DPWs are the new DPCs : Deferred Procedure Waits in Windows 10 21H1

windows-internals.com/dpws-are-the-new-dpcs-deferred-procedure-waits-in-windows-10-21h1

By Yarden Shafir

With the Windows `21H1` (Iron/"Fe") feature complete deadline looming, the last few Dev Channel builds have had some very interesting changes and additions, which will probably require a few separate blog posts to cover fully. One of those was in a surprising part of the code – object wait dispatching.

The new build introduced a few new functions:

- `KeRegisterObjectDpc` (despite the name, it's an internal non-exported function)
- `ExQueueDpcEventWait`
- `ExCancelDpcEventWait`
- `ExCreateDpcEvent`
- `ExDeleteDpcEvent`

All those functions are part of a new and interesting functionality – the ability to wait on an (event) object and to execute a `DPC` when it becomes signaled. Until now, if a driver wanted to wait on an object it had to do so synchronously – the current thread would be put in a wait state until the object that is waited on was signaled, or the wait timed out (or an `APC` executed, if the wait was alertable). User mode applications typically perform waits in the same manner, however, since Windows 8, they've also have had the ability to perform asynchronous waits through the Thread Pool API. This new functionality associates an I/O Completion Port with a "Wait Packet", obviating the need to have a waiting thread.

The change in `21H1`, through the addition of these APIs, marks a major change for kernel-mode waits by introducing kernel-mode asynchronous waits: a driver can now supply a `DPC` that will be executed when the event object that is waited on is signaled all while continuing its execution in the meantime.

The Mechanism

To use this new capability, a driver must first initialize a so-called "DPC Event". To initialize this structure we have the new API `ExCreateDpcEvent`:

```
NTSTATUS
ExCreateDpcEvent (
    _Outptr_ PVOID *DpcEvent,
    _Outptr_ PKEVENT *Event,
    _In_ PKDPC Dpc
);
```

Internally, this allocates a new undocumented structure that I chose to call `DPC_WAIT_EVENT`:

```
typedef struct _DPC_WAIT_EVENT
{
    KWAIT_BLOCK WaitBlock;
    PKDPC Dpc;
    PKEVENT Event;
} DPC_WAIT_EVENT, *PDPC_WAIT_EVENT;
```

This API receives a DPC that the caller must have previously initialized with `KeInitializeDpc` (you can guess who spent a day debugging things by forgetting to do this), and in turn creates an event object and allocates a `DPC_WAIT_EVENT` structure that is returned to the caller, filling in a pointer to the caller's `DPC`, the newly allocated event, and setting the wait block state to `WaitBlockInactive`.

Then, the driver needs to call the new `ExQueueDpcEventWait` function, passing in the structure:

```
BOOLEAN
ExQueueDpcEventWait (
    _In_ PDPC_WAIT_EVENT DpcEvent,
    _In_ BOOLEAN QueueIfSignaled
    )
{
    if (DpcEvent->WaitBlock.BlockState != WaitBlockInactive)
    {
        RtlFailFast(FAST_FAIL_INVALID_ARG);
    }
    return KeRegisterObjectDpc(DpcEvent->Event,
                               DpcEvent->Dpc,
                               &DpcEvent->WaitBlock,
                               QueueIfSignaled);
}
```

As can be seen, this function is very simple – it unpacks the structure and sends the contents to the internal `KeRegisterObjectDpc`:

```
BOOLEAN
KeRegisterObjectDpc (
    _In_ PVOID Object,
    _In_ PRKDPC Dpc,
    _In_ PKWAIT_BLOCK WaitBlock,
    _In_ BOOLEAN QueueIfSignaled
);
```

You might wonder, like me – doesn't the "`e`" in "`Ke`" stand for "exported"? Was I lied to the whole time? Is this a mistake? Was this a last minute change? Does MS not have any design or code review? I'm as confused as you are.

But before talking about `KeRegisterObjectDpc`, we need to investigate another small detail. To enable this functionality, the `KWAIT_BLOCK` structure can now store a `KDPC` to queue, and the `WAIT_TYPE` enumeration has a new `WaitDpc` option:

```
typedef struct _KWAIT_BLOCK
{
    LIST_ENTRY WaitListEntry;
    UCHAR WaitType;
    volatile UCHAR BlockState;
    USHORT WaitKey;
#if defined(_WIN64)
    LONG SpareLong;
#endif
    union {
        struct KTHREAD* Thread;
        struct KQUEUE* NotificationQueue;
        struct KDPC* Dpc;
    };
    PVOID Object;
    PVOID SparePtr;
} KWAIT_BLOCK, *PKWAIT_BLOCK, *PRKWAIT_BLOCK;
```

```
typedef enum _WAIT_TYPE
{
    WaitAll,
    WaitAny,
    WaitNotification,
```

```
    WaitDequeue,
    WaitDpc,
} WAIT_TYPE;
```

Now we can look at `KeRegisterObjectDpc`, which is pretty simple and does the following:

1. Initializes the wait block
    1. Sets the `BlockState` field to `WaitBlockActive`,
    2. Sets the `WaitType` field to `WaitDpc`
    3. Sets the `Dpc` field to point to the received `DPC`
    4. Sets the Object field to the received object.
2. Raises the `IRQL` to `DISPATCH_LEVEL`
3. Acquires the lock for the object, found in its `DISPATCHER_HEADER`.
4. If the object is not signaled – inserts the wait block into the wait list for the object and releases the lock, then lowers the `IRQL`
5. Otherwise, if the object is signaled:
    1. Satisfies the wait for the object, resetting the signal state as required for the object
    2. If the `QueueIfSignaled` parameter was set, goes to step 3
    3. Otherwise,
        1. Sets `BlockState` to `WaitBlockInactive`
        2. Queues the `DPC`

    Releases the lock and calls `KiExitDispatcher` (which will lower the `IRQL` and make the `DPC` execute immediately).

Then the function returns. If the object was not signaled, the driver execution will continue and when the object gets signaled, the `DPC` will be executed. If the object is already signaled, the `DPC` will be executed immediately (unless the `QueueIfSignaled` parameter was set to `TRUE`)

If the wait is no longer needed, the driver should call `ExCancelDpcEventWait` to remove the wait block from the wait queue. And when the event is not needed it should call `ExDeleteDpcEvent` to dereference the event and free the opaque `DPC_WAIT_EVENT` structure.

Meanwhile, the various internal dispatcher functions that take care of signaling an object have been extended to handle the `WaitDpc` case – instead of unwaiting the thread (`WaitAny` / `WaitAll`), or waking up a queue waiter (`WaitNotification`), a call to `KeInsertQueueDpc` is now done for the `WaitDpc` case (since wait satisfaction is done at `DISPATCH_LEVEL`, the `DPC` will then immediately execute once `KiExitDispatcher` is called by one of these functions).

The Limitations

You might have noticed that while the functionality in `KeRegisterObjectDpc` is generic, all these structures and exported functions only support an event object. Furthermore, when looking inside `ExCreateDpcEvent`, we can see that it *only* creates an event object:

```
status = ObCreateObject(KernelMode,
                        ExEventObjectType,
                        NULL,
                        KernelMode,
                        NULL,
                        sizeof(KEVENT),
                        0,
                        0,
                        &event);
```

But as `KeRegisterObjectDpc` suggests, an event is not the only object that can be asynchronously waited on. The usage of `KiWaitSatisfyOther` suggests that any generic dispatcher object, except for mutexes, which need to handle ownership rules, can be used. Since a driver might need to wait on a process, a thread, a semaphore, or any other object — why are we only allowed to wait on an event here?

The answer in this case is probably that this was not designed to be a generic feature available to all drivers. So far, I could only see one Windows component calling these new functions – `Vid.sys` (the Hyper-V Virtualization Infrastructure Driver) Digging deeper, it looks like it is using this new capability to implement the new `WHvCreateTrigger` API added to the documented Hyper-V Platform API in `WinHvPlatform.h`. "Triggers" are a new exposed `21H1` functionality to send virtual interrupts to a Hyper-V Partition. The importance of Microsoft's Azure/Hyper-V platform play is clearly evident here – low level changes to the kernel dispatcher, for the first time in a decade, simply to optimize the performance of virtual machine-related APIs.

As such, since it is only designed to support this one specific case, this feature is built to only wait on an event object. But even with that in mind, the design is a bit funny – `ExCreateDpcEvent` will create an event object and return it to the caller, which then has to re-open it with `ObOpenObjectByPointer` to use it in any way, since most wait-related APIs require a `HANDLE` (as does exposing the object to user-mode, as `Vid.sys` intends to do). And we can see vid.sys doing exactly that:

```
ExInitializeRundownProtection(&vidStruct->ex_rundown_ref58);
KeInitializeDpc(&vidStruct->Dpc, VidTriggerpDpcRoutine, vidStruct);
status = ExCreateDpcEvent(
             &vidStruct->WaitStruct,
             &event,
             &vidStruct->Dpc);
if ( status >= 0 )
{
    status = ObOpenObjectByPointer(
                 event,
                 0,
                 NULL,
                 EVENT_MODIFY_STATE,
                 ExEventObjectType,
                 NULL,
                 &Handle);
    if ( status >= 0 )
```

Why not simply expose `KeRegisterObjectDpc` and let it receive an object pointer that will be waited on, since this function doesn't care about the object type? Why do we even need a new structure to manage this information? I don't know. The current implementation doesn't seem like the most logical one, and it limits the feature significantly, but it is the Microsoft way.

If I had to guess, I would expect to see this feature changing in the future to support more object types as Microsoft internally finds more uses for asynchronous waits in the kernel. I will not be surprised to see an `ExQueueDpcEventWaitEx` function added soon... and perhaps documenting this API to 3rd parties.

But not all is lost. If you're willing to bend the rules a little and upset a few people in the OSR forums, you can wait on any non-mutex (dispatcher) object you want, simply by replacing the pointer inside the `DPC_WAIT_EVENT` structure that is returned back to you. Neither `ExQueueDpcEventWait` or `KeRegisterObjectDpc` care about which type of object is being passed in, as long as it's a legitimate dispatcher object. I'm sure there's an `NT_ASSERT` in the checked build, but it's not like those still exist.

The risk here (as OSR people will gladly tell you) is that the new structure is undocumented and might change with no warning, as are the functions handling it. So, replacing the pointer and hoping that the offset hasn't changed and that the functions will not be affected by this change is a risky choice that is not recommended in a production environment. Now that I've said it, I have no doubt we will see crash dumps caused by AV products attempting to do exactly that, poorly.

PoC

To demonstrate how this mechanism works and how it can be used for objects other than events I wrote a small driver that registers a `DPC` that waits for a process to terminate.

On `DriverEntry`, this driver initializes a push lock that will be used later. It also registers a process creation callback:

```
NTSTATUS
DriverEntry (
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
    )
{
    DriverObject->DriverUnload = DriverUnload;
    ExInitializePushLock(&g_WaitLock);
    return PsSetCreateProcessNotifyRoutineEx(&CreateProcessNotifyRoutineEx, FALSE);
}
```

Whenever our `CreateProcessNotifyRoutineEx` callback is called, it checks if the new process name ends with "cmd.exe":

```
VOID
CreateProcessNotifyRoutineEx (
    _In_ PEPROCESS Process,
    _In_ HANDLE ProcessId,
    _In_ PPS_CREATE_NOTIFY_INFO CreateInfo
    )
{
    NTSTATUS status;
    DECLARE_CONST_UNICODE_STRING(cmdString, L"cmd.exe");

    UNREFERENCED_PARAMETER(ProcessId);

    //
    // If process name is cmd.exe, create a dpc
    // that will wait for the process to terminate
    //
    if ((!CreateInfo) ||
        (!RtlSuffixUnicodeString(&cmdString, CreateInfo->ImageFileName, FALSE)))
    {
        return;
    }
    ...
}
```

If the process is cmd.exe, we will create a `DPC_WAIT_EVENT` structure that will wait for the process to be signaled, which happens when the process terminates. For the purpose of this PoC I wanted to keep things simple and avoid having to keep track of multiple wait blocks. So only the first cmd.exe process will be waited on and the rest will be ignored.

First, we need to declare some global variables for the important structures, as well as the lock that we initialized on DriverEntry and the DPC routine that will be called when the process terminates:

```
static KDEFERRED_ROUTINE DpcRoutine;
PDPC_WAIT_EVENT g_DpcWait;
EX_PUSH_LOCK g_WaitLock;
KDPC g_Dpc;
PKEVENT g_Event;

static
void
DpcRoutine (
    _In_ PKDPC Dpc,
    _In_ PVOID DeferredContext,
    _In_ PVOID SystemArgument1,
    _In_ PVOID SystemArgument2
    )
{
    DbgPrintEx(DPFLTR_IHVDRIVER_ID,
               DPFLTR_ERROR_LEVEL,
               "Process terminated\n");
}
```

Then, back in our process creation callback, we will initialize the `DPC` object and allocate a `DPC_WAIT_EVENT` structure using `KeInitializeDpc` and `ExCreateDpcEvent`. To avoid a race we will use our lock.

```
void
CreateProcessNotifyRoutineEx (
    ...
    )
{
    ...
    ExAcquirePushLockExclusive(&g_WaitLock);
    if (g_DpcWait == nullptr)
    {
        KeInitializeDpc(&g_Dpc, DpcRoutine, &g_Dpc);
        status = ExCreateDpcEvent(&g_DpcWait, &g_Event, &g_Dpc);
        if (!NT_SUCCESS(status))
        {
            DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                       DPFLTR_ERROR_LEVEL,
                       "ExCreateDpcEvent failed with status: 0x%x\n",
                       status);
            ExReleasePushLockExclusive(&g_WaitLock);
            return;
        }
        ...
    }
    ExReleasePushLockExclusive(&g_WaitLock);
}
```

`ExCreateDpcEvent` creates an event object and places a pointer to it in our new `DPC_WAIT_EVENT` structure. But since we want to wait on a process, we need to replace that event pointer with the pointer to the `EPROCESS` of the new `Cmd.exe` process. Then we can go on to queue our wait block for the process:

```
void
CreateProcessNotifyRoutineEx (
    _In_ PEPROCESS Process,
```

```
    ...
    )
{
    NTSTATUS status;
    //
    // Only wait on one process
    //
    ExAcquirePushLockExclusive(&g_WaitLock);
    if (g_DpcWait == nullptr)
    {
        KeInitializeDpc(&g_Dpc, DpcRoutine, &g_Dpc);
        status = ExCreateDpcEvent(&g_DpcWait, &g_Event, &g_Dpc);
        if (!NT_SUCCESS(status))
        {
            DbgPrintEx(DPFLTR_IHVDRIVER_ID,
                       DPFLTR_ERROR_LEVEL,
                       "ExCreateDpcEvent failed with status: 0x%x\n",
                       status);
            ExReleasePushLockExclusive(&g_WaitLock);
            return;
        }
        NT_ASSERT(g_DpcWait->Object == g_Event);
        g_DpcWait->Object = (PVOID)Process;
        ExQueueDpcEventWait(g_DpcWait, TRUE);
    }
    ExReleasePushLockExclusive(&g_WaitLock);
}
```

And that's it! When the process terminates our `DPC` routine will be called, and we can choose to do whatever we want there:

```
2: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffe04`5743eb08 fffff805`7f285ce5     DpcWaitOnProcess!DpcRoutine [C:\U
01 ffffe04`5743eb10 fffff805`7f284cfd     nt!KiExecuteAllDpcs+0x505
02 ffffe04`5743ed00 fffff805`7f422c55     nt!KiRetireDpcList+0x24d
03 ffffe04`5743efb0 fffff805`7f422a40     nt!KxRetireDpcList+0x5
04 ffffe04`596533e0 fffff805`7f422066     nt!KiDispatchInterruptContinue
05 ffffe04`59653410 fffff805`7f25c739     nt!KiDpcInterrupt+0x326
06 ffffe04`596535a0 fffff805`7f2ef5e9     nt!KiExitDispatcher+0x4c9
07 ffffe04`59653640 fffff805`7f6ce953     nt!KeSetProcess+0x245
08 ffffe04`596536b0 fffff805`7f6c2b57     nt!PspRundownSingleProcess+0x237
09 ffffe04`59653740 fffff805`7f779f68     nt!PspExitThread+0x56b
0a ffffe04`59653830 fffff805`7f21224d     nt!KiSchedulerApcTerminate+0x38
0b ffffe04`59653870 fffff805`7f41feb0     nt!KiDeliverApc+0x62d
0c ffffe04`59653930 fffff805`7f42d25f     nt!KiInitiateUserApc+0x70
0d ffffe04`59653a70 00007ffc`f3cba304     nt!KiSystemServiceExit+0x9f
0e 0000003e`1f0ff718 00000000`00000000     0x00007ffc`f3cba304
```

The only other thing we need to remember is to clean up after ourselves before unloading, by setting the pointer back to the event (that we saved for that purpose), canceling the wait and deleting the `DPC_WAIT_EVENT` structure:

```
VOID
DriverUnload (
    _In_ PDRIVER_OBJECT DriverObject
```

```
    )
{
    UNREFERENCED_PARAMETER(DriverObject);

    PsSetCreateProcessNotifyRoutineEx(&CreateProcessNotifyRoutineEx, TRUE);

    //
    // Change the DPC_WAIT_EVENT structure to point back to the event,
    // cancel the wait and destroy the structure
    //
    if (g_DpcWait != nullptr)
    {
        g_DpcWait->Object = g_Event;
        ExCancelDpcEventWait(g_DpcWait);
        ExDeleteDpcEvent(g_DpcWait);
    }
}
```

Forensics

Apart from the legitimate uses of asynchronous wait for drivers, this is also a new and stealthy way to wait on all different kinds of objects without using other, more well-known ways that are easy to notice and detect, such as using process callbacks to wait on process termination.

The main way to detect whether someone is using this technique is to inspect the wait queues of objects in the system. For example, let's use the Windbg Debugger Data Model to inspect the wait queues of all processes in the system. To get a nice table view we'll only show the first wait block for each process, though of course that doesn't give us the full picture:

```
dx -g @$procWaits = @$cursession.Processes.Where(p =>
(__int64)&p.KernelObject.Pcb.Header.WaitListHead !=
(__int64)p.KernelObject.Pcb.Header.WaitListHead.Flink).Select(p =>
Debugger.Utility.Collections.FromListEntry(p.KernelObject.Pcb.Header.WaitListHead,
"nt!_KWAIT_BLOCK", "WaitListEntry")[0]).Select(p => new { WaitType = p.WaitType, BlockState =
p.BlockState, Thread = p.Thread, Dpc = p.Dpc, Object = p.Object, Name = ((char*)
((nt!_EPROCESS*)p.Object)->ImageFileName).ToDisplayString("sb")})
```

	WaitType	BlockState	(+) Thread	(+) Dpc	Object	Name
[0x1f4]	0x1	0x4	0xffffcb8afe410040	0xffffcb8afe410040	0xffffcb8afe2b8080	csrss.exe
[0x240]	0x2	0x4	0xffffcb8b001563c0	0xffffcb8b001563c0	0xffffcb8afeef9080	wininit.exe
[0x248]	0x2	0x4	0xffffcb8b001563c0	0xffffcb8b001563c0	0xffffcb8afef55140	csrss.exe
[0x2a8]	0x2	0x4	0xffffcb8b001563c0	0xffffcb8b001563c0	0xffffcb8afefb6080	winlogon.exe
[0x2d4]	0x2	0x4	0xffffcb8afeced780	0xffffcb8afeced780	0xffffcb8b0002b180	services.exe
[0x2e8]	0x2	0x4	0xffffcb8afeced780	0xffffcb8afeced780	0xffffcb8b00031180	lsass.exe
[0x368]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b00049080	svchost.exe
[0x3ec]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b0012f240	svchost.exe
[0x1a0]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b0002f080	svchost.exe
[0x48c]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03c96240	svchost.exe
[0x498]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03ca3080	svchost.exe
[0x4a4]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03cb5080	svchost.exe
[0x4ac]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03cb8080	svchost.exe
[0x518]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03d020c0	svchost.exe
[0x538]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03d09080	svchost.exe
[0x554]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03d0d080	svchost.exe
[0x584]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03d630c0	svchost.exe
[0x5c8]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03d8c080	svchost.exe
[0x608]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03def080	svchost.exe
[0x660]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03dbd080	vm3dservice.ex
[0x670]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03e37080	svchost.exe
[0x6c4]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03eb7080	svchost.exe
[0x6cc]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03df8080	svchost.exe
[0x6d4]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03eb8080	svchost.exe
[0x6e8]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03eb6080	svchost.exe
[0x76c]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03f2c080	svchost.exe
[0x790]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03f5a080	svchost.exe
[0x798]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b03f680c0	svchost.exe
[0x7e8]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b0507d0c0	svchost.exe
[0x7f4]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b05082080	svchost.exe
[0x814]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b05129080	svchost.exe
[0x828]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b05133080	svchost.exe
[0x864]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b051a00c0	svchost.exe
[0x86c]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8afdfb9080	svchost.exe
[0x878]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8afdfb3080	svchost.exe
[0x8d0]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8afdf73080	svchost.exe
[0x920]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8afdf1a080	svchost.exe
[0x974]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8afdfd9080	svchost.exe
[0x99c]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8afef840c0	spoolsv.exe
[0x9d4]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8afefa5080	svchost.exe
[0xa00]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b052b4080	svchost.exe
[0xb04]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b053bc0c0	svchost.exe
[0xb0c]	0x2	0x4	0xffffcb8afefcee00	0xffffcb8afefcee00	0xffffcb8b053c0080	svchost.exe

We mostly see here waits of type `WaitNotification` (`2`), which is what we expect to see – user-mode threads asynchronously waiting for processes to exit. Now let's run our driver and run a new query which will only pick processes that have wait blocks with type `WaitDpc` (`4`):

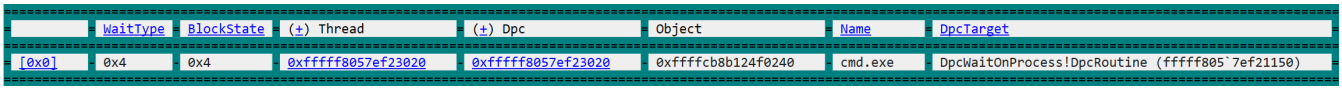
```
dx @$dpcwaits = @$cursession.Processes.Where(p =>
(__int64)&p.KernelObject.Pcb.Header.WaitListHead !=
(__int64)p.KernelObject.Pcb.Header.WaitListHead.Flink &&
Debugger.Utility.Collections.FromListEntry(p.KernelObject.Pcb.Header.WaitListHead,
"nt!_KWAIT_BLOCK", "WaitListEntry").Where(p => p.WaitType == 4).Count() != 0)
```

```
[0x6b0]          : cmd.exe [Switch To]
```

Now we only get one result – the cmd.exe process that our driver is waiting on. Now we can dump its whole wait queue and see who is waiting on it. We will also use a little helper function to show us the symbol that the `DPC` 's `DeferredRoutine` is pointing to:

```
dx -r0 @$getsym = (x => Debugger.Utility.Control.ExecuteCommand(".printf\"%y\", " +
((__int64)x).ToDisplayString("x")))
```

```
dx -g
Debugger.Utility.Collections.FromListEntry(@$dpcwaits.First().KernelObject.Pcb.Header.WaitListHead,
"nt!_KWAIT_BLOCK", "WaitListEntry").Select(p => new { WaitType = p.WaitType, BlockState =
p.BlockState, Thread = p.Thread, Dpc = p.Dpc, Object = p.Object, Name = ((char*)
((nt!_EPROCESS*)p.Object)->ImageFileName).ToDisplayString("sb"), DpcTarget = (@$getsym(p.Dpc-
>DeferredRoutine))[0]})
```



	WaitType	BlockState	(+) Thread	(+) Dpc	Object	Name	DpcTarget
[0x0]	0x4	0x4	0xfffff8057ef23020	0xfffff8057ef23020	0xffffcb8b124f0240	cmd.exe	DpcWaitOnProcess!DpcRoutine (fffff805`7ef21150)

Only one wait block is queued for this process and its pointing to our driver!

This analysis process can also be converted to JavaScript to have a bit more control over the presentation of the results, or to `C` to automatically check the wait queues of different objects (keep in mind it is extremely unsafe to do this at runtime due to the lock synchronization required – using the `COM/C++` Debugger API to do forensics on a memory dump or live dump is the preferred way to go).

Conclusion

This new addition to the Windows kernel is exciting since it allows the option of asynchronous waits for drivers, a capability that only existed for user-mode until now. I hope we will see this extended to properly support more object types soon, making this feature generically useful to all drivers in various cases.

The implementation of all the functions discussed in this post can be found here.

Read our other blog posts: