *Object = (OSRCUSTOM)object;
}

Figure 3 – Implementing OsrCustomCreate
```
After instantiating the native generic WDFOBJECT, we allocate a private data area that will be stored with our custom Object instance. This private data area (OSRCUSTOM_INTERNAL_DATA, shown in Figure 4) is like a WDF Object Context, but is not accessible by our user. We then initialize the private data area. Finally, we call WdfObjectAddCustomTypeWithData, providing a handle to the native Object type, the data type of my custom Object, a pointer to the private data area that we allocated from pool, and a pointer to an EvtDestroyCallback in which we simply return the custom data area we allocated from pool. You can see the EvtDestroyCallback we created for OSRCUSTOM in Figure 5 (next page).

And So It Goes

Now that you’ve provided all the stuff necessary to configure and instantiate your custom Object, all you need to do is provide some methods that operate on or using your Object. Our OSRCUSTOM Object provides two such methods:

- OsrCustomIncrementCount – Increments the Count on the Object
- OsrCustomGetCount – Retrieves the Count on the Object

The code for OsrCustomIncrementCount is shown in Figure 6 (next page).

There are three (slightly) interesting things to review in Figure 6. First, you can see that we retrieve a pointer to the private data area that’s stored with the instance of our Object using the function WdfObjectGetCustomTypeData. Next, note that we implement the counter increment operation using an interlocked increment operation. We do this because, according to the WDF rules, all WDF Objects are internally thread safe... no other locks are required to keep them sane. Finally, note that we invoke the user-provided EvtOsrCustomInformCount Event Processing Callback if the counter is incremented to a multiple of ten. We do this directly in-line here... you might choose to do this in a more sophisticated manner if required.

```c
typedef struct _OSRCUSTOM_INTERNAL_DATA {
    LONG Counter;
    PEVT_OSR_CUSTOM_INFORM_COUNT EvtOsrCustomInformCount;
} OSRCUSTOM_INTERNAL_DATA, *POSRCUSTOM_INTERNAL_DATA;
```

Figure 4 – Custom Internal Data

Why Engage with OSR?

We Know What We Know
And...We Know What we Don’t Know

We are not experts in everything. We’re not even experts in everything to do with Windows. But we think there are a few things that we do pretty darn well. We understand how the Windows OS works. We understand devices, drivers, and file systems on Windows. We’re pretty proud of what we know about the Windows storage subsystem.

What makes OSR unique is that we can explain these things to your team, provide you new insight, and if you’re undertaking a Windows system software project, help you understand the full range of your options. AND we also write kick-ass kernel-mode Windows code. Really. We do.

Why not fire-off an email and find out how OSR can help. If we can’t help you, we’ll tell you that, too.

Contact: sales@osr.com
VOID
EvtWdfDestroyCustom(  
    _In_  WDFOBJECT Object)
{
    PVOID data;
    POSRCUSTOM_INTERNAL_DATA internalData;
    data = (PVOID)wdfObjectGetCustomTypeData(Object, OSRCUSTOM);
    if (data == nullptr) {
        DbgPrint("wdfObjectGetCustomTypeData for object data failed\n");
        goto done;
    }
    internalData = (POSRCUSTOM_INTERNAL_DATA)data;
    DbgPrint("Count at free is %u\n", internalData->Counter);
    ExFreePool(data);
}
done:
    return;
}

Figure 5 – Destroy callback for OSRCUSTOM

NTSTATUS  
OsrCustomIncrementCount(OSRCUSTOM Object)
{
    NTSTATUS status;  
    POSRCUSTOM_INTERNAL_DATA customData;  
    customData = (POSRCUSTOM_INTERNAL_DATA)wdfObjectGetCustomTypeData(Object, OSRCUSTOM);
    if (customData == nullptr) {
        DbgPrint("wdfObjectGetCustomTypeData for object data failed\n");
        status = STATUS_INVALID_ADDRESS;
        goto done;
    }
    InterlockedIncrement(&customData->Counter);
    DbgPrint("Thing count is now %u\n", customData->Counter);
    // If the count has reached another boundary of 10, call the Event  
    // Processing Callback to let the user know.
    if ((customData->Counter % 10) == 0) {
        if (customData->EvtOsrCustomInformCount != NULL) {
            (customData->EvtOsrCustomInformCount)(Object);
        }
    }
    status = STATUS_SUCCESS;
}
done:
    return status;
}

Figure 6 – OsrCustomIncrementCount

(CONTINUED ON PAGE 12)
Finally, we have the OsrCustomGetCount property function, shown in Figure 7.

Again, it’s important to remember to follow the standard WDF rules when you implement these functions. Remember “Set” and “Get” functions can never fail, and thus do not return NTSTATUS. We follow that pattern here, returning the value of Count as the output from the function.

But Why?
So now you know how to create custom WDF Object types. Let’s talk a bit about how you can use these and why you might want to create one.

In its most basic form, custom WDF Object types provide a way to derive a unique Object type from a native base WDF Object. Because the Framework implements a C Language interface, we can’t do this through simple inheritance. However, by using the WDF custom Object type mechanism we can create new Objects that have their own, private data areas, their own methods, and even their own Event Processing Callbacks. And, the new Objects that we create will still have all the attributes of the underlying Object type on which we based them. No matter how you look at it, when you create a new WDF Object type, you’re creating a child Object of an existing, base, native WDF Object type.

```c
LONG OsrCustomGetCount(OSRCUSTOM Object) {
    NTSTATUS status;
    POSRCUSTOM_INTERNAL_DATA internalData;
    internalData = (POSRCUSTOM_INTERNAL_DATA)wdfObjectGetCustomTypeData(Object, OSRCUSTOM);
    // GET cannot fail... so...
    // ASSERT(internalData != nullptr);
    return internalData->Counter;
}
```

Figure 7 – Implementation of “Getter” for Count Property

OSR Custom Software Development
I Dunno...These Other Guys are Cheaper...Why Don’t We Use Them?

Why? We’ll tell you why. Because you can’t afford to hire an inexperienced consultant or contract programming house, that’s why. The money you think you’ll save in hiring inexpensive help by-the-hour will disappear once you realize this trial and error method of development has turned your time and materials project into a lengthy “mopping up” exercise...long after your “inexpensive” programming team is gone.

You deserve (and should demand) definitive expertise. You shouldn’t pay for inexperienced devs to attempt to develop your solution. What you need are fixed-price solutions with guaranteed results. Contact the OSR Sales team at sales@osr.com to discuss your next project.
I’d argue, however, that due to the complexity of the interface and the infrastructure that you need to create to make your Object work like an ordinary WDF Object, simply wanting to casually create your own flavor of WDFREQUEST for use in your custom driver probably is not a good way to use the WDF custom Object capability.

Rather, I’d suggest that the best use of this facility is to create common infrastructural Objects that your team can use across drivers. This might be to support specific device types or architectures, like Microsoft does with Framework Class Extensions. Or it might be to support specific attributes or mechanisms of processing for your device(s). Above all, for me the differentiating factor that determines whether I should create a custom WDF Object is if the specialized Object type that I want to create is generic and useful across multiple drivers. I can’t justify the work, and the required infrastructure, for any other reason.
We’ve finally started running our new and improved Developing File System Minifilters for Windows seminar! Of course, this means that I’ve spent a considerable amount of time musing about Windows file systems, file system filters, and the Filter Manager framework. While this isn’t exactly unusual, recently my thoughts have revolved around answering a deceivingly difficult question: how do we teach a developer to be a successful file system filter driver writer in 2019?

To answer this question, I decided that the beginning was the best place to start. Architecturally, what is a Windows file system filter anyway? Also, Filter Manager wasn’t released until the Windows Vista/XP SP2 timeframe. Why? Clearly folks were writing file system filters before its release, so why introduce a new model? What problems does it solve? And, more importantly, what problems does it not solve?

In Windows, we use a layered, packet based I/O model. Each I/O request is represented by a unique I/O Request Packet (IRP), which is sufficient to fully describe any I/O request in the system. I/O requests are initially presented to the top of a Device Stack, which is a set of attached Device Objects. The I/O requests then flow down the Device Stack, being passed from driver to driver until the I/O request is completed.

If the I/O request reaches the bottom of a Device Stack, the driver at the bottom may choose to pass the request on to the top of another Device Stack. For example, if an application attempts to read data from a file, the I/O request is initially presented to the top of the file system stack. If the request reaches the bottom of the file system stack, the I/O request may then be passed on to the top of the volume stack. At the bottom of the volume stack, the I/O request may then be passed on to the top of the disk stack. At the bottom of the disk stack, the I/O request may then be passed on to the top of the storage adapter stack.

Figure 1 is a representation of this processing if all these Device Stacks had a single Device Object.

An important thing to note is that the I/O requests are passed using a call through model. This means that, using Figure 1 as our example, NTFS passes the request along by calling directly into the Volume driver. The Volume driver then directly calls into the Disk driver. The Disk driver then passes the request by calling directly into the Storage Adapter driver. Depending on the number of Device Objects and number of Device Stacks involved, this can create significantly long call chains (as we’ll see in a moment).

An awesome feature of the I/O Manager in Windows is that each individual Device Stack may contain one or more filter Device Objects. By attaching a filter Device Object to a Device Stack, a filter driver writer may intercept I/O requests as they pass through the Device Stack. For example, a filter in the file system stack intercepts file level operations before (“pre-”) the file system driver has a chance to process them. Likewise, a filter in the volume stack sees volume level operations before the volume driver has a chance to process them.

Filter drivers are also given the opportunity to intercept operations after (“post-”) the lower drivers process the request. For example, a file system filter driver could intercept the data read from a file by opting to process the I/O request after the file system has read the file data.

Windows ships with many filter drivers in these and related stacks. Figure 2 shows a complete picture of the filters provided with a clean Windows 10 1703 installation.

The number of filter drivers present on a default installation of Windows amplifies the issue with the call through model. In order to get an I/O request to the storage adapter, we need to create a call chain that includes eight different drivers!

(CONTINUED ON PAGE 15)
FS Filter in 2019... (Cont.)

(CONTINUED FROM PAGE 14)

Boilerplate Legacy File System Filter
Given our discussion thus far, let’s talk about how someone would write a file system filter driver in this model. Along the way we’ll highlight some different challenges we might face just in our interactions with the system, let alone in whatever other value add processing we might try to provide (e.g. A/V scanning).

The first goal is we’re going to need to get a filter Device Object attached to the file system stack. Once our filter is attached, we’ll see all the I/O requests coming from the application before (“pre-”) the file system (Figure 3).

As mentioned previously, the file system filter may also see the operations after (“post-“) the file system’s processing.

There are several problems that quickly happen when you talk about putting a filter in the file system stack. We’ll highlight some of these here.

Filter Layering
There is only one file system filter in this figure. However, if there were multiple there is no architecturally defined order for how multiple filters in the stack are layered. The OSR Filter might be above another other filters in the file system stack, or it might be below the other filters. The best we can do is try to choose the right load order group for our filter and hope that works well enough.

Dynamic Loading and Unloading
Filter drivers can only be dynamically attached to the top of the stack. For example, I could not take a running system and insert another filter driver between the OSR Filter and NTFS. Additionally, filter drivers cannot safely, dynamically detach from an active Device Stack.

The only way to properly layer the filter or detach the filter (and thus unload your driver) is by destroying the Device Stack and rebuilding it with the filter in place. In the case of your system volume, this means instantiating anywhere but at the top of the device stack or unloading your filter driver requires a reboot.

Mechanics of Attachment and Detachment
HOW the file system stack instantiates itself and tears itself down is a deep rat hole filled with weird edge cases. Just managing the attachment and detachment of the filter to the stack requires a significant amount of arcane knowledge that no one should need to know (including special case handling for when you eject a floppy diskette and then re-insert it!).

(CONTINUED ON PAGE 16)
FS Filter in 2019... (Cont.)

(CONTINUED FROM PAGE 15)

**Propagating I/O Request Packets (IRPs)**
Once a filter Device Object is attached to the Device Stack, it is responsible for processing all I/O Request Packets sent to the Device Stack. Even if a filter driver is only interested in processing read requests, it still must intercept all requests supported by the lower drivers in the Device Stack.

**Trouble with Fast I/O Data Operations**
As mentioned earlier, Windows uses a packet based I/O model. For example, each call to `ReadFile` in user mode creates a new, unique I/O Request Packet (IRP) to represent the I/O operation. The IRP is then passed around from driver to driver until the request is completed.

Long ago, someone made an interesting observation: lots of IRPs only make it as far as the file system before they are completed. This is particularly true in the case of cached file I/O. The file system simply copies the data out of the file system cache and returns it directly to the user. In these cases, all the work to build the IRP and pass it along was wasted.

Thus, the idea of “Fast I/O” was born. The file systems provide something called a Fast I/O Dispatch Table, which contains entry points to, for example, read data from a cached file. Instead of building the IRP, the I/O Manager bypasses the Device Stack entirely and calls directly into the file system to retrieve the data.

Now for the fun part: the I/O Manager always calls the Fast I/O Dispatch Table located at the top of the Device Stack. Once a filter driver is attached at the top, the filter driver receives the Fast I/O requests. If the filter driver does not register a Fast I/O Dispatch Table, then Fast I/O processing is disabled for the Device Stack and we lose the benefit.

Thus, it is the responsibility of the filter driver to “pass” Fast I/O requests down the stack by calling the lower driver’s Fast I/O Dispatch Table. There is no support for this processing provided by the system, thus every driver must invent and provide this code to not lose functionality in the system.

**Trouble with Fast I/O Non-Data Operations**
You would be foolish to think that the Fast I/O Dispatch Table only contains callbacks related to I/O!

To maintain a consistent locking hierarchy between the file system, Cache Manager, and Memory Manager, there are also some callbacks in the Fast I/O Dispatch Table related to acquiring locks in the file system. For example, the Memory Manager call’s the file system’s `AcquireFileForNtCreateSection` callback when a user attempts to memory map a file.

Unlike the Fast I/O callbacks related to data operations, the locking related callbacks are not sent to the top of the Device Stack. They are always sent directly to the file system.

(CONTINUED ON PAGE 17)
FS Filter in 2019... (Cont.)

(CONTINUED FROM PAGE 16)

Sounds great! Finally something we don’t need to deal with... However, a file system filter might actually be interested in these callbacks. There are many cases when a filter might want to know that an application is memory mapping a file. If the callback bypasses the file system filter, how do we get that notification?

On XP and later a mechanism was added: FsRtlRegisterFileSystemFilterCallbacks. This allows the filter to receive notification (and even fail) locking requests related to various Cache and Memory Manager activities. Note that these callbacks do not need to propagate execution to lower filter drivers, the operating system takes care of the layering for us. They also take an orthogonal set of arguments from the standard IRP and Fast I/O based callbacks.

Trouble with Recursion

File system operation is recursive by nature. For example, imagine an application sends a cached read request to the file system. This I/O request will flow down the file system Device Stack (as discussed) before reaching the file system.

The file system will then ask the Cache Manager to please copy the data from the cache into the user data buffer. But, what if the data is not in the cache? The Cache Manager must then perform a non-cached read of the file data to retrieve the data from disk. Of course, it does this by submitting a new read request to the top of the file system stack. Thus, a file system filter may pass a read request down the stack and, before it returns, may see another read request arriving from the same thread (Figure 4).

While recursive operation is a way of life in the file system stack, file system filters can amplify the problems of recursive processing.

First, remember that this is all done using a call through model. Thus the call stack here can be enormous. It was not at all uncommon to see stack overflow bugchecks when there were multiple legacy file system filters present.

Figure 4—Recursion Amp’ed Up in the Minifilter World

(CONTINUED ON PAGE 18)
Secondly, if a file system filter generates its own recursive I/O then the filter must be careful to not deadlock itself (or other filters). For example, imagine an anti-virus filter that does not want to allow an application to open the file before it has been scanned. If the file system filter generates recursive I/O operations to read the file, the A/V scanning filter will see its own read I/O requests arrive at the top of the device stack. It must be sure to allow those reads to occur, even though the file scan has not yet completed.

A Lot of Work to do Nothing
To give you a sense of scope, old versions of the Windows Driver Kit shipped with a “Simple” legacy file system filter driver. This filter did absolutely nothing except attach a filter Device Object, pass all IRP and Fast I/O requests without modification, and register for the file system filter callbacks. The total line count for this “do nothing” filter? 6,800!

Now Let’s Get to Work!
Now that we have almost 7,000 lines of boilerplate code, we can get to work! There are lots common things that almost every file system filter driver needs to do.

Contexts, Contexts, Contexts
One of the things that we will undoubtedly want to do is attach context to various different things. For example, we might want to create a context for the underlying file system and volume that we’re filtering. Are we filtering FAT? NTFS? Is the drive removable or fixed? Or are we filtering the network?

We might also want context associated with the files we are filtering. For example, in our write request handler how do we know which file is being written?

We might also want context associated with the individual streams of a file. For example, in our write request handler how do we know which stream of the file is being written?

We might also want context associated with the individual opens of a stream. For example, in our write request handler how do we which open of the stream was used to write the file?

Retrieving File Names
Of course, if you’re in a file system filter driver then you probably want to know the name of the file being read, written, renamed, deleted, etc.

Unfortunately, something that is seemingly so obviously necessary is a very involved and complicated task. There are unwritten rules on when you can and cannot query the names of a file for fear of a deadlock. The overhead of this activity can also be significant, especially when querying name information over the network.

Also, in the legacy filter model every filter that wants a name must perform its own name query. Thus, we have a very involved and complicated task being performed by every file system filter in the stack.

Communicating with User Mode
It is not uncommon for a file system filter to work in conjunction with a user mode service. The user mode service might be used simply to send control or configuration information to the driver, or to consume logging activity generated by the driver. In many cases, the user mode service can be used to offload complex processing that is much better suited to user mode development (e.g. binary analysis prior to execution).

A Ton of Work to do a Little
Because I love counting lines of code, we’ll look to the old FileSpy sample to get a sense of scope in the leap from “do nothing” to “do the common things you probably need.” The FileSpy sample creates file system and volume context, as well as per-stream contexts. It retrieved file names for individual file operations, and logs file system activity to a user mode console application. The grand total for the filter only (i.e. not including user mode components): 16,200!
The NT Insider  
Sep - Oct 2019

FS Filter in 2019... (Cont.)

(CONTINUED FROM PAGE 18)

Clearly Needs Fixed...
So, there you have the state of file system filter development in the year 2000. The highlights include:

- No ability to dynamically load and unload the filter
- Non-deterministic behavior in terms of the layering of filters
- The simple act of attaching the Device Object to the stack is needlessly complicated
- Increased execution stack utilization for each and every filter added
- Filters intercept requests in three different ways: IRPs, Fast I/O, and File System Filter Callbacks
- When there are multiple filters loaded, there is a significant amount of duplicate work, especially around querying file names
- Filters that generate recursive I/O requests need to perform extra work to avoid blocking their own execution.
- Filters must also be prepared to handle recursive I/O from lower filters
- Filters that need to communicate with user mode must create all their own communication code
- A filter that does nothing takes 6,000 lines of code
- A filter that kind of does something takes 16,000 lines of code

This is all in addition to the fact that the Windows file system I/O interface is inherently complicated. Much of this is due to the long history of the interface and the many edge conditions that have been created over the years. As mentioned, there is a significant amount of recursive activity generated by the file system. During these recursive I/O operations, locks may be held by the lower file system, thus making it dangerous to attempt incompatible operations. The file systems are also deeply integrated with the Cache and Memory Manager subsystems in Windows, which have their own rules and own locking requirements.

Lastly, each Windows file system is different than the other by way of implementation. Thus, the behavior that you see while interacting with a particular file system may be architecturally similar but different in its observed behavior.

At this point, a group of very senior file system and kernel architects and developers at Microsoft (i.e. “very smart people”) decided this needed fixed. Not because they had nothing better to do, but because a large percentage of Windows crashes were being blamed on file system filter drivers. While you could try to blame the filter developers for this, the filtering model was objectively broken. File system filter drivers were clearly an afterthought in the original design of NT. It was time to sit down and architect and design a way to write a file system filter driver on Windows.

Enter Filter Manager!
And this is where Filter Manager and the Filter Manager Framework come in. The team behind Filter Manager wanted to make it easier and more reliable to write a file system filter driver on Windows. In doing so, there were two significant requirements that had to be met:

1) The “new” way to write a file system filter must be as flexible as the existing model. If there was a filter type that could be written the old way, but not in the new way, then the new way was a failure
2) The underlying file system architecture in Windows could not be radically changed. In other words, the goal of simplifying file system filter development could not be met by simplifying the file system interface

Thus, the ultimate solution to the problem was to create a framework for writing file system filter drivers. The framework provides the one
FS Filter in 2019... (Cont.)

(CONTINUED FROM PAGE 19)

legacy file system filter driver necessary in the system, and consumers of the framework plug in as “minifilters”. As I/O requests arrive at the Filter Manager legacy filter Device Object, Filter Manager calls the minifilters using a call out model. After each minifilter processes the request, Filter Manager then calls through to the next Device Object in the Device Stack (Figure 5).

However, Filter Manager does not simply pass the native operating system operations to the minifilters. Instead, Filter Manager creates its own abstractions of the native operations (we’ll see the benefit of this shortly).

What Filter Manager Does...

Filter Manager is great and does a lot of things for you. This includes but is not limited to the following.

Allow for Dynamic Load and Unload

Filter Manager itself is a legacy file system filter driver, thus it cannot dynamically load and unload. However, it provides support so that minifilters can dynamically load and unload (if they choose).

Deterministic Layering Behavior

Filter Manager introduces the concept of altitudes, where higher altitude filters are called before lower altitude filters for operations en route to the file system. Altitudes are groups by functionality (e.g. AntiVirus, Encryption, etc.) and Filter Manager ensures that the filters are called in the correct order based on their altitudes.

Simplifies Filter Attachment

Filter Manager still needs to deal with all those annoying details of how the file system stack instantiates and tears down. However, it does not expose the minifilter driver writer to all this non-sense. Instead, Filter Manager creates an abstraction called an Instance. An Instance is an instantiation of a minifilter within a file system stack (e.g., in Figure 5 there are three Instances). Instance setup and teardown follows a sane set of rules and requires minimal code (and no special case floppy code!).

As an added bonus, a single filter is even allowed to have multiple instances within the same stack! This has some useful applications, including breaking a complex filter up into multiple different instances. For example, in our File Encryption Solution Framework (FESF) we use separate instances for data transformation (i.e. encryption) and managing the on disk structure of our encrypted files.

Decreases Stack Utilization

Filter Manager’s use of a call out model to the minifilters significantly decreases the stack utilization of I/O operations. It also means that adding more filters does not instantly increase the stack utilization of every single I/O request.

Provides a Coherent Context Model

As mentioned previously, file system filter drivers undoubtedly need to attach context to things such as volumes, files, streams, and even File Objects (i.e., open instances of files or streams). Filter Manager provides a consistent context API for attaching and retrieving context for these objects. Filter Manager also provides additional support for attaching context to instances, transactions, and memory mappings.

Provides a Coherent Callback Model

Legacy file system filters needed to deal with IRPs, Fast I/O, and FsRtl filter callbacks to capture all possible file system operations. Filter Manager rationalizes all these different callbacks into a single, unified callback using a Callback Data structure. This Callback Data structure provides a consistent view of file system operations and is the one structure that minifilters need to understand to filter any file system operation.

(CONTINUED ON PAGE 21)
FS Filter in 2019... (Cont.)

(CONTINUED FROM PAGE 20)

**Provides Name Query Support (with Cross-Instance Caching)**

Filter Manager provides a consistent interface for querying file names for any given file system operation. In addition, Filter Manager understands all the situations in which it is not safe for a file system filter to query name information. So, instead of a difficult to diagnose deadlock in your filter, Filter Manager simply returns an error if it is currently unsafe for you to query name information.

Even better, Filter Manager caches the results of name query operations and shares the results amongst minifilter instances. Thus, if one filter queries the name for a given I/O request, other filters can benefit from the cached result.

**Provides Support for Avoiding Recursive I/O**

Filter Manager allows minifilters to target I/O requests at specific altitudes. Thus, a minifilter can choose to send an I/O request only to those minifilters that are at lower altitudes. This means that a minifilter does not need to write defensive code to detect its own recursive operations.

(CONTINUED ON PAGE 22)

---

Windows or Linux File Encryption? Let Us Help!

OSR’s File Encryption Solution Framework (FESF)

We know that implementing per-file encryption solutions is challenging—and we’ve been doing it for more than 20 years!

Of course, it’s also a challenge to convince devs and their managers of the difficulties they face. If you’re new to Windows, you might be looking at the WDK examples and think “This looks easy!” If you’re targeting Linux you might think “Hey, I can always start with one from an open source package.”

Then you can spend months — MANY months — getting something to work. We run into this almost daily. Devs who are “one bug away” from getting their solution to work. But that “one bug” turns into “one more bug” and this goes on... for months.

Please don’t let this be you. OSR can help save you a great deal of time and money and help make your Windows file encryption solution successful. OSR’s latest toolkit—the File Encryption Solution Framework (FESF) builds on a time-tested infrastructure, but moves all the core development for a file encryption solution to USER mode. You don’t need to be an expert in Windows file system or kernel programming, and your time is better spent defining “policy” of your solution (what you want to encrypt, when, and with what algorithm and key management) instead of wrestling with the subtle and painful nuances in filtering file systems.

Want to try a fully-functional evaluation of FESF for FREE? Just contact the OSR sales team and they’ll get you started.

Contact: [sales@osr.com](mailto:sales@osr.com)

Free FESF Eval Available Now!
Provides Support for User/Kernel Communication
Given that the majority of file system filter drivers communicate with user mode, Filter Manager provides a nice Communication Port package for bi-directional communication. User mode applications can easily send messages to the minifilter, and the minifilter can easily send messages to user mode (with or without a response).

Significantly Decreases the Size and Complexity of Boilerplate Code
Clearly a major goal in the creation of Filter Manager was to reduce the amount of boilerplate code in a filter. If every filter needs complex attachment code, then Filter Manager should handle that. If every filter is going to perform name queries, then Filter Manager should handle that. If every filter is going to communicate with user mode, then Filter Manager should handle that.

What Filter Manager Does Not...
Filter Manager is so well considered and so well executed, that it almost makes writing filters easy! A simple sample is only ~900 lines of code (mostly comments) and there are lots of examples provided on GitHub. It’s all very approachable and, in fact, quite easy to get something demonstrably working in a short period of time.

Even with all that, writing filter drivers is still hard in 2019. You can see this from the level of questions that show up on NTFSD. Everyone thinks that their filter is pretty much working, but there’s one or two little problems to solve. Unfortunately, those problems are things like their filter doesn’t work with Notepad. Or the system hangs when they try to save an Excel document. We see it all the time, folks think they’re just a few weeks away from having a product and they go right off a cliff. Unfortunately, those one or two issues could easily take months to resolve.
FS Filter in 2019... (Cont.)

So, what’s happening? Are the samples not good enough? Is the documentation not good enough? Or is it that Filter Manager failed in its goal to make file system filter drivers easier to write and maintain?

The answer to this question goes back to fundamental requirement #2 in the development of Filter Manager:

\[ \text{The underlying file system architecture in Windows could not be radically changed. In other words, the goal of simplifying file system filter development could not be met by simplifying the file system interface} \]

Thus, the biggest hurdle to learning to write a file system filter in 2019 is the same hurdle you faced in 1999: understanding the complexity of the underlying file system interface. Sure, the mechanics of how you assemble a filter are important, but the devil is in the details of the underlying interface. This understanding greatly influenced the development of our minifilter seminar, where we spend lots of time talking about the why of the Filter Manager callbacks instead of just focusing on the how.

We’ve worked really hard to create a seminar with the right information, hopefully you’ll get a chance to join us and find out!

---

Windows Internals & Software Drivers
For SW Engineers, Security Researchers & Threat Analysts

Read what your colleagues have to say about this seminar. We can’t put it any better!

\[ \text{Scott showed a strong mastery of Windows internals and an incredible knowledge base that he was capable of passing on to us.} \]

\[ \text{Scott’s knowledge in everything was incredible. Not only was he able to answer every question students had, but we also knew the complete history of why things were designed a certain way and how they worked!} \]

\[ \text{Mr. Noone was definitely the best instructor I have had in my 20 years of being a programmer. He has a real knack for explaining things and added immense value to the slides and demos} \]

\[ \text{“The instructor is extremely knowledgeable regarding Windows internals. He has the communications skills to provide an informative, in-depth seminar with just the right amount of entertainment value.”} \]

Next Presentation:

Dulles/Sterling, VA  18-22 November

- Feedback from attendees of THIS seminar
There’s a New Pool Allocator In Town
LFH Kernel Pool Allocator Challenges the Incumbent

Windows 10 RS5 introduced a new pool allocator in the kernel for the first time in, well, forever. Interestingly, the newly introduced kernel pool allocator is actually the existing user mode Low Fragmentation Heap (LFH) allocator. While there’s something to be said for the, “if it ain’t broke” mentality, having a single allocator in the O/S certainly makes a certain amount of sense from a maintainability perspective. Also, the user mode heap allocator has undergone significant revisions over the years to better reflect modern security practices, so it makes sense to share those benefits with kernel mode as well.

Most of us wouldn’t even notice or care that there’s a new pool allocator (except for the fact that it broke `!pool`, that is). However, over the years I have debugged so many BAD_POOL_HEADER bugchecks that I was curious about how the new pool allocator responded to some obvious driver bugs. Specifically, I wondered about the following cases:

1) Buffer Overruns
2) Double Frees
3) Use After Frees

So, I seized the unique opportunity to intentionally write buggy code (and, yes, I did at one point end up with a bug in my buggy code that caused it to not be buggy). The buggy code provides IOCTLs to generate each buggy scenario and the code to handle the IOCTLs is shown in Figure 1.

I then ran each test to pit the Windows 7 allocator against the Windows 10 19H1 allocator to see which one performed better in detecting the bugs. Note that this was not a rigorous, scientific study involving thousands of iterations. Each one was run about three times max to validate that the behavior was at least somewhat repeatable.

```c
size = 268;
allocation = ExAllocatePoolWithTag(NonPagedPool,
size,
'KLS0');
DbgPrint("!pool 0x%p (size - 0x%x)\n", allocation, size);
switch (IoControlCode) {
    case IOCTL_OSRLK_OVERRUN: {
        DbgPrint("Zeroing 0x%x\n", size * 2);
        RtlZeroMemory(allocation, size * 2);
        ExFreePool(allocation);
        break;
    }
    case IOCTL_OSRLK_DOUBLE_FREE: {
        DbgPrint("Freeing twice\n");
        ExFreePool(allocation);
        ExFreePool(allocation);
        break;
    }
    case IOCTL_OSRLK_USE_AFTER_FREE: {
        DbgPrint("Freeing then zeroing\n");
        ExFreePool(allocation);
        RtlZeroMemory(allocation, size);
        break;
    }
}
```

Figure 1—Let’s Write Some Buggy Software to Test!
Pool Allocator... (Cont.)

(CONTINUED FROM PAGE 24)

Now, without further ado, the results...

Overrun Challenge

Windows 7

On Windows 7 the system immediately crashed with a BAD_POOL_HEADER:

BAD_POOL_HEADER (19)
The pool is already corrupt at the time of the current request.
This may or may not be due to the caller.
The internal pool links must be walked to figure out a possible cause of
the problem, and then special pool applied to the suspect tags or the driver
verifier to a suspect driver.
Arguments:
Arg1: 0000000000000020, a pool block header size is corrupt.
Arg2: ffffa801a26bde0, The pool entry we were looking for within the page.
Arg3: ffffa801a26bf00, The next pool entry.
Arg4: 0000000000412003a, (reserved)

Running !pool on second argument to the bugcheck walks the pool page and shows us where we went off a cliff:

1: kd> !pool @$bug_param2
Pool page ffffa801a26bde0 region is Nonpaged pool
 ffffa801a26b000 size:  sc0 previous size:  0 (Allocated)   Txrn
 ffffa801a26b5c0 size:  1b0 previous size:  sc0 (Free)       Free
 ffffa801a26b770 size:  c0 previous size:  1b0 (Allocated) FMsI
 ffffa801a26b830 size:  150 previous size:  c0 (Allocated) File (Protected)
 ffffa801a26b980 size:  c0 previous size:  150 (Allocated) FMsI
 ffffa801a26ba40 size:  3a0 previous size:  c0 (Free)       FMic
 *fffffa801a26bde0 size:  120 previous size:  3a0 (Free ) *OSLK
Owning component : Unknown (update pooltag.txt)

fffffa801a26bf00 doesn't look like a valid small pool allocation, checking to see
if the entire page is actually part of a large page allocation...

Windows 10 19H1

Running the same test on Windows 10 produced no crash. Running !pool on the freed buffer shows a corruption of the page just
like on Windows 7:

1: kd> !pool 0xFFFFCD02FB902050
Pool page ffffcd02fb902050 region is Nonpaged pool
 ffffc02fb902000 size:   30 previous size:  0 (Free) ........

ffffcd02fb902040 doesn't look like a valid small pool allocation, checking to see
if the entire page is actually part of a large page allocation...

But no BAD_POOL_HEADER crash.

(CONTINUED ON PAGE 26)
Interestingly, I ran the test again and this time I did hit a crash. However, it was an IRQL_NOT_LESS_THAN_OR_EQUAL bugcheck in the bowels of the heap allocator on the next allocation:

IRQL_NOT_LESS_OR_EQUAL (a)
An attempt was made to access a pageable (or completely invalid) address at an interrupt request level (IRQL) that is too high. This is usually caused by drivers using improper addresses.
If a kernel debugger is available get the stack backtrace.
Arguments:
Arg1: ffffcd02fb8b5022, memory referenced
Arg2: 0000000000000002, IRQL
Arg3: 0000000000000000, bitfield:
  bit 0: value 0 = read operation, 1 = write operation
  bit 3: value 0 = not an execute operation, 1 = execute operation (only on chips which support this level of status)
Arg4: fffff80574452c49, address which referenced memory

Overrun Challenge Winner: Windows 7. Corruption was detected immediately when the buffer was freed and we were provided a clear bugcheck description.

Double Free Challenge
Windows 7

On Windows 7 the system immediately crashed with a BAD_POOL_CALLER:

BAD_POOL_CALLER (c2)
The current thread is making a bad pool request. Typically this is at a bad IRQL level or double freeing the same allocation, etc.
Arguments:
Arg1: 0000000000000007, Attempt to free pool which was already freed
Arg2: 00000000000001097, Pool tag value from the pool header
Arg3: 00000000004120009, Contents of the first 4 bytes of the pool header
Arg4: fffff805b3de9f0, Address of the block of pool being deallocated

Windows 10 19H1
Running the same test on Windows 10 produced no crash. Much like last time, running the test a second time did indeed result in a system crash, though this time it was properly at the point of the second free:

KERNEL_MODE_HEAP_CORRUPTION (13a)
The kernel mode heap manager has detected corruption in a heap.
Arguments:
Arg1: 00000000000000011, Type of corruption detected
Arg2: fffff91030000100, Address of the heap that reported the corruption
Arg3: fffff9103dd133a0, Address at which the corruption was detected
Arg4: 0000000000000000
Curious about what \texttt{Arg1 == 0x11} meant, I took a SWAG and grep’d the type information for something related to “heap” and “type”:

\begin{verbatim}
0: kd> dt nt\_heap\_type
  ntkrnlmp\_HEAP\_SEG\_RANGE\_TYPE
  ntkrnlmp\_HEAP\_FAILURE\_TYPE
\end{verbatim}

That was lucky! Dumping \texttt{HEAP\_FAILURE\_TYPE} we see that 0x11 (0n17) maps to \texttt{heap\_failure\_segment\_lfh\_double\_free}:

\begin{verbatim}
0: kd> dt nt\_HEAP\_FAILURE\_TYPE
  heap\_failure\_internal = 0n0
  heap\_failure\_unknown = 0n1
  heap\_failure\_generic = 0n2
  heap\_failure\_entry\_corruption = 0n3
  heap\_failure\_multiple\_entries\_corruption = 0n4
  heap\_failure\_virtual\_block\_corruption = 0n5
  heap\_failure\_buffer\_overrun = 0n6
  heap\_failure\_buffer\_underrun = 0n7
  heap\_failure\_block\_not\_busy = 0n8
  heap\_failure\_invalid\_argument = 0n9
  heap\_failure\_invalid\_allocation\_type = 0n10
  heap\_failure\_usage\_after\_free = 0n11
  heap\_failure\_cross\_heap\_operation = 0n12
  heap\_failure\_freelists\_corruption = 0n13
  heap\_failure\_listentry\_corruption = 0n14
  heap\_failure\_lfh\_bitmap\_mismatch = 0n15
  heap\_failure\_segment\_lfh\_bitmap\_corruption = 0n16
  heap\_failure\_segment\_lfh\_double\_free = 0n17
  heap\_failure\_vs\_subsegment\_corruption = 0n18
  heap\_failure\_null\_heap = 0n19
  heap\_failure\_allocation\_limit = 0n20
  heap\_failure\_commit\_limit = 0n21
\end{verbatim}

So there is double free detection, but for some reason it didn’t trigger on the first pass.

\textbf{Double Free Challenge Winner:} Very close, but Windows 7 because it was caught the first time every time we tested. Windows 10 also lost points because the debugger didn’t provide additional reason information and we only came to it through a lucky guess.

\textbf{Use After Free Challenge}

\textit{Windows 7}

Running this test on Windows 7 produced \textit{no crash}.

\textit{Windows 10 19H1}

Running this test on Windows 10 produced \textit{no crash}.

\textbf{Use After Free Challenge Winner:} Tie. To be fair, it would take a lot of extra processing in the allocator to find this bug, so not surprising that the bug was not caught by either allocator.

\textbf{Overall Results}

Though there are surely benefits to the new allocator, in our opinion the old allocator wins in its ability to detection corruptions of the pool in cases tested.

\textbf{What About With Driver Verifier?}

Of course, the best way to find pool corruptions is with Driver Verifier and Special Pool. I’m happy to report that both allocators caught all three bugs equally, so no loss in functionality there.
Scanning File Data
FsRtlCreateSectionForDataScan (And Flt Variant) Explained

It’s a common requirement for a file system filter driver to scan file data as part of its normal operation. For example, a file system filter may not want to allow a user to open a file until it has a chance to calculate its MD5 hash. The filter can achieve this by registering an IRP_MJ_CREATE handler and allowing or denying the request to access the file based on the file’s hash value.

The first interesting problem we have is if we want to filter the IRP_MJ_CREATE operation before the file system (PreCreate) or after the file system (PostCreate). If the filter chooses to monitor the operation in PreCreate, the file is not yet opened and thus the file data cannot be retrieved. The filter must perform its own open (e.g. by using FltCreateFileEx2) and scan the file data with the resulting file object.

For this reason, it is almost always a better choice to scan the file in the PostCreate callback. Instead of performing yet another open request, we can simply hijack the user’s open request and use their file object to perform the data scan. In order to guarantee that we have the necessary access to the file, we can delay this processing until we see an open requesting data access. We can also cache our scan result within a Stream Context structure that we can quickly retrieve on subsequent attempts to open the file.

OK, now that we have a File Object we can read from, we have three choices for how to read the file data:

1) Non-Cached I/O
2) Cached I/O
3) Memory Mapped I/O

Let’s look at each of these in turn.

Non-Cached I/O
With a File Object in hand, we can call FltReadFile and generate non-cached I/O to the file. This reads the data directly from disk and must be done in sector aligned chunks.

Reading the file in this way is fine and works, but it has a terrible downside: the resulting read data is not cached (duh). If the user application then wants to read the file, we need to re-read the entire file. If we’re going to make the user wait to open the file until we’ve read some (or all) of the file, then the least we can do is not make her fetch it from disk again.

(CONTINUED ON PAGE 29)
Cached I/O

With a File Object in hand, we can call FltReadFile and choose to generate cached I/O instead. This reads the data from disk into the file system cache, then the data is copied from the cache into our supplied data buffer. The data can be read in arbitrary sizes and alignments.

Reading the file in this way is fine and works. It even has the benefit of caching the data that was read, which means that if the user reads the file using cached I/O, the data will (likely) still be in memory. If the user does non-cached I/O, then oh well we lose and need to read the file twice anyway.

We still have a couple of subtle downsides. First of all, what if the user doesn’t read the file data? Then we’ve just put a bunch of stuff in the cache that the user doesn’t care about, potentially evicting things that he does care about. Alternatively, what if the user then proceeds to memory map the file? The memory mapping will be backed by the same pages as the cache, thus you’ll get the benefit of the cached pages. However, we would still have a bunch of unnecessary Cache Manager structures and state floating around.

Memory Mapped I/O

Hopefully you can tell at this point that we’re leading you to a better solution: memory mapped I/O. With memory mapped I/O, the pages that we fault in while scanning the file data are cached in memory. If the user does non-cached I/O, then we still lose and need to read the file twice. However, if the user performs cached I/O, the pages will be used to satisfy the user’s cached I/O requests. Even better, if the user memory maps the file, the pages can be used to back that as well.

Thus, given that we can’t pre-determine how the user application is going to access the file, using memory mapped I/O is the most flexible approach. Memory mapping is also nice because accessing file content as pointer and length is more natural when trying to do things such as calculate hashes or interpret file content (e.g. parse headers).

In fact, scanning a file using a memory mapping is such a beneficial way to achieve this goal that there is an API designed for just this purpose: FsRtlCreateSectionForDataScan.

FsRtlCreateSectionForDataScan

For reference, here is the FsRtlCreateSectionForDataScan function prototype:

```c
NTSTATUS FsRtlCreateSectionForDataScan(
    _Out_     PHANDLE            SectionHandle,
    _Out_     PVOID              *SectionObject,
    _Out_opt_ PLARGE_INTEGER     SectionFileSize,
    _In_      PFILE_OBJECT       FileObject,
    _In_      ACCESS_MASK        DesiredAccess,
    _In_opt_  POBJECT_ATTRIBUTES ObjectAttributes,
    _In_opt_  PLARGE_INTEGER     MaximumSize,
    _In_      ULONG              SectionPageProtection,
    _In_      ULONG              AllocationAttributes,
    _In_      ULONG              Flags
);
```

The most important bits are that this API takes a pointer to a File Object and returns a Section (i.e. File Mapping) handle. The handle can either be created as a user mode handle or as a kernel mode handle, depending on the values set in the ObjectAttributes argument. The Section handle can then be passed to ZwMapViewOfSection in kernel mode or, even better, the MapViewOfFile API in user mode.

(Continued from page 28)
The fact that this API allows for the creation of a user mode handle makes the API incredibly flexible. For example, it is often desirable to offload the work of calculating hashes or parsing file structures to a user mode service. The only important note is that `FsRtlCreateSectionForDataScan` creates user mode handles in the *current user process*. Thus, if you call this API in `PostCreate` the user mode handle will be created in the process attempting to open the file, not your user mode service. Extra work must be done to make the call to `FsRtlCreateSectionForDataScan` in the target process of interest. For example, the filter may create this section in response to a message received via a Filter Manager Completion Port.

There are a couple of other interesting things to know about this API. First of all, this API will *not* allow the filter to map the file for *execute* access. This means that for executables, the data read from the file represents the structure of the file on disk, not the executable version in memory.

The second issue is a bit more subtle. In our theoretical example, we have taken a File Object from `PostCreate` and called `FsRtlCreateSectionForDataScan` on it. This File Object is from a user open request, which was directed to the top of the file system stack. When our file system filter then attempts to memory map the file, the request to memory map *must go to the top of the file system stack*. For example, when you call this API you should expect your own filter (and those above you) to be called at `IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION`. Also, any paging I/O requests triggered by accessing a mapped view of the section will arrive at the top of the file system stack.

This is a bit unusual for Filter Manager mini-filter drivers because we’re generally used to only sending I/O requests down the filter stack to avoid recursion. However, in this case we are hijacking the user’s File Object, thus our operations must look like operations that would have come from the user application.

*But wait!* There’s a `FltCreateSectionForDataScan`! I know that using Flt APIs prevent recursion and result in I/O requests going down the filter stack, so that must prevent the recursive paging I/Os, right??

That would be a great guess, but, nope, not the case at all...

**Then What Good is FltCreateSectionForDataScan?**
To quote *Jurassic Park*, “hold on to your butts”, because the Flt version of this API is a bit insane...

Once we have an active mapping to a section, there are a bunch of failure cases that can be triggered in the file system. Most importantly, attempts to purge file data from memory can fail due to the active mapping. This can result in cache coherency issues that wouldn’t have otherwise occurred. For example, if a user performed a non-cached write on a cached file, the file system will attempt to flush and purge the file data to reconcile the in memory data with the on disk data before allowing the non-cached write. Normally this is a best effort activity as Windows does not guarantee coherency in this case. However, if a filter is creating data scan sections then the scenario might become more likely.

(Continued on page 31)
This is the problem that FltCreateSectionForDataScan attempts to solve. As part of setting up the section, Filter Manager sends the file system an FSCTL_SET_PURGE_FAILURE_MODE request. This indicates to the file system that there is an active section for data scan on the specified file. If a conflicting operation occurs on the file while the section is valid, the file system reports the error to Filter Manager by failing the I/O request with STATUS_PURGE_FAILED. Filter Manager in turn calls the offending mini-filter(s) at their SectionConflictNotification callbacks to tear down their sections and allow the file system to retry the file operation.

Every filter needs to decide how to deal with this situation. A scan is in progress to determine if Process A should be allowed access to the file. At the same time, Process B (who presumably was already granted access) performs an operation that is incompatible with the section. A reasonable action in this case might be for the filter to deny Process A’s attempt to access the file and simply retry the scan on a subsequent access.

When the filter is done with its processing, it must call FltCloseSectionForDataScan to indicate to Filter Manager that the scan is complete. Filter Manager then notifies the file system that it may return to its default purge failure processing.

One final note: FltCreateSectionForDataScan was not introduced until Windows 8. Thus, if your filter needs to support Windows 7 and later, the recommendation would be to fall back to the FsRtl version on Windows 7 and use the Flt version where available (see FltGetRoutineAddress).

Kernel Debugging & Crash Analysis

100% of students taking this seminar say they would recommend it to their colleagues/friends!

You probably don’t need somebody to tell you how to set up WinDbg. But most people could probably use a few hints when it comes to actually using WinDbg to find the root cause of a system failure. Surprise! It turns out that debugging and crash analysis is a skill that can actually be taught. And learning that skill is what you’ll focus on in this seminar.

The instructor exhibited a very comprehensive knowledge of the material, added with an incredible ease in explaining a very complex subject. I highly recommend this course.

This is the best seminar I attended. I learn more in a week than in a year in College!

- Feedback from two different attendees of THIS seminar


Next presentation: Manchester, NH (OSR)
30 March—3 April 2020
Even though they may feel spontaneous, a system crash is always an explicit decision by a kernel component to bring the machine down. Sometimes the machine is in such a bad state that immediately resetting the entire system is the only reasonable thing to do. For example, if the Memory Manager detects a corruption in the operating system we can’t simply let the corruption pass and possibly make things worse, so it crashes the machine to get out of a bad situation.

At this point we’d also like to know why the system was in a bad state, so the O/S writes the contents of memory out into a crash dump file. Some poor soul can then be chained to a chair and forced to stare at the dump in WinDbg until the bug is found.

Given their usefulness, it might also be beneficial to have a crash dump not only in cases that are fatal but in ones that are undesirable. For example, imagine that NDIS tries to reset your network adapter but the reset takes too long. We don’t necessarily want to crash the machine because of this, but we might want to generate a crash dump so we can analyze the state of the system and figure out what’s causing the delay.

This is the idea behind the Live Kernel Reports feature introduced in Windows 8. I personally didn’t notice this feature until last year, though it instantly intrigued me. Basically, any driver in the system can call the undocumented `DbgkWerCaptureLiveKernelDump` API and generate a kernel summary dump and/or a minidump in the `C:\Windows\LiveKernelReports` folder. This allows each driver to decide what conditions might require some additional scrutiny and non-intrusively generate a crash dump for further study. Check out the folder on your own system right now and you might even see a few dump files in there.

Out of curiosity I wanted to see which Windows drivers used this functionality on a clean install of 19H1 and received quite a few hits:

```plaintext
0: kd> x *!_imp_DbgkWerCaptureLiveKernelDump
ffff8ec0`fd5b59d0 win32kfull!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
ffff8ec0`fbee337f cdd!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
fffff803`d20862e8 cldflt!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
fffff803`d25021c8 srv2!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
fffff805`1f7db5a8 WPF!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
fffff805`1f8380f0 WppRecorder!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
fffff805`1fb95040 pdc!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
fffff805`201264b0 Ntfs!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
fffff805`204a52d8 ndis!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
fffff805`20916f00 tcpip!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
fffff805`20d2a3b0 USXHCI!_imp_DbgkWerCaptureLiveKernelDump = <no type information>
```

Note that this only indicates the loaded drivers uses this functionality. A string search in the `C:\Windows\System32\Drivers` directory uncovered even more.

† UFW

(CONTINUED ON PAGE 33)
To be fair, in general these Live Kernel Reports aren’t terribly interesting. After all, they represent non-fatal failures and thus shouldn’t be too much bother to the user. However, I spend a lot of time debugging system crashes that point to no obvious culprit. In those cases I like to think about what’s different about the crashing system that might give me some new clues as I try to solve the riddle. Maybe there’s a component that’s been logging errors to the Event Log for the last month. Or maybe there’s a bunch of minidumps from previous crashes that the user didn’t even notice. The crash dumps in the Live Kernel Reports folder work in the same way. For example, maybe the NIC has been generating Live Kernel Reports since the time the driver was last updated.

But I Want Live Kernel Reports Too!
I also happen to be a dev, and so I want Live Kernel Reports for my drivers too! Given that it’s an entirely undocumented feature I’m not foolish enough to want to ship code that generates them, but I see a huge value here in terms of testing. I’d like to let the test team go crazy beating on our code while the driver silently generates Live Kernel Reports along with error logs for the development team to inspect. Sure, we could just crash the machines, but in many cases that’s unnecessary and we want to see how our code recovers from error conditions anyway.

So, after kicking that around as a, “wouldn’t that be nice?” idea for a while, I finally got around to doing a quick feasibility study by generating a Live Kernel Report from a test driver. I’m not Mr. Reverse Engineer, but I can fumble around enough to come up with a function prototype with enough parameters to get me going. Note that I gave up after the first 6 parameters (mostly due to lack of time and interest, as you’ll soon see that this experiment quickly hit a brick wall):

```c
EXTERN_C
_IRQL_requires_(PASSIVE_LEVEL)
NTKERNELAPI
NTSTATUS
DbgkWerCaptureLiveKernelDump(
    _In_z_ PWCHAR ComponentName,
    _In_ ULONG LiveDumpCode,
    _In_ ULONG_PTR LiveDumpParam1,
    _In_ ULONG_PTR LiveDumpParam2,
    _In_ ULONG_PTR LiveDumpParam3,
    _In_ ULONG_PTR LiveDumpParam4,
    _In_ ULONG_PTR I,
    _In_ ULONG_PTR Dont,
    _In_ ULONG Know
);
```

We Know What We Know
And...We Know What we Don’t Know

We are not experts in everything. We’re not even experts in everything to do with Windows. But we think there are a few things that we do pretty darn well. We understand how the Windows OS works. We understand devices, drivers, and file systems on Windows. We’re pretty proud of what we know about the Windows storage subsystem.

What makes OSR unique is that we can explain these things to your team, provide you new insight, and if you’re undertaking a Windows system software project, help you understand the full range of your options. AND we also write kick-ass kernel-mode Windows code. Really. We do.

Why not fire-off an email and find out how OSR can help. If we can’t help you, we’ll tell you that, too.
Contact: sales@osr.com
Live Kernel Reports...(Cont.)

(Continued from page 33)

It took me a bit to get to that point and I was pretty excited when my module linked and loaded on the target. I then called the API and...........nothing happened. No crash, no debug output, no Live Kernel Report, nothing. Never one to give up so easily, I poked around and saw that there were DbgPrintEx calls being made inside this API and this led to two debug print filters that can be set to get more information:

0: kd> ed nt!Kd_WER_Mask f
0: kd> ed nt!Kd_CRASHDUMP_Mask f

With those flags set I received much more interesting output when I tried to generate the Live Kernel Report:

WERKERNELHOST: WerpCheckPolicy: Requested Policy is 2
WERKERNELHOST: System memory threshold met. Memory Threshold 274877906944 bytes, SystemMemory 8589398016 bytes
WERKERNELHOST: System threshold time not met. Threshold time 132051744578202800, Current time 132047486278525109
WERKERNELHOST: WerpCheckPolicy: Requested Policy 2 is higher than granted 0
WERKERNELHOST: CheckPolicy throttled dump creation for Component

Note the highlighted message about the “system threshold time” not being met. Presumably the numbers specified are timestamps (in decimal), so we can translate them into system times using the .formats command. Here is the result for the threshold time:

0: kd> .formats 0n132051744578202800
Evaluate expression:
Hex: 01d5245c`af8deb0
decimal: 132051744578202800
Octal: 00072511036257436620
Binary: 00000001 11010101 00100100 01011110 10101111 10001101 11011100 10110000
Chars: ..$
Time: Sun Jun 16 12:00:57.820 2019 (UTC - 4:00)
Float: low -2.58159e-010 high 7.8296e-038
Double: 7.89244e-300

And for the current time:

0: kd> .formats 0n132047486278525109
Evaluate expression:
Hex: 01d5207d`391d60b5
decimal: 132047486278525109
Octal: 0007251007647107260265
Binary: 00000001 11010101 00100000 01111101 00111001 00011101 01100000 10110101
Chars: ..
Time: Tue Jun 11 13:43:47.852 2019 (UTC - 4:00)
Float: low 0.000150087 high 7.82905e-038
Double: 7.88679e-300

Clearly the code is trying to avoid generating too many Live Kernel Reports and thus has put some arbitrary deadline in the future for the next time a dump may be written. I searched but could not find a way to bypass this (i.e. a, “no, please do wear out my SSD and fill my drive, I don’t care!” flag) but alas did not find one. This dashed any and all hopes I had for using this as a testing mechanism in the lab.

Of course, I simply could not leave this alone until I saw my driver generate a Live Kernel Report. Using the debugger I NOP’d out the offending check and, hurray, I saw my dump file!

<table>
<thead>
<tr>
<th>This PC \ Local Disk (C:) \ Windows \ LiveKernelReports</th>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OSRLK</td>
<td>6/11/2019 12:01 PM</td>
<td>File folder</td>
<td>1,935,852 KB</td>
</tr>
<tr>
<td></td>
<td>OSRLK-20190611-1200.dmp</td>
<td>6/11/2019 12:00 PM</td>
<td>DMP File</td>
<td></td>
</tr>
</tbody>
</table>

(Continued on page 35)
Live Kernel Reports...(Cont.)

(CONTINUED FROM PAGE 34)

However, not too long later the dump file disappeared on me. Checking out Process Monitor, I can see that sometime after the Live Kernel Report is generated WerFault.exe runs and cleans out the folder.

This doesn’t appear to happen every time and for every dump, but there’s some algorithm behind the scenes to make sure this folder doesn’t grow unbounded.

Not Generically Useful, But Still Useful
Sadly, but not surprisingly, this mechanism is way too undocumented and special purpose built to be generically useful. I’ll continue to look at Live Kernel Reports as part of analyzing systems though and through this experiment I’ve learned that just because there aren’t any dumps in the Live Kernel Reports folder it doesn’t mean that they aren’t being generated.
IoTarget, Stack Size and Requests—Oh My!

Guest Article

By
Nik Twerdochlib

While progressing through the WHQL battle for one of our drivers, we hit a proverbial wall; full speed. The "wall" popped up out of nowhere under Windows 8.1 while running the DevFund tests. Almost all these tests failed, yet the same test continued to pass on other platforms: Windows 7, Windows 10, Windows Server 2019... So, my first step down the path of enlightenment begins.

After gaining some better insight into the issue, we decided to make a change to how IOCTLs would be sent from the user mode service. Prior to this, a user mode service would open a handle to the device and pass IOCTLs directly. This driver emulates a physical device (Smart Card Reader), and as such we had been skeptical about handling the interaction with the device this way. A Smart Card Reader can be accessed a few different ways. So, it must support potentially having multiple connections. A symbolic link is even created to support access from legacy software. This meant care had to be taken with the handle the user mode service established. The user mode service also controls when one of these devices is created, what its configuration is, and when it is removed through a virtual bus driver.

The change we decided to implement was to remove the complexity of having the user mode service manage the additional handle to the Smart Card Reader device and instead have the user mode service communicate solely through the bus driver. This allowed us to simplify the logic in the service, reduce the overhead of managing the open handle to the device, and allow our custom IOCTLs to be processed as internal IOCTLs in the device. This change would also address the issue of contending with all the Smart Card IOCTLs being sent to the device by the Smart Card Manager (SCardMgr). It also gave us peace of mind of not having our custom IOCTLs accessible through publicly available device.

To provide the ability for the bus driver to accept IOCTLs for a device and then forward them to that device, an IoTarget is created for each new device created on the bus. A unique identification number is stored in a context for the IoTarget, allowing for a method of determining which IoTarget represents the desired target device. IOCTLs received by the bus driver are first formatted as internal IOCTLs using WdfIoTargetFormatRequestForInternalIoctl. A completion routine is added to allow for the bus driver to complete the Request, and then sent to the target device using WdfRequestSend asynchronously.

The Problem

Initial testing after all these changes were put in place was a little surprising. The Requests failed the format call, always! Our trusty companion WinDbg showed that the error was STATUS_REQUEST_NOT_ACCEPTED.

Alright, why? Based on the error code's "name" I immediately set out to understand why the function driver would not accept the request. At first, I suspected a discrepancy in the actual IOCTL value, but that would have been too easy. I dumped the WDF log in the debugger expecting to see some relevant message(s) about the request arriving and then being rejected. After all the error was STATUS_REQUEST_NOT_ACCEPTED. You know that level of disappointment you get when you don't see something you fully expected to see? Well, this was one of those times. There was nothing in the log containing any reference to the Request that had been attempted to be formatted and sent. Hopefully I am not the only developer that did not know exactly what WdfIoTargetFormatRequestForInternalIoctl is really doing under the hood...

Turning focus back to the bus driver, I dumped the WDF log and found the relevant entries:

347: FxIoTargetFormatIoctl - enter: WDFIOTARGET 0x0000307E69F15228, WDFREQUEST 0x0000307E6A7ED198, IOCTL 0x55ff9c0e, internal 0, input WDFMEMORY 0x0000307E6A7ED0C1, output WDFMEMORY 0x0000307E6A7ED0B1
348: FxRequestBase::ValidateTarget - Cannot reallocate PIRP for WDFREQUEST 0000307E6A7ED198 using WDFIOTARGET 0000307E69F15228, 0xc00000d0(STATUS_REQUEST_NOT_ACCEPTED)
349: FxIoTargetFormatIoctl - Exit WDFIOTARGET 0x0000307E69F15228, WDFREQUEST 0x0000307E6A7ED198, 0xc00000d0 (STATUS_REQUEST_NOT_ACCEPTED)

(CONTINUED ON PAGE 37)
IoTarget, Stack Size...(Cont.)

(CONTINUED FROM PAGE 36)

Confident that the problem was confined to the bus driver, I set about to understand why. The task I was trying to perform was a simple one, an implemented in hundreds of drivers. Why does it fail here? Time to leverage the recently open sourced WDF code by Microsoft.

In Search of the Truth

When I first learned Microsoft published the WDF source on github.com, I remember thinking “How cool!”. I went straight to it and gave it a quick run through. Now it was time for me to dig deep and understand. Searching for the phrase “Cannot reallocate PIPR for WDFREQUEST” led me to FxRequestBase::ValidateTarget. I worked my way backwards from this point to gain a better understanding of what was going on. From the WinDbg session, I knew that the WDFREQUEST was valid, and that its current IRP was also valid. So, I could ascertain from the code in this method that the call to FxIoTarget::HasValidStackSize was not failing, and thus it must be the call to FxIoTarget::HasEnoughStackLocations that was the culprit. Now the question as to why?

At this point the thought crossed my mind that perhaps I had pushed this particular Windows 8.1 test system just a smidge too far, and it would be worthwhile to switch to a fresh system just to confirm I wasn’t in fact going down a rabbit hole! On a fresh test system, the same issue occurred. It was also reproducible on a few different Windows 10 test systems.

To gain a better understanding of what was going on, I moved the driver over to a Windows 10 test system where I could step through the WDF code within the WinDbg session. This was a big help and proved extremely insightful. Stepping through the code landed me in FxIoTarget::HasEnoughStackLocations. This exposed that the WDFREQUEST had a smaller stack size than the target. Some deeper insight into this function driver: As stated earlier, it emulates a Smart Card Reader. For those whom have not developed a Smart Card Reader driver under windows, Microsoft provides a WDM library (smclib.lib) to make the implementation process much easier, which it does. This library assumes the responsibility of handling all Smart Card related IOCTLs and provides callbacks to allow for the hosting driver to customize certain functionality. As Smart Card IOCTLs arrive, the driver simply needs to pass them down to smclib with a call to SmartCardDeviceControl.

During device creation, we must increase the stack size by a value of 1 so that we can add a completion routine, allowing the driver to complete the request rather than through smclib. This little detail turned out to be one of the keys to understanding the problem.

The Ah-Ha (No Sh*t) Moment

Something I can honestly state as not having given much thought to it previously, is that even when a driver owns a Request, that does not necessarily mean the driver originated the underlying IRP that Framework Request represents. The Request in this scenario originates from a user mode service calling DeviceloControl. The Framework did not create the underlying IRP

(CONTINUED ON PAGE 38)
**IoTarget, Stack Size... (Cont.)**

(continued from page 37)

associated with the Request, instead it was dispatched to this driver. As a result of this, the Framework is not free to deallocate that original IRP and allocate a new one with the necessary number of I/O Stack Locations. Thus, the Framework cannot allocate the required additional stack locations and the call to `FxBaseRequest::HasEnoughStackLocations` fails.

This can be further verified by following the flow through the Framework source. Starting from where IRPs arrive in the Framework, `FxPkgIo::Dispatch`, an `FxIrp` object is created to represent the incoming IRP. `FxPkgIo::DispatchStep1` is then called which passes the request to any registered dynamic dispatch callbacks if one exists, otherwise it calls `FxPkgIo::DispatchStep2` which handles placing the request on the appropriate queue. In `FxPkgIo::DispatchStep2` a call to `FxRequest::_CreateForPackage` is made to create an `FxRequest` object associating the IRP to the device. The `FxRequest` object is created by a call to `FxRequestFromLookaside`, which allocates the request from the driver tracking pool. The `FxRequestFromLookaside` class is derived from `FxRequest` and passes in a key value as the third argument to the constructor of its base class: `FxRequestDoesNotOwnIrp`.

We now have confirmation that the driver does not own the underlying IRP and the answer to why the Framework isn’t correcting the stack size issue for us behind the scene.

**The Fix**

Having now identified the root cause of the issue as the stack size of the bus driver device being less than the stack size of the function driver device, what do we need to do to fix the issue? The stack size of the bus driver device must be increased to include enough stack locations to allow a Request to be forwarded to the function driver. In my case, the function driver gets initialized with a stack size of 2 which is then increased by 1 to accommodate for the use of a WDM IO completion routine. This leaves it with a stack size of 3. The IoTarget that is created to connect to the function device will increase the stack size by 1, so the stack size of the bus driver will need to be incremented by a value of 5 (the stack size presented in the IoTarget plus 1) to allow for the forward operation. This allows the check for having enough stack locations to pass when formatting the Request to forward it to an IoTarget. With this change in place, the bus driver was in fact able to successfully format the Request and forward it on to the child device.

The takeaway from this endeavor has caused me to reflect upon the intricacies of the relationship between WDF and WDM. These include the importance of not losing sight of WDM functionality underneath WDF, how beneficial Microsoft’s release of the WDF source really is, oh, and the importance of understanding the relationship between your drivers.

**Nik Twerdochlib got his introduction to programming at a young age shortly after his father brought home a Texas Instruments TI99. This soon gave way to a lifelong fascination with the interaction of software and hardware and embedded computing. Nik can be reached at nt@nikom.net.**
OSR is Hiring

Do You Have the “Right Stuff” to Join Our Team?

We’re Looking for Software Development Engineers (0 to 5 years experience)

If you’re into operating systems, how software interfaces with hardware, file systems, or just working on challenging software problems where performance and code size still matter... we might have a career for you at OSR.

We’re looking for folks to write and debug system software (both user-mode and kernel-mode) and work with clients as members of a small team to solve complex technical problems. Depending on your level of experience, interest, and expertise you might also contribute to solution architectures, author design documents, perform code reviews, perform crash dump and problem analysis (for in-house and client problems), teach some of the seminars we offer throughout the world, and/or write technical papers or articles. There are many different ways you can contribute at OSR.

While Windows kernel-mode architecture, design, and programming are what we do most here at OSR, we also do a substantial (and growing) amount of kernel-mode work on Linux and (even) MacOS.

Right now, we’re looking for less experienced engineers: From engineers just finishing their studies to those who have as much as 5 years of system software development experience.

Those that join our engineering team are passionate about operating system internals. They live for working on some of the world’s most interesting, and most difficult, system software projects. They thrive on growing and demonstrating their expertise daily. They welcome the opportunity to work with some of the most exciting companies and technologies in the computing industry. They welcome working with clients from differing disciplines, backgrounds and cultures. They revel in the camaraderie that comes with working on a small team that can “make a difference”. And they enjoy seeing the results of their work first-hand and reaping the rewards that come from it.

If this all sounds good to you, then we VERY much want to hear from you. Drop a note with your resume and tell us why we can’t afford not to hire you!

NOTES: We’re sorry, but because we’re a small company we absolutely do not have the resources to support visa requests (including H1-B, B-1, or OPT extensions to F-1 visas), with the exception of TN visas for residents of Canada or Mexico. These positions also initially require working in Manchester, NH, USA. We cannot accommodate applicants who must work remotely.

Contact: recruiting@osr.com

Transparent File Encryption

- OSR supplies all kernel-mode code
- You dynamically determine policy per file, on each access, entirely from user mode.
- Quality, commented, sample code supplied.
- High performance.

Windows

Proven in multiple commercial products.

FESF Evaluation Editions are available FREE from OSR!

Linux

Fully interoperable with Windows version!

Contact OSR Sales to learn more!
OSR Seminar Schedule

<table>
<thead>
<tr>
<th>Seminar</th>
<th>Dates</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDF Drivers I: Core Concepts</td>
<td>21-25 October</td>
<td>Manchester, NH (&amp; ONLINE)</td>
</tr>
<tr>
<td>Internals &amp; Software Drivers</td>
<td>18-22 November</td>
<td>Dulles/Sterling, VA</td>
</tr>
<tr>
<td>Kernel Debugging &amp; Crash Analysis</td>
<td>30 March—3 April 2020</td>
<td>Manchester, NH</td>
</tr>
<tr>
<td>Developing File Systems Mini-Filters</td>
<td>27 April—1 May 2020</td>
<td>Manchester, NH</td>
</tr>
</tbody>
</table>

More Dates/Locations Available—See [website](#) for details

Join OSR’s Seminar Update Mailing List

OSR Seminars
We Practice What We Teach For a Reason

When we say “we practice what we teach”, this mantra directly translates into the value we bring to our seminars. But don’t take our word for it...

One of the best. The first thing I did when I got back to work was plan the next OSR seminar to attend!

-Recent attendee of an OSR seminar

THE NT INSIDER - Hey...Get Your Own!

Just visit The NT Insider page and subscribe with your email — and you’ll get notification whenever we release a new issue of The NT Insider.