Ring 0 to Ring -1 Attacks

HYPER-V IPC INTERNALS

SYSCAN 2015

ALEX IONESCU

@AIONESCU

WHO AM I?

Chief Architect at CrowdStrike, a security startup

Previously worked at Apple on iOS Core Platform Team

Co-author of *Windows Internals 5th and 6th Editions*

Reverse engineering NT since 2000 – main kernel developer of ReactOS

Instructor of worldwide Windows Internals classes

Conference speaking:

- SyScan 2015-2012
- NoSuchCon 2014-2013, Breakpoint 2012
- Recon 2014-2010, 2006
- Blackhat 2015, 2013, 2008

For more info, see www.alex-ionescu.com

WHAT THIS TALK IS ABOUT

The Microsoft Hypervisor (Hyper-V/Viridian) was introduced almost a decade ago

- Originally for Server only, it now ships on Clients too
- Powers not just Windows, but Azure and Xbox One too

Very few internal details on it have ever emerged

Ironically, Microsoft had the best details for it back in the WinHEC days

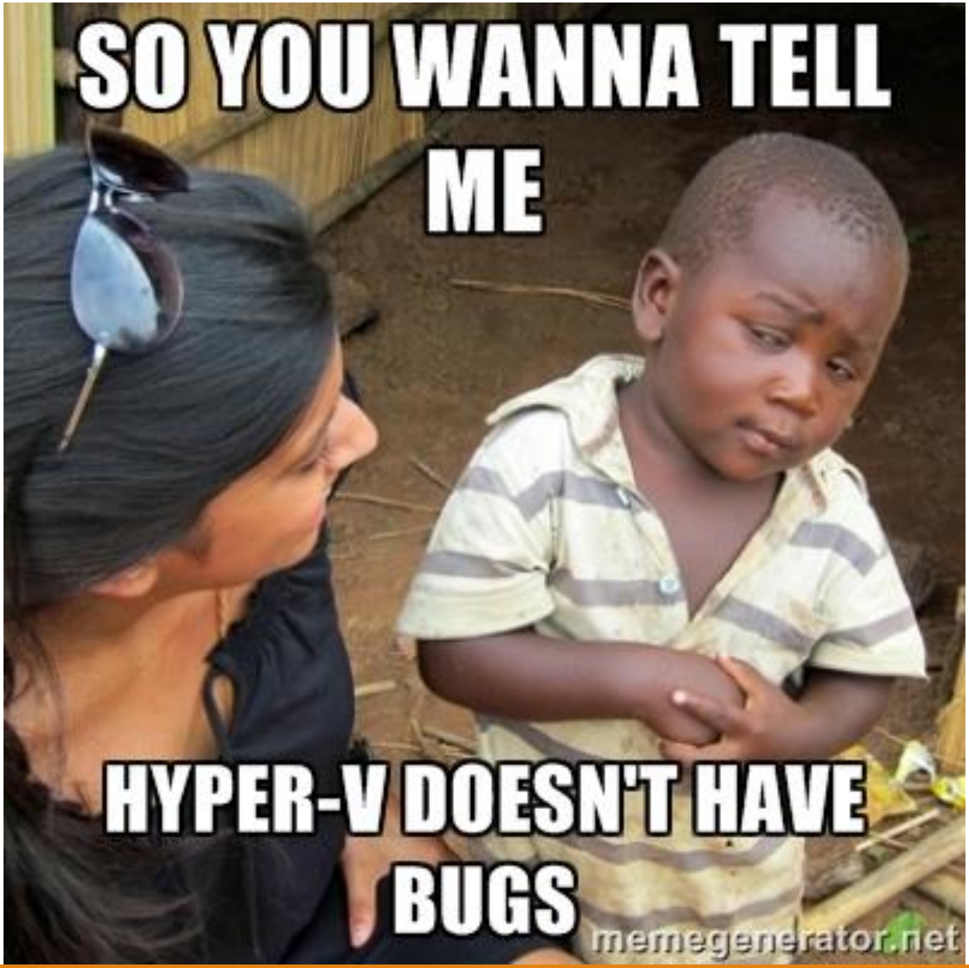
- [ref: Brandon Baker, Maryrita Steinhour]

Some external researchers have looked at it

- MS11-047, MS13-092
- [ref: ENRW / Matthias Luft & Felix Wilhelm]
- “Any hypervisor is not a new security layer; it’s a new place to find bugs.”

Nobody has talked about how to *interface* with it (just fuzzing for bugs)

TWO BUGS. IN. A. DECADE



WHAT THIS TALK IS ALSO ABOUT

~~Azure 0 Days / Unicorns~~

~~Hypervisor Rootkits / Blue, purple, red, magic, or any kind of pills~~

Providing helpful direction on how to interface/play/mess with Hyper-V

- For research and learning purposes
- Please don’t be an idiot and productize/ship any of this

Unfortunately, interacting with Hyper-V requires some knowledge of Windows driver development

- And, especially, PnP Driver Development ☹
- PnP Driver Development Sucks. Seriously. But it’s OK. WE HAVE MEMES.

And of course, a few interesting bugs/security issues

A few brief notes on the future of Hyper-V in Windows 10

OUTLINE

Hyper-V Architecture in 10 minutes or less ~~or your money back~~
- VMM Type 1 and Type 2
- Overall Architecture / VMBus / VMWP / VSC
- Partitions, Synthetic Interrupt Controller (SynIC), Overlay Pages

Programming With Hyper-V
- Hypervisor Top Level Functional Specification Documentation
- Hypervisor Development Kit (HDK) Headers, WinHv and Vid Library
- Hypercalls
- IPC Ports (Events, Messages, Monitors)

Windows PnP Driver Development ~~in 10 minutes or less~~

Demo Time

QA & Wrap-up

Hyper-V Architecture

VMM Types

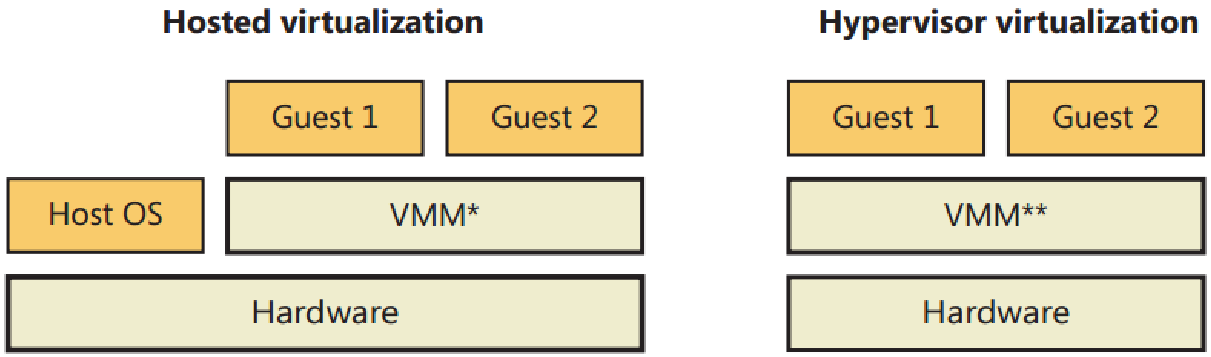
Type-2 VMM: Host OS runs Virtual Machine, which runs guest environments

- .NET CLR, Java VM

Hybrid VMM: Host OS runs with Virtual Machine, which runs guest environments

- VMWare, QEMU, Virtual PC

Type-1 VMM: Virtual Machine runs on barebones hardware, which runs Host OS and guest environments



Hyper-V Core Boot

The Hyper-V Core is composed of the hypervisor kernel, loader, boot driver and debugger transport

- Hvloader.exe/efi (Windows 8+) – Hypervisor Loader
- Hvix64.exe – VMX Hypervisor Kernel
- Hvax64.exe – SVM Hypervisor Kernel
- Kdhvcom.dll – Hypervisor Kernel Debugger Transport Library
- Hvservice.sys (Windows 7: hvboot.sys) – Hypervisor Boot Driver

Before Windows 8, an early boot driver (hvboot.sys) parsed boot options to look for the hypervisor settings

- *hvboot!HbHvLaunchHypervisor* uses the BAL to launch Hvix64.exe or Hvax64.exe based on detected platform type

In Windows 8+, the Boot Loader (Winload.exe/efi) calls *OslArchHypervisorSetup* to check for BCD Options

- Calls *HvlpLaunchHvLoader* as needed, which loads Hvloader.efi/exe

Partitions

Each VM Instance, or unit of isolation, is called a *partition*

The core virtualization stack runs in the *root* partition, which has full access to hardware

- However, the root partition also runs virtualized!

The Hyper-V Management Services on the root partition create child partitions as needed

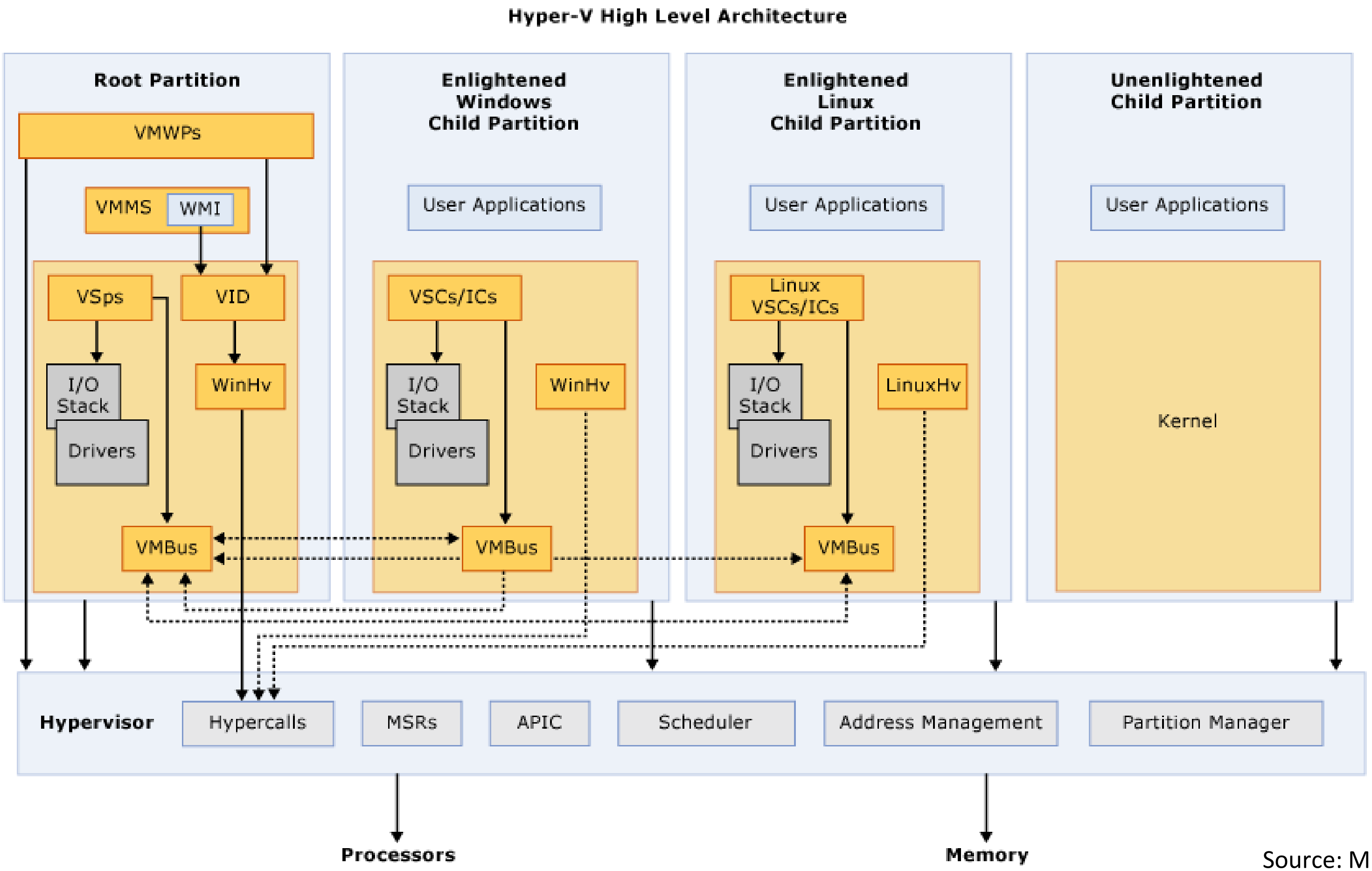
Partitions communicate to the hypervisor kernel by using *hypercalls*

- Similar to system calls, but with call-specific ACLs

Partitions can be

- Root/Parent Partitions (Windows)
- Enlightened Child Partitions (Windows or Linux)
- Unenlightened Child Partitions (3rd party OS or legacy Linux)

Hyper-V Architecture



Source: Microsoft © TechNet

Hyper-V Devices

Child partitions can have access to hardware by using Virtual Devices

Virtual Devices (Vdevs) are implemented as a pair:

- Virtualization Service Consumers (VSCs), which run on the child and redirect device requests
- Virtualization Service Providers (VSPs), which run on the root partition and handle device access requests from children partitions

Communication is done through the Virtual Machine Bus (VMBus)

- Manages “channels” between different root and children partitions

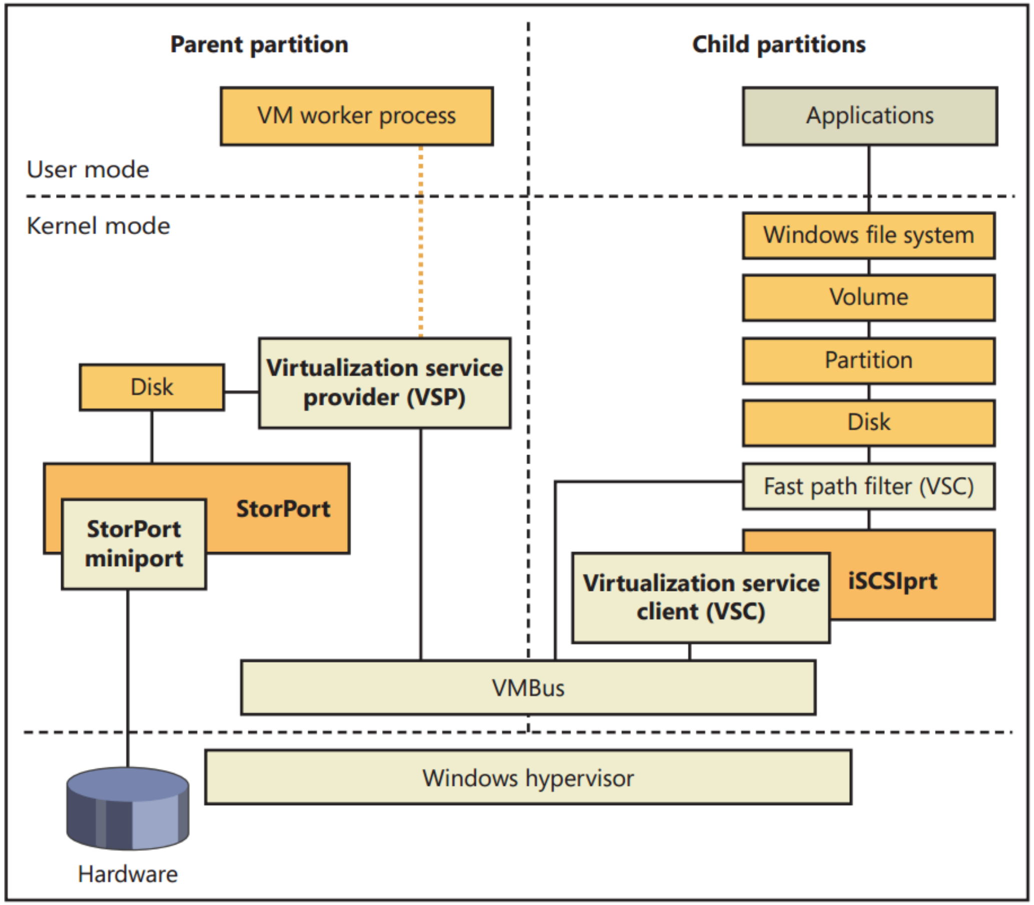
Sometimes called ICs (Integration Components) or Synthetic Devices

- “Enlightened I/O” is yet another term

For legacy devices (Serial Port, etc), emulated devices are used instead

- No quick path

Enlightened I/O



Source: Microsoft © TechNet

Virtualization Infrastructure Driver

The VID is the main “glue” in the kernel responsible for connecting the Virtual Machine Management Services (VMMS) with the Hyper-V Kernel

- Lives in a driver called Vid.sys

VID uses the Hypercall interface in order to send management commands to the hypervisor, such as

- Partition suspend/resume
- VP add/remove and policies
- Dynamic memory
- Partition create/delete
- Device visibility

VID is also in charge of MMIO emulation for HAL-type drivers

- Also emulates ROMs

User-mode side Vid.dll loads in Vmms.exe/Vmwp.exe and uses IOCTLs

CPU Management

Root partition will assign certain logical processors to certain children partitions

- These processors are now called VPs or *virtual processors*

The same logical processor on the root partition can be shared among multiple children partitions

- A scheduler determines and distributes workloads across the different partitions

The root partition can install *intercepts* associated to certain events on the children partitions

- I/O Port Access, MSR Access, Exceptions, CPUID, GPA Access

Hyper-V will suspend the virtual processor and send the root partition a message

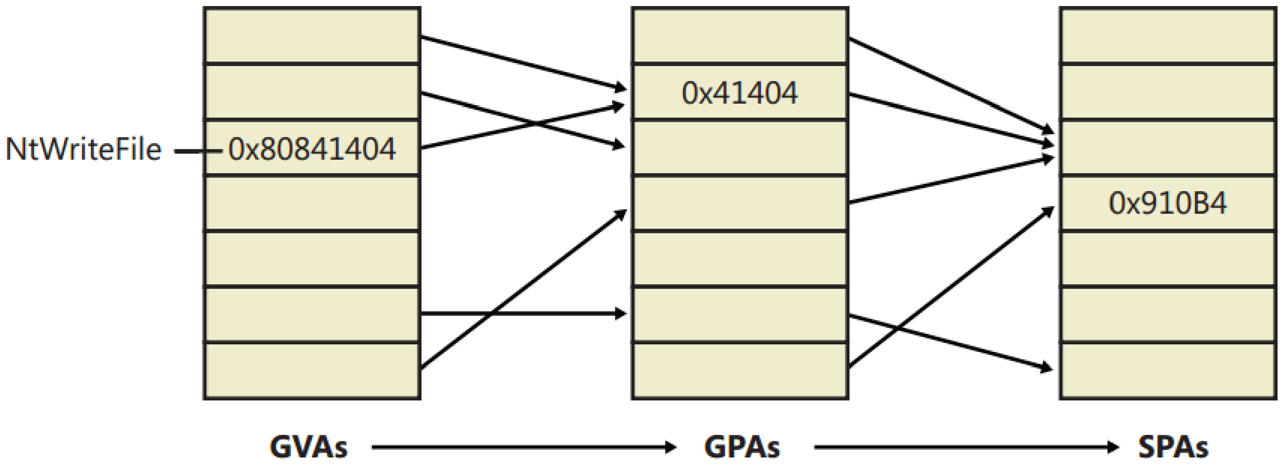
Root partition processes the message and must resume the VP

Memory Management

Memory in Hyper-V is separated into three types

- GVA – Guest Virtual Address (VA in the Child Partition)
- GPA – Guest Physical Address (PA in the Child Partition)
- SPA – System Physical Address (PA in the Root Partition)

Using SLAT/Nested Page Tables, Hyper-V can map GVA->SPA



GPA Space

On the root partition, GPA and SPA are identity mapped

- Pages cannot be unmapped, but different access rights can be set

GPA can be in 3 states

- Mapped (GPA->SPA mapping exists)
- Unmapped (GPA->SPA mapping does not exist)
- Inaccessible (GPA is not valid for access)

Most unmapped GPA pages in the root partition are accessible

- Provides ability to access MMIO devices
- But some are not – for example, Local APIC is owned by the Hyper-V kernel

Unmapped GPA access causes message to root partition

Overlay GPA is used for “virtual” data structures such as hypercall page and statistics page, which are shared among all partitions

Virtual Interrupt Control

The hypervisor virtualizes the local APIC and extends it into a Synthetic Interrupt Controller (SynIC)

External and Internal Interrupts delivered to VPs are virtualized using 16 local vector tables known as SINT0-SINT15 (SINTx)

Internal interrupts are generated when a VP writes to the APIC Interrupt Command Register (ICR)

External interrupts are generated when a physical device generates an interrupt or the hypervisor has to deliver an internal timer or trace message

SynIC is also involved in inter-partition communication when using

- Message Flags
- Messages
- Monitored Notifications

Hypercall Interface

Allows code running under Hyper-V to call into the Hyper-V Kernel

- Either for management tasks, such as creating new partitions, getting tracing, debugging, and statistics information, installing intercepts, …
- Or for inter-partition communication using ports
- Or for providing enlightenments for the guest OS
  - Such as optimizing TLB Flushing or Spinlock Waiting

Hypercalls can be “simple” or “repeat”

- Simple calls perform a single operation with fixed-size input
- Rep calls perform repeated operations based on a starting index and count

Three calling conventions:

- Pass arguments in in/out data structure
- Pass arguments in x64 integer registers
- Pass arguments in x64 XMM vector registers

Hypercalls

Hypercalls can only be done by code running at Ring 0

- Internally, *vmcall* instruction is used, but Hyper-V kernel will generate #UD exception if CPL is not 0

Hypercalls return HV_STATUS return values, which are documented

- RDX:RAX used on x64

Hypercalls must return within 50 microseconds back to the partition

- Rep calls cannot guarantee this, therefore use *hypercall continuations* to resume execution after timeout
- Simple calls almost always complete in time, other than a few exceptions

Input and output must be sent in aligned GPA addresses that do not straddle over a page

Microsoft provides a C interface to hypercalls, abstracting these details

Programming With Hyper-V

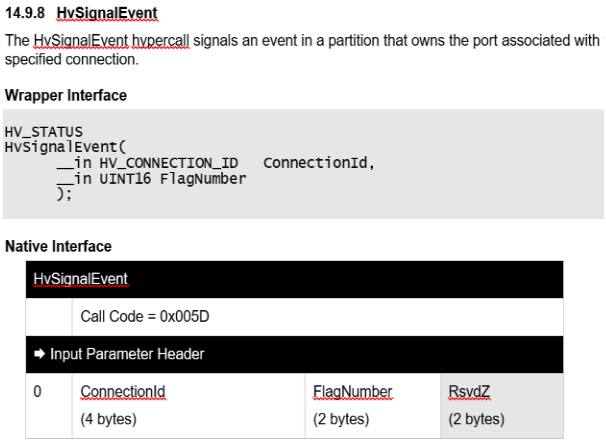
> *"There's no published way to do this and no support from Microsoft.*
>
> *[…] So your system will most likely crash […]"*
>
> Jake Oshins
> Hyper-V I/O Architect
> Windows Kernel Group

Programming With Hyper-V

Hyper-V Documentation

Microsoft has actually done an outstanding job documenting the Hyper-V hypervisor and related infrastructure

Hyper-V has better documentation than the kernel



14.9.8 HvSignalEvent

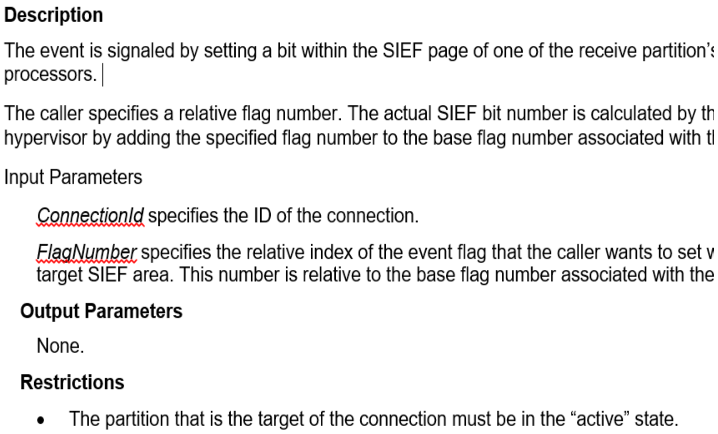
The HvSignalEvent hypercall signals an event in a partition that owns the port associated with specified connection.

Wrapper Interface

```
HV_STATUS
HvSignalEvent(
      __in HV_CONNECTION_ID   ConnectionId,
      __in UINT16 FlagNumber
      );
```

Native Interface

HvSignalEvent			
	Call Code = 0x005D		
➡ Input Parameter Header			
0	ConnectionId (4 bytes)	FlagNumber (2 bytes)	RsvdZ (2 bytes)



Description

The event is signaled by setting a bit within the SIEF page of one of the receive partition's processors.

The caller specifies a relative flag number. The actual SIEF bit number is calculated by th hypervisor by adding the specified flag number to the base flag number associated with tl

Input Parameters

ConnectionId specifies the ID of the connection.

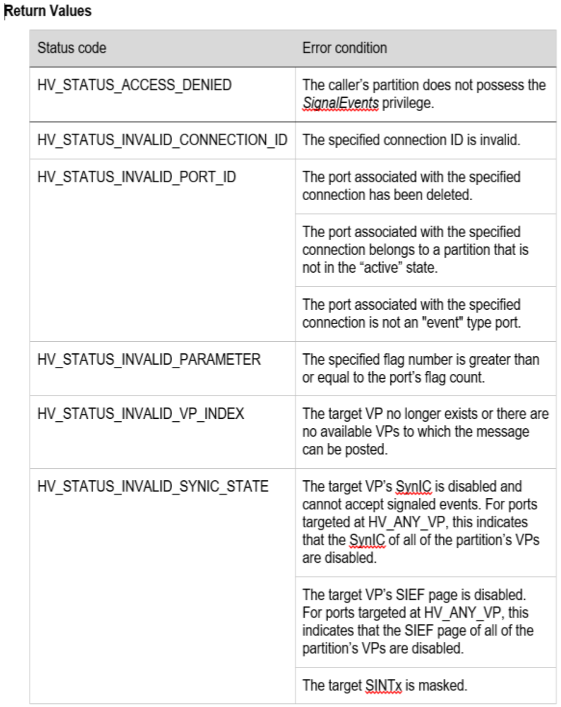
FlagNumber specifies the relative index of the event flag that the caller wants to set v target SIEF area. This number is relative to the base flag number associated with the

Output Parameters

None.

Restrictions

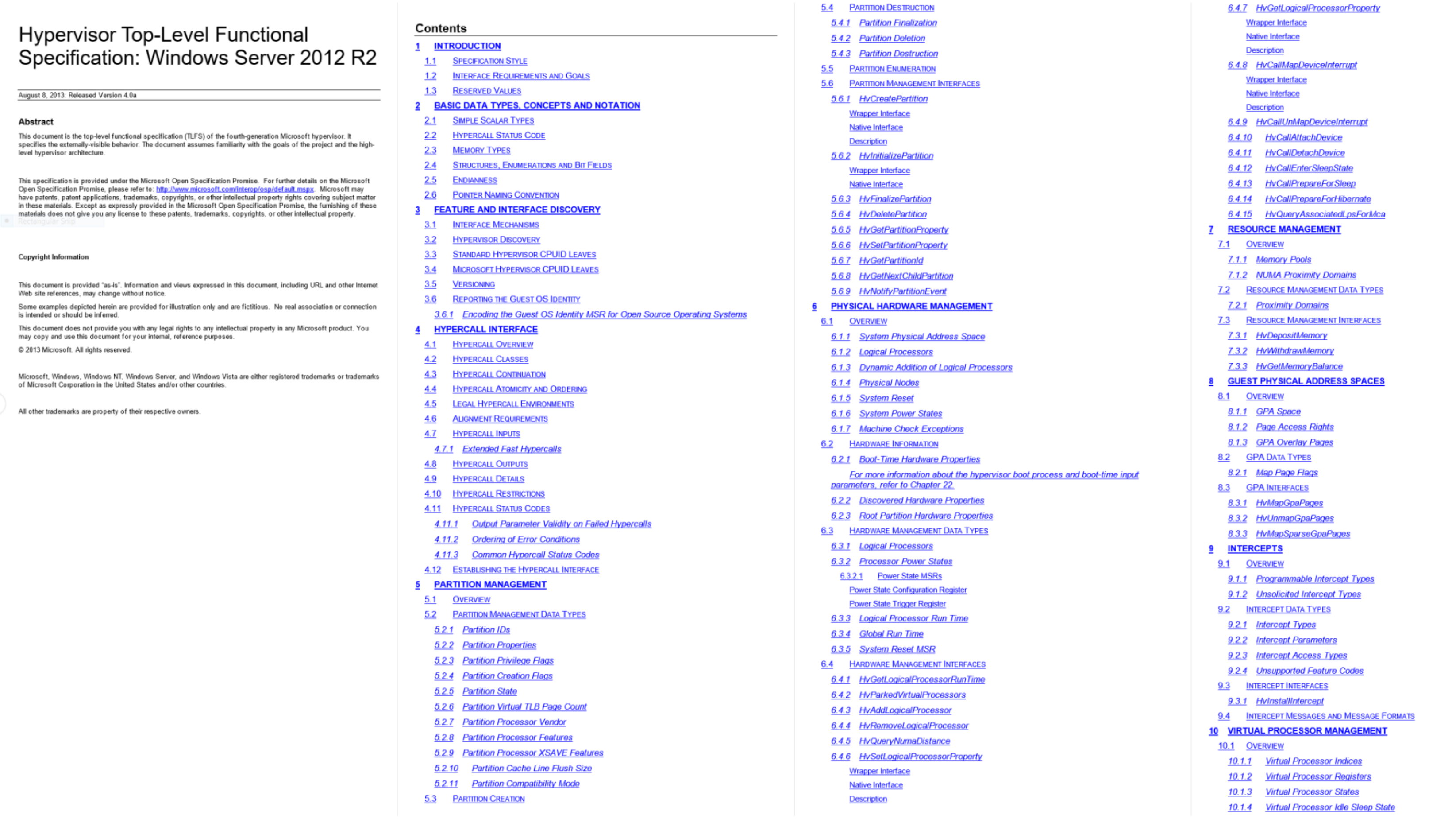
- The partition that is the target of the connection must be in the "active" state.



Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller's partition does not possess the *SignalEvents* privilege.
HV_STATUS_INVALID_CONNECTION_ID	The specified connection ID is invalid.
HV_STATUS_INVALID_PORT_ID	The port associated with the specified connection has been deleted.
	The port associated with the specified connection belongs to a partition that is not in the "active" state.
	The port associated with the specified connection is not an "event" type port.
HV_STATUS_INVALID_PARAMETER	The specified flag number is greater than or equal to the port's flag count.
HV_STATUS_INVALID_VP_INDEX	The target VP no longer exists or there are no available VPs to which the message can be posted.
HV_STATUS_INVALID_SYNIC_STATE	The target VP's SynIC is disabled and cannot accept signaled events. For ports targeted at HV_ANY_VP, this indicates that the SynIC of all of the partition's VPs are disabled.
	The target VP's SIEF page is disabled. For ports targeted at HV_ANY_VP, this indicates that the SIEF page of all of the partition's VPs are disabled.
	The target SINTx is masked.

Top Level Functional Specification



Hypervisor Top-Level Functional Specification: Windows Server 2012 R2

August 8, 2013: Released Version 4.0a

Abstract

This document is the top-level functional specification (TLFS) of the fourth-generation Microsoft hypervisor. It specifies the externally-visible behavior. The document assumes familiarity with the goals of the project and the high-level hypervisor architecture.

Copyright Information

Contents

TLFS 4.0a

Latest version released last year covers Windows 8.1 and Server 2012 R2

TLFS describes internals of the hypervisor, as well as handling of virtual memory, scheduling, IPC, event logging, debugging, processor management, intercepts, and more

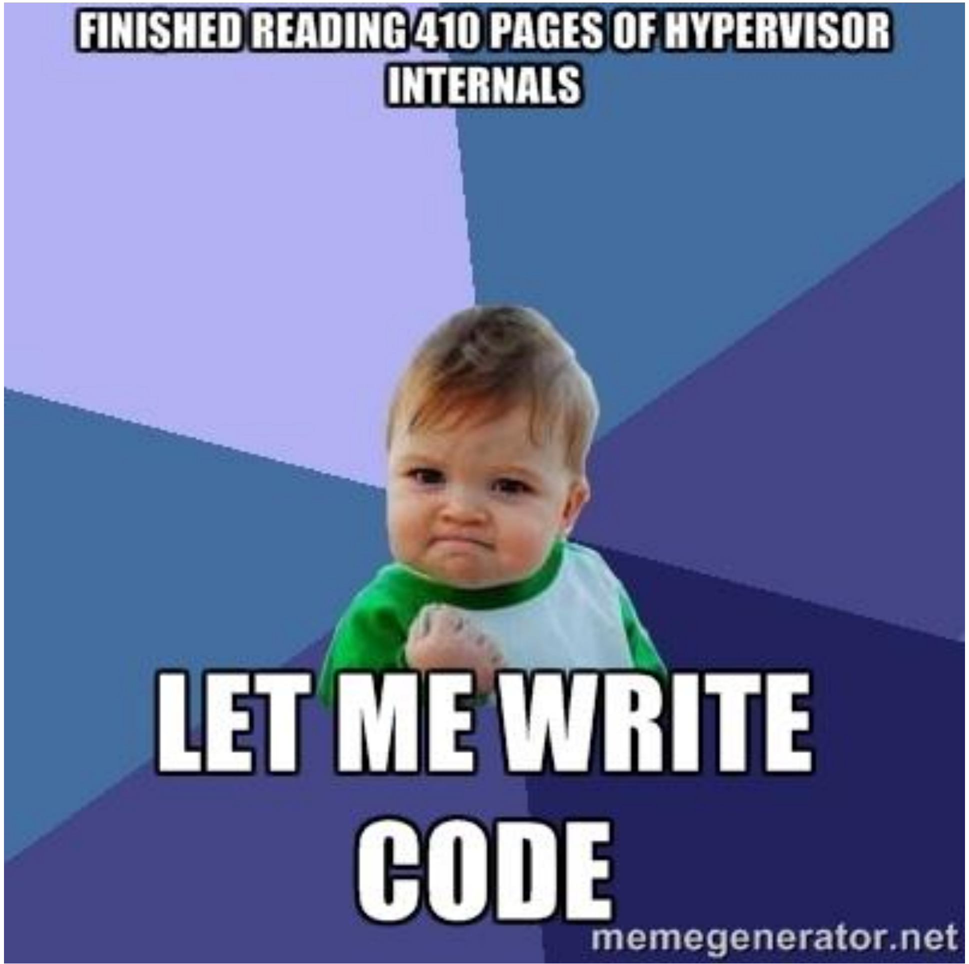
(Almost) all hypercalls are documented, with full parameter and structure definitions

Designed to allow 3rd party OS vendors to interoperate with the hypervisor and create their own “enlightenments” for faster virtualization

Also covers CPUID and MSRs specific to the hypervisor

http://download.microsoft.com/download/A/B/4/AB43A34E-BDD0-4FA6-BDEF-79EEF16E880B/Hypervisor%20Top%20Level%20Functional%20Specification%20v4.0.docx

More like TL;DR 4.0a



Hyper-V Development

To develop for Hyper-V, one needs three key pieces:

- Hypervisor Guest Development Kit (HVGDK.H) and HyperCall Headers (WINHV.H)
  - HVGDK is the entire technical specification provided in header format
  - WinHv headers provide an interface to the hypercalls by using standard Windows Driver API
- Virtualization Infrastructure Driver Headers (VID.H, VIDDEFS.H)
  - Allows registration of intercepts, partition management, and state transition resilience
- Import Libraries (WINHV.LIB, VID.LIB)
  - Allows linking with the WinHv driver to access the hypercall interface, and with Vid.sys/Vid.dll

Vista Windows Driver Kit (Build 6000) ships with the HVGDK/WINHV headers

- But not the libraries to link with!

VID is considered undocumented and not meant to be interfaced with

- Because of this, even with the HVGDK, programming the hypervisor is potentially dangerous, as there is no notification/management for sleep/resume/migration operations that can happen to a partition

Missing Libraries

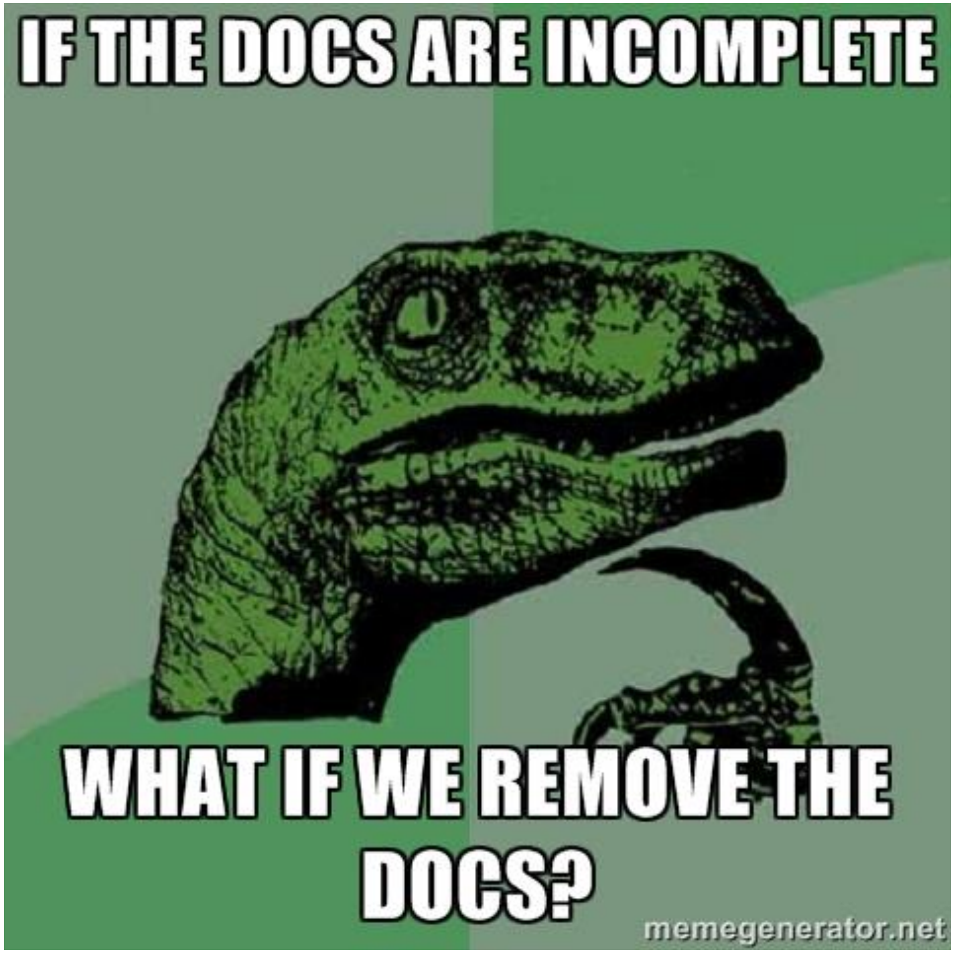
Wannabe Hyper-V Developer:

> *"This is remarkably silly. Microsoft has published the interfaces but not the .lib file and developers with good intentions are off doing miserable hackery in order to get their stuff working, as they have no other choice."*

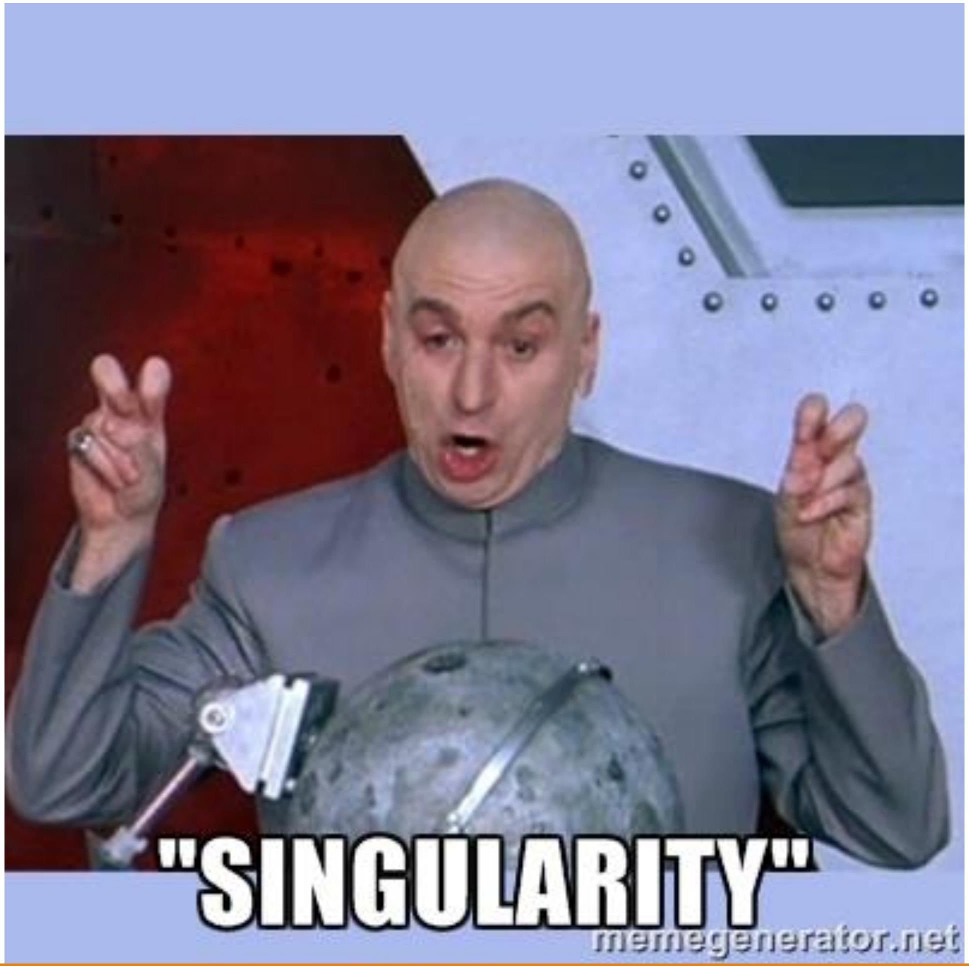
Hyper-V Architect:

> *"Yup. I agree completely. The published interfaces will disappear in the next documentation drop. Since there is not really anything you could build with them that will work end-to-end, publishing them doesn't help anyone."*

HVGDK Removed in WDK 7+



But still present in Singularity…



Obtaining the Required Bits

For the HVGDK, you can use Singularity:

- https://singularity.svn.codeplex.com/svn/base/Windows/Inc/hvgdk.h
- Vid headers are also there (but we won’t cover Vid programming now)

However, this is becoming out-of-date. The more recent (but differently factored) headers are available in the Linux Integration Services

- https://github.com/LIS/LIS3.5
  - HvStatus.h, HvTypes.h, Hv.h, HvHcApi.h HvVpApi.h, etc…

For WinHV Headers, you’ll need to find an old version of the WDK

- May be available on MSDN

How to get the Hypercall Library? Make your own!

- Use dumpbin to dump exports to .def file
- Link a .lib file based on the .def file
- Won’t work on x86 due to decorations: you’ll need a stub .c file to compile

Changes in Windows 7+

The virtualization stack in Windows 7 has been split to separate "root" partition stack components from "child" components

- For example, VMBus is now VMBus.sys and VMBusr.sys
- The same has happened to WinHV: WinHvr.sys, WinHv.sys

This means you can no longer create a single driver that auto-detects the type of partition it's in

- Can't link to two separate import libraries with functions having the same name
- You'll need winhvr.lib and winhv.lib, and separate drivers

Also, be wary of API changes between one version of WinHV and the next

- Windows 7 Port APIs now have a NUMA Node Requirement parameter
- Windows 10 SynIC APIs now expect a group affinity (GROUP_AFFINITY)

Inter-Partition Communication

Events

- Represented as a single bit flag
- Array of 2048 bit flags is provided for each SynIC, covering 1/16th of the SIEF Page
- Sender *signals* an event, which ORs the bit and sends an interrupt if not cleared

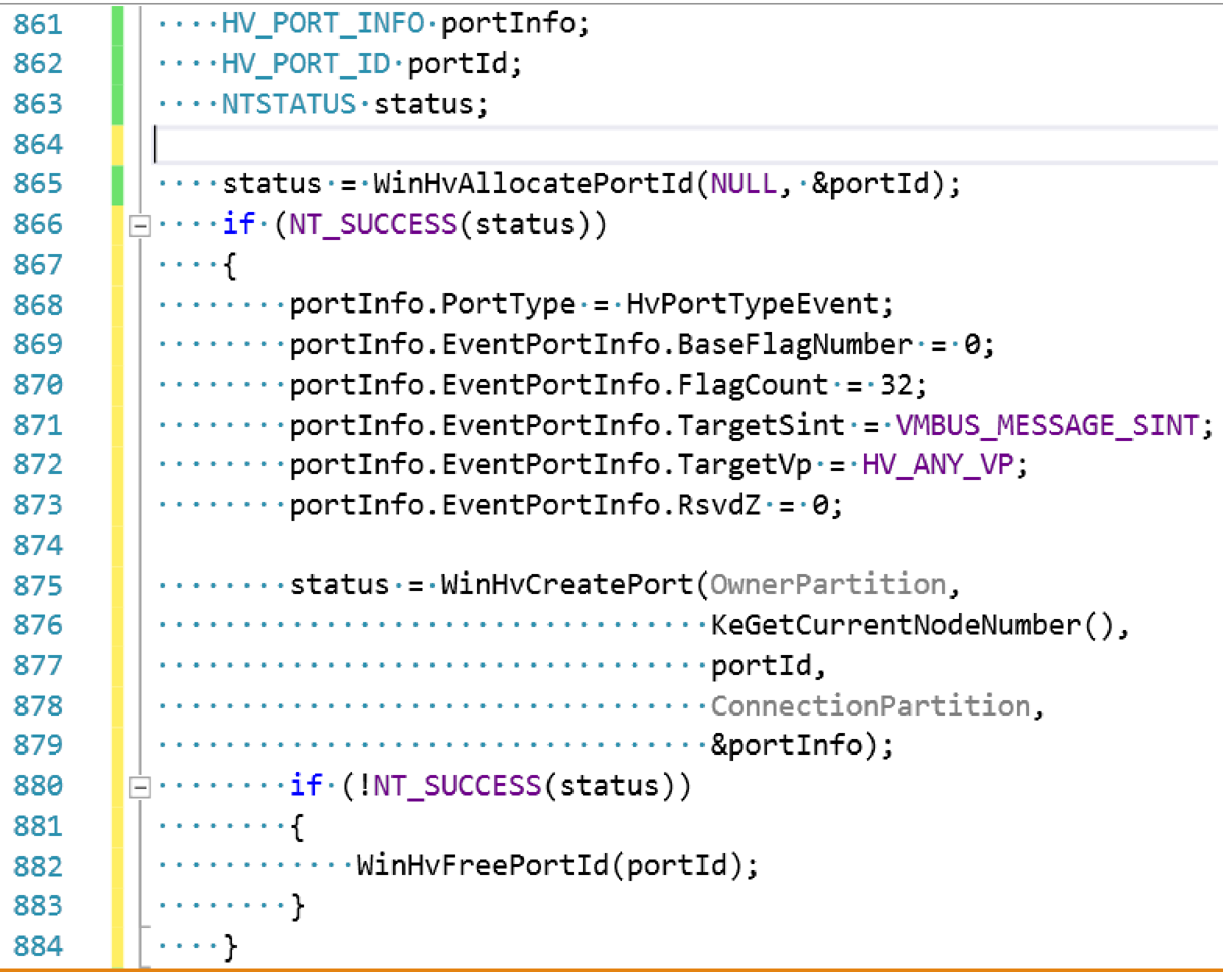
Messages

- Represented as an arbitrary byte array of up to 256 bytes
- Array of 256 byte buffers is provided for each SynIC, covering 1/16th of SIM Page
- Sender *posts* a message, which is added to a queue and sends an interrupt
  - Sender acknowledges by writing to End-Of-Message (EOM) MSR or by APIC End-Of-Interrupt (EOI)
- Hypervisor will re-deliver message if not handled within a few milliseconds
- Guaranteed ordering <u>if all delivery is within the same VP</u>

Monitored Notifications

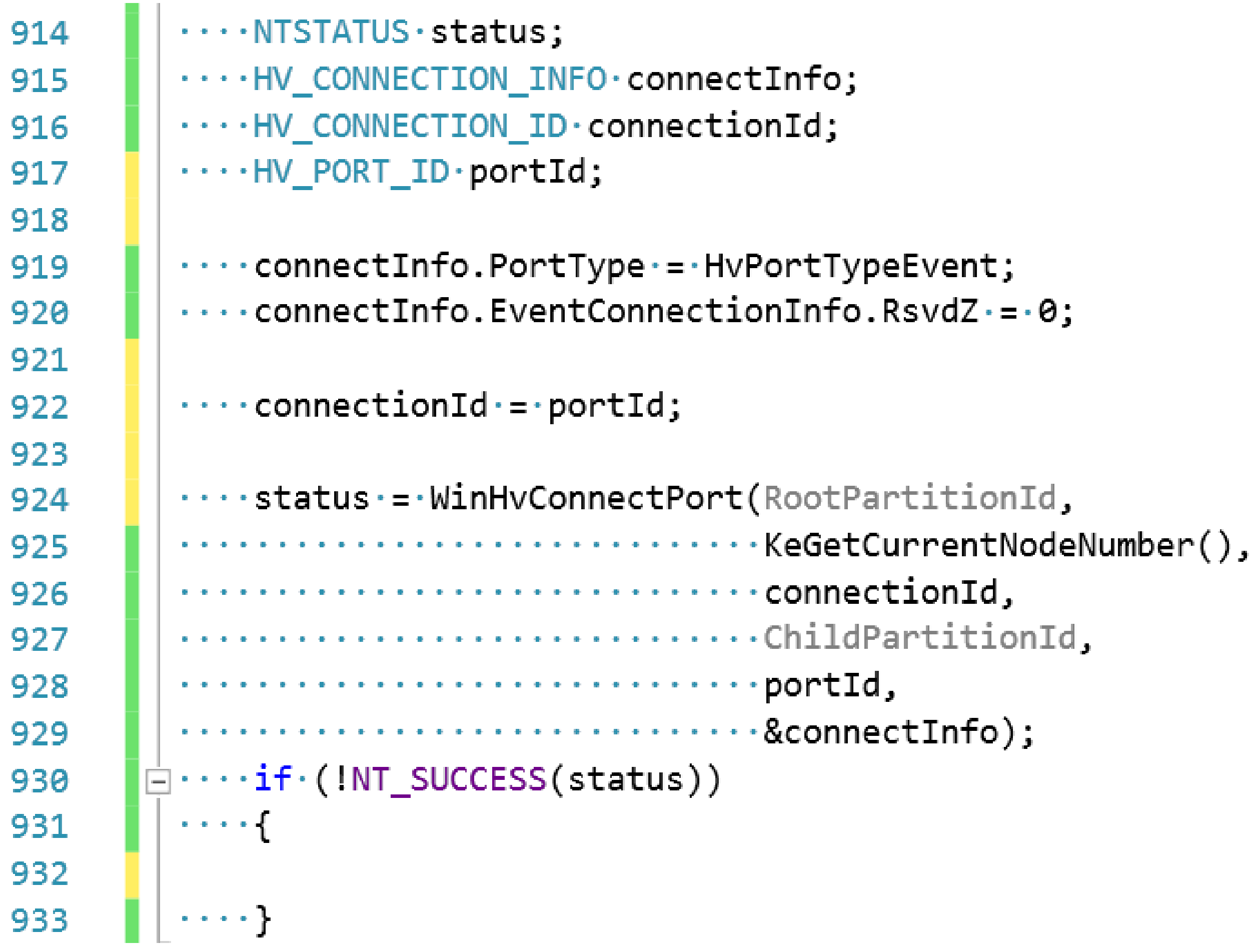
- See TL;DR 4.1

Creating a Port



```
    HV_PORT_INFO portInfo;
    HV_PORT_ID portId;
    NTSTATUS status;

    status = WinHvAllocatePortId(NULL, &portId);
    if (NT_SUCCESS(status))
    {
        portInfo.PortType = HvPortTypeEvent;
        portInfo.EventPortInfo.BaseFlagNumber = 0;
        portInfo.EventPortInfo.FlagCount = 32;
        portInfo.EventPortInfo.TargetSint = VMBUS_MESSAGE_SINT;
        portInfo.EventPortInfo.TargetVp = HV_ANY_VP;
        portInfo.EventPortInfo.RsvdZ = 0;

        status = WinHvCreatePort(OwnerPartition,
                                 KeGetCurrentNodeNumber(),
                                 portId,
                                 ConnectionPartition,
                                 &portInfo);
        if (!NT_SUCCESS(status))
        {
            WinHvFreePortId(portId);
        }
    }
```

Connecting to a Port



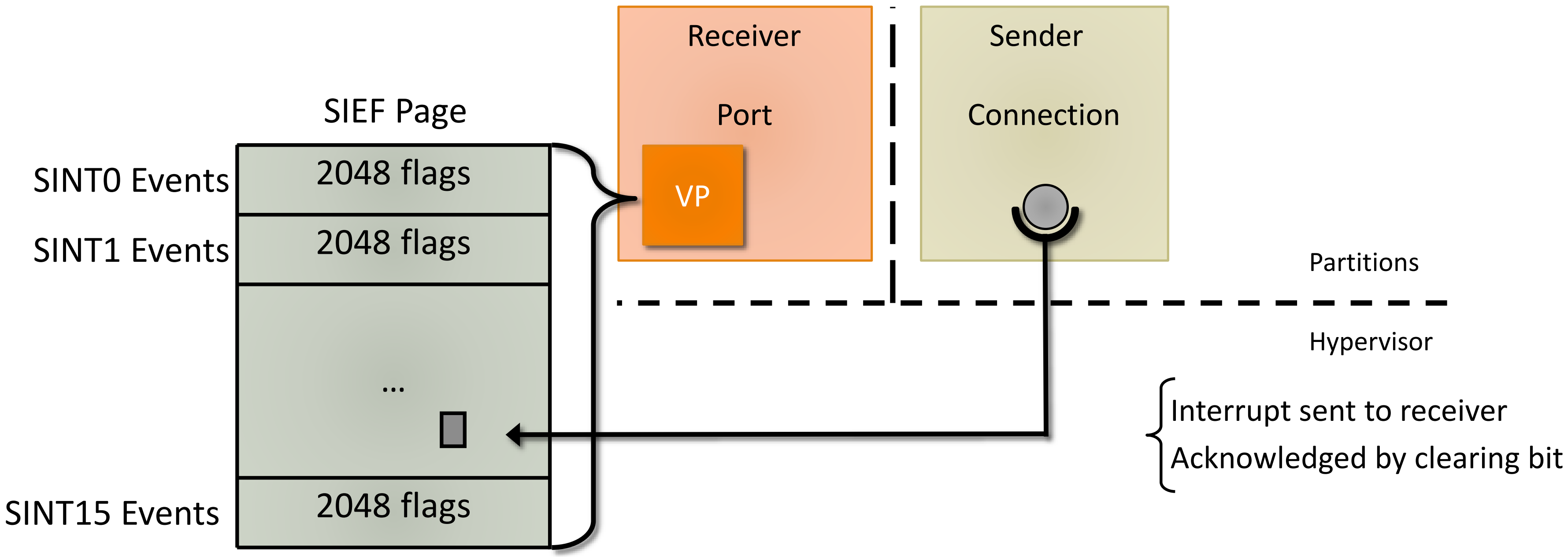
```
    NTSTATUS status;
    HV_CONNECTION_INFO connectInfo;
    HV_CONNECTION_ID connectionId;
    HV_PORT_ID portId;

    connectInfo.PortType = HvPortTypeEvent;
    connectInfo.EventConnectionInfo.RsvdZ = 0;

    connectionId = portId;

    status = WinHvConnectPort(RootPartitionId,
                              KeGetCurrentNodeNumber(),
                              connectionId,
                              ChildPartitionId,
                              portId,
                              &connectInfo);
    if (!NT_SUCCESS(status))
    {

    }
```

Sending and Receiving Events

WinHV maps the per-VP SynIC event page using an overlay page

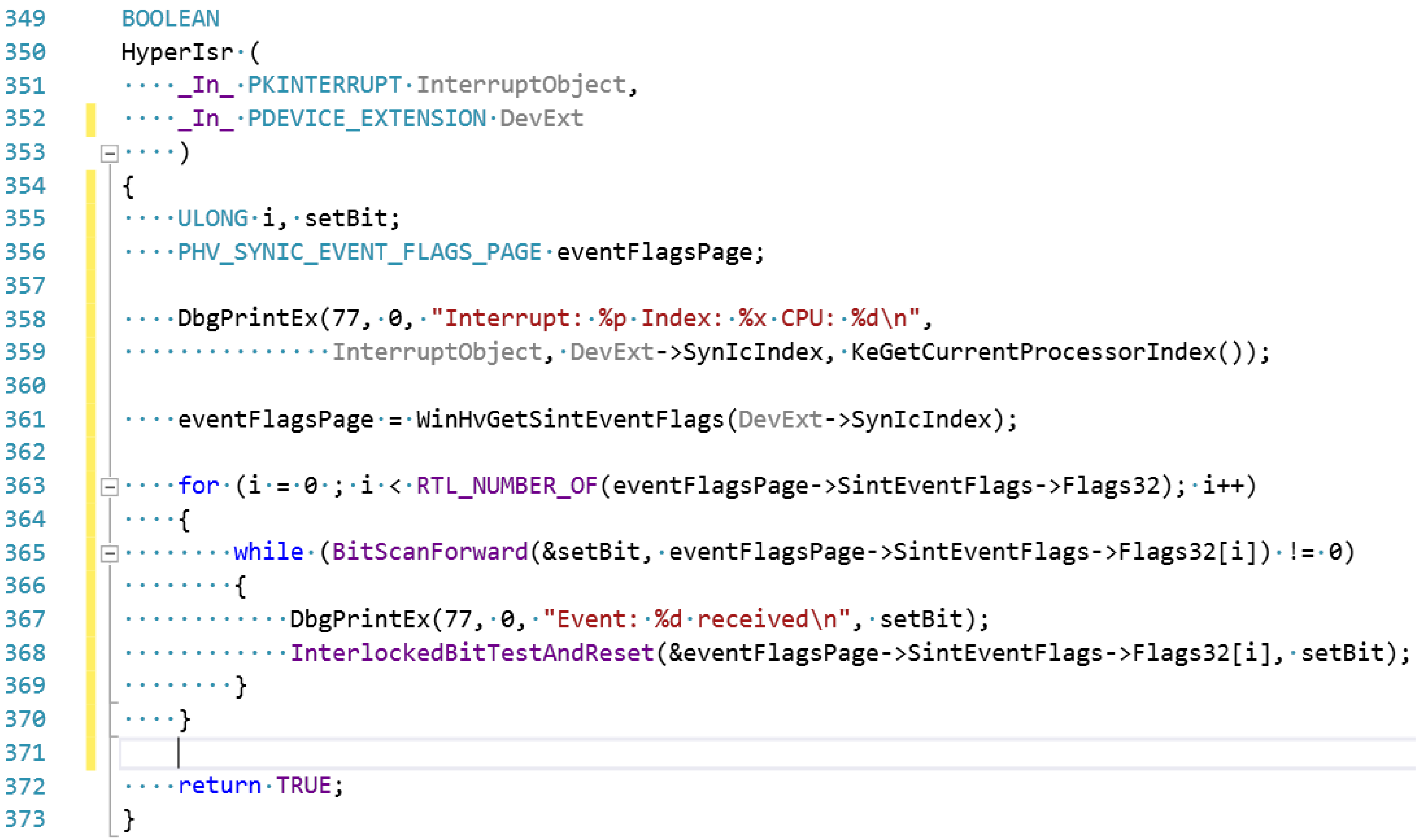
Sender uses *HvSignalEvent*, which sets corresponding flag bit

Receiver calls WinHV to receive address of flags for given SynIC



Interrupt sent to receiver
Acknowledged by clearing bit

Receiver Handling an Event



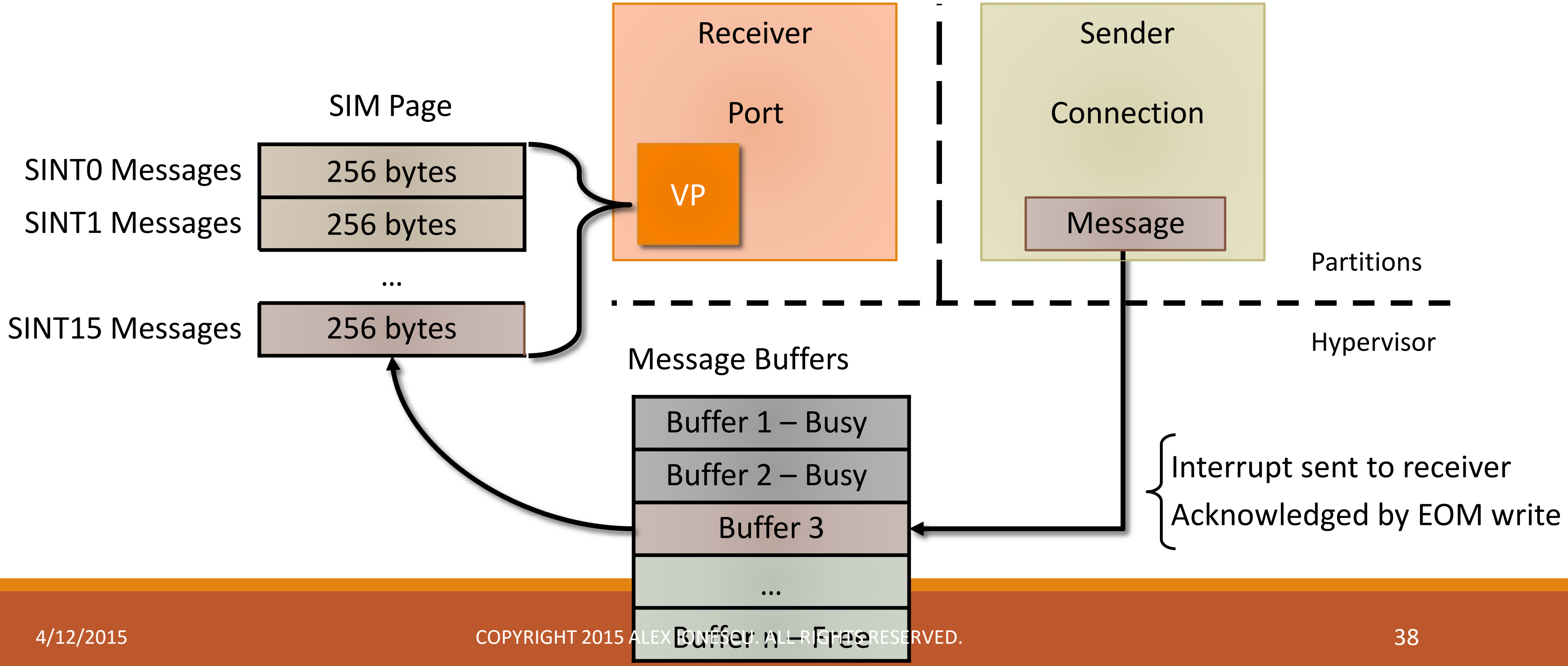
```
BOOLEAN
HyperIsr (
    _In_ PKINTERRUPT InterruptObject,
    _In_ PDEVICE_EXTENSION DevExt
    )
{
    ULONG i, setBit;
    PHV_SYNIC_EVENT_FLAGS_PAGE eventFlagsPage;

    DbgPrintEx(77, 0, "Interrupt: %p Index: %x CPU: %d\n",
               InterruptObject, DevExt->SynIcIndex, KeGetCurrentProcessorIndex());

    eventFlagsPage = WinHvGetSintEventFlags(DevExt->SynIcIndex);

    for (i = 0 ; i < RTL_NUMBER_OF(eventFlagsPage->SintEventFlags->Flags32); i++)
    {
        while (BitScanForward(&setBit, eventFlagsPage->SintEventFlags->Flags32[i]) != 0)
        {
            DbgPrintEx(77, 0, "Event: %d received\n", setBit);
            InterlockedBitTestAndReset(&eventFlagsPage->SintEventFlags->Flags32[i], setBit);
        }
    }

    return TRUE;
}
```

Sending Messages

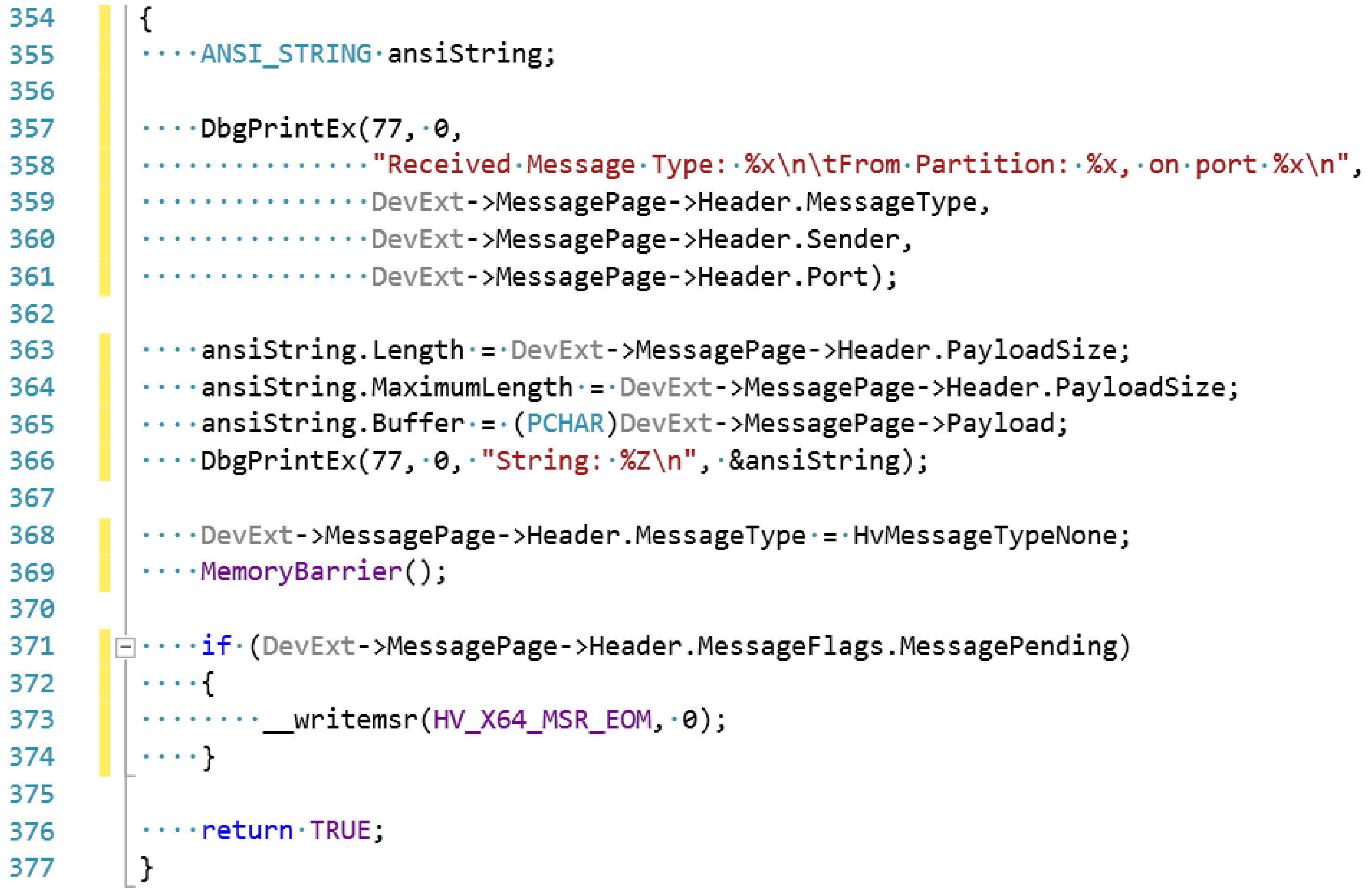
WinHV maps the per-VP SynIC message page with an overlay

Sender uses *HvPostMessage,* which copies into a per-receiver buffer and queues the message

Message header contains type, payload size, port ID and connection ID



Receiver Handling a Message



```
{
    ANSI_STRING ansiString;

    DbgPrintEx(77, 0,
               "Received Message Type: %x\n\tFrom Partition: %x, on port %x\n",
               DevExt->MessagePage->Header.MessageType,
               DevExt->MessagePage->Header.Sender,
               DevExt->MessagePage->Header.Port);

    ansiString.Length = DevExt->MessagePage->Header.PayloadSize;
    ansiString.MaximumLength = DevExt->MessagePage->Header.PayloadSize;
    ansiString.Buffer = (PCHAR)DevExt->MessagePage->Payload;
    DbgPrintEx(77, 0, "String: %Z\n", &ansiString);

    DevExt->MessagePage->Header.MessageType = HvMessageTypeNone;
    MemoryBarrier();

    if (DevExt->MessagePage->Header.MessageFlags.MessagePending)
    {
        __writemsr(HV_X64_MSR_EOM, 0);
    }

    return TRUE;
}
```

Receiving Hyper-V Interrupts

Windows PnP Driver Development

Two Types of Windows Drivers

When people write windows drivers, they are generally of two types:

- Legacy non-Plug-and-Play Drivers (also called kernel modules)
- Hardware Plug-and-Play Drivers (also called WDM drivers, or WDF drivers)

Although both of these run in the kernel and have full Ring 0 rights, there are important internal differences in how they are handled

- PnP Drivers are expected to handle certain I/O operations from the kernel's Plug and Play Manager (called PIRPs)
- PnP Drivers receive a *bus-enumerated device node* (DEVICE_NODE) structure
  - This ties them to hardware and the bus specific enumeration protocol
- PnP Drivers are allowed to request, filter, and translate resources
  - They can register and handle interrupts
  - They can create DMA Adapter Objects and perform DMA operations on the system
- PnP Drivers have access to additional Windows Kernel APIs

PnP-Driver-Only Windows APIs

IoReportTargetDeviceChange(Asynchronous)

Io(Get/Register)DeviceInteface

IoOpenDeviceRegistryKey

Io(Synchronous)InvalidateDeviceRelations

IoRequestDeviceEject(Ex)

IoInvalidateDeviceState

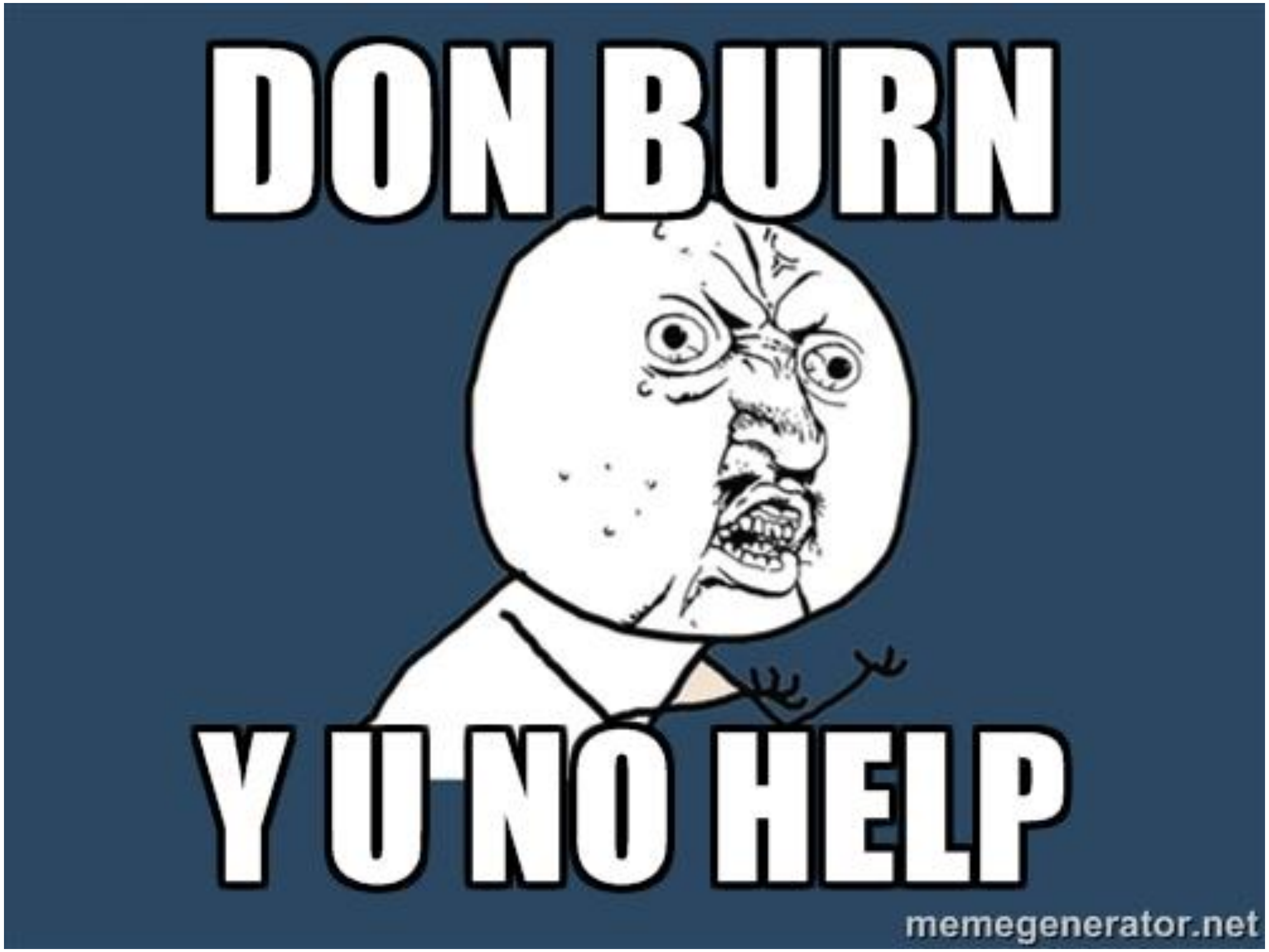
IoGetDmaAdapter

IoGetDeviceInstanceName

Io(Get/Set)DevicePropertyData

IoConnectInterrupt

Getting Started with Drivers…



What makes you PnP?

Internally, what the kernel really considers as a PnP Driver is a driver that has a device node (and without the DNF_LEGACY_RESOURCE_DEVICENODE flag set)

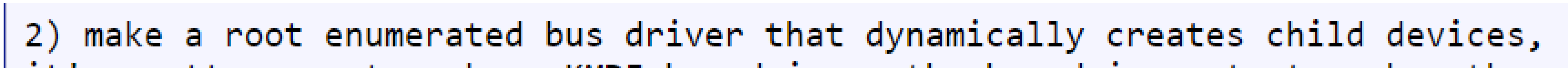
- This type of driver is called a “PDO”, or Physical Device Object
- It directly manages a hardware device

So how can we *become* a PDO?

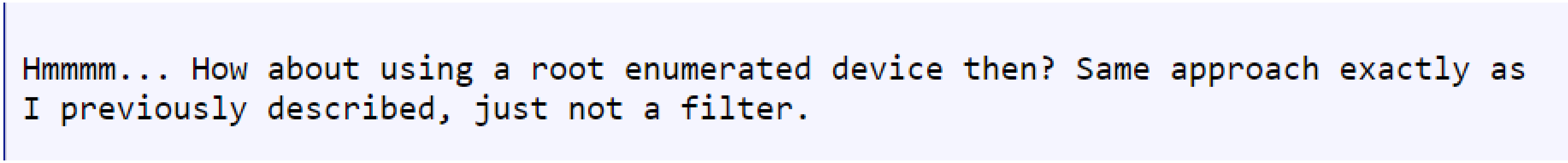
- Clearly this is needed for a virtual device like VMBus which needs to receive interrupts

It turns out that this is a highly guarded process, with a “standard” way of doing things

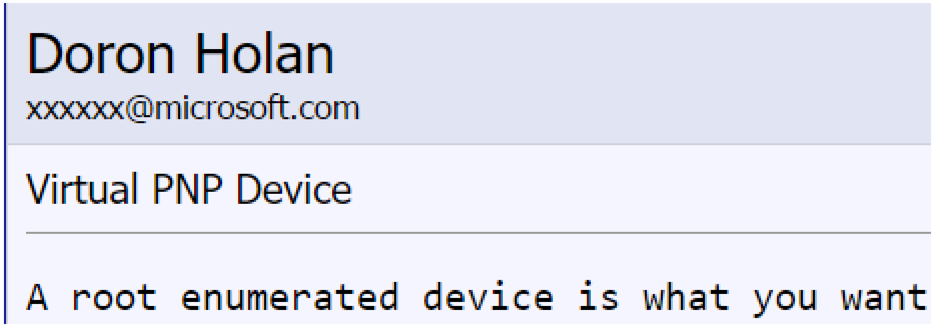
- Causes a very visible contract to exist between user, hardware, and kernel
- Requires writing an INF file, becoming a root bus enumerated driver, etc…



2) make a root enumerated bus driver that dynamically creates child devices,



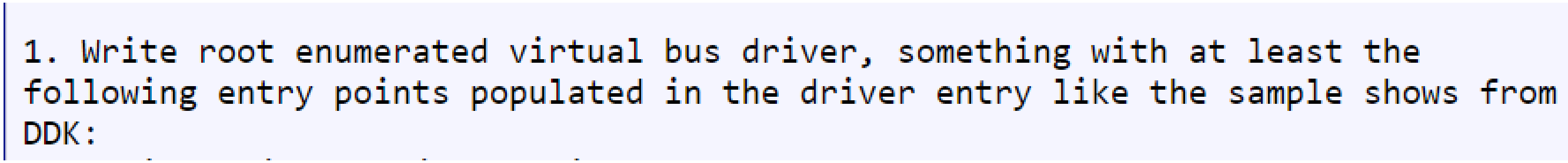
Hmmmm... How about using a root enumerated device then? Same approach exactly as I previously described, just not a filter.



Doron Holan
xxxxxx@microsoft.com

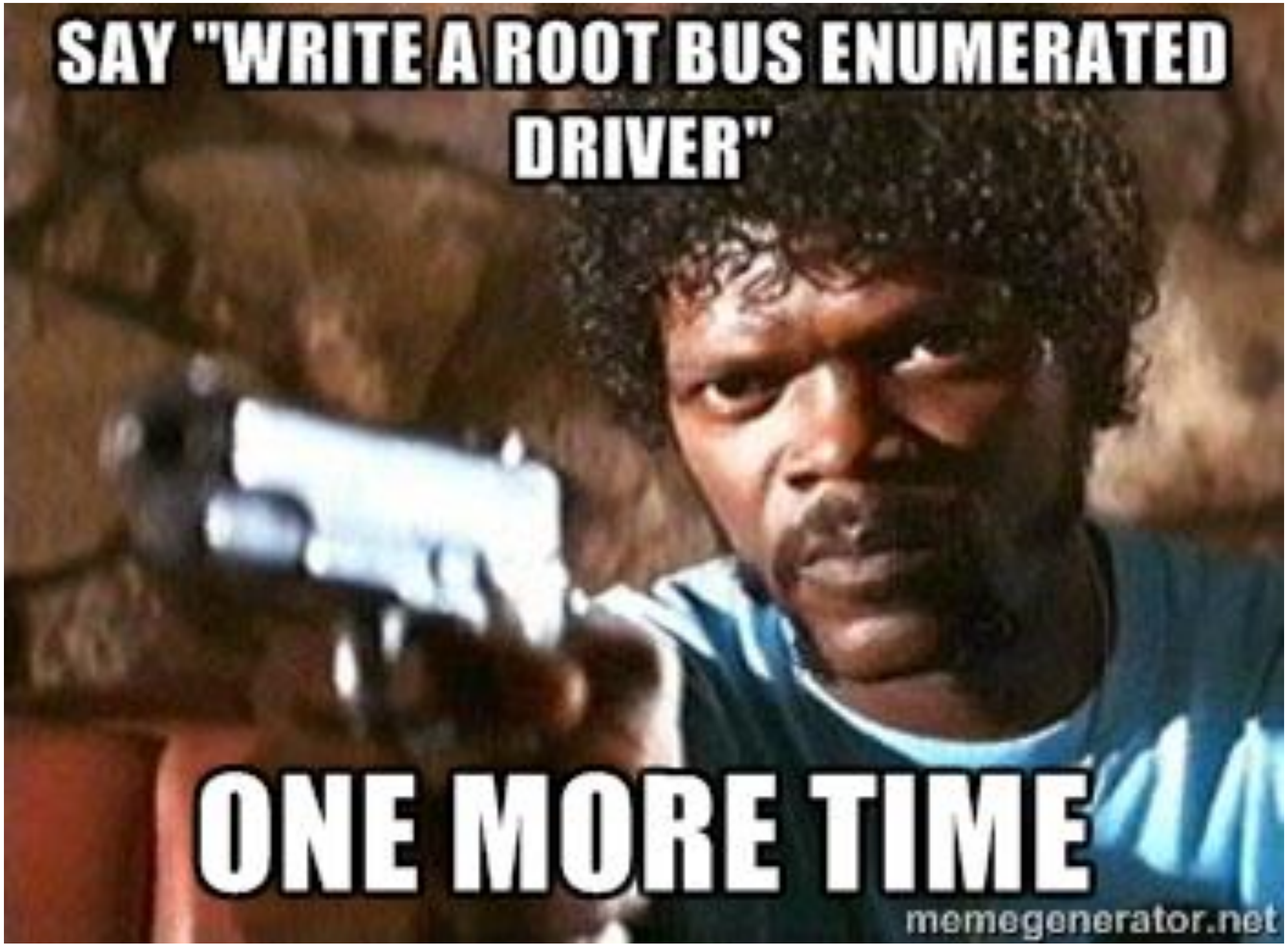
Virtual PNP Device

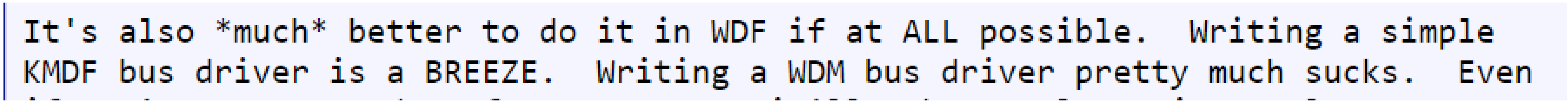
A root enumerated device is what you want



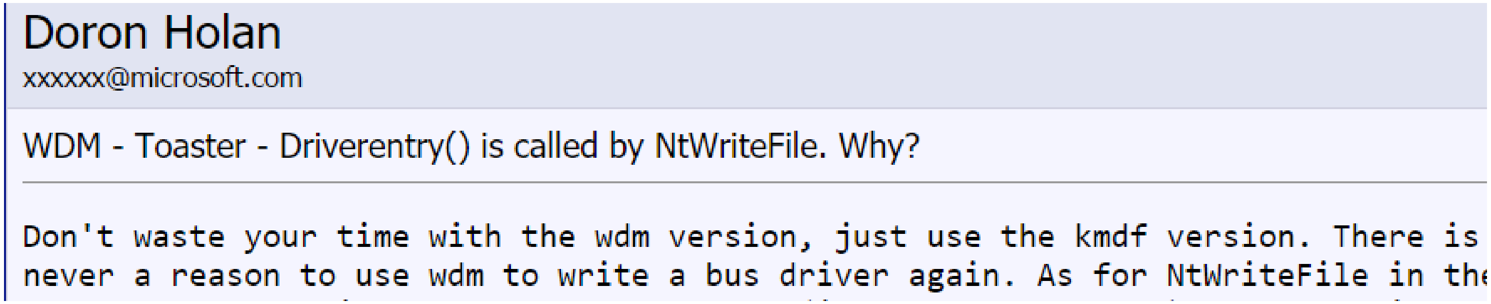
1. Write root enumerated virtual bus driver, something with at least the following entry points populated in the driver entry like the sample shows from DDK:

Root Bus Enumerated Drivers





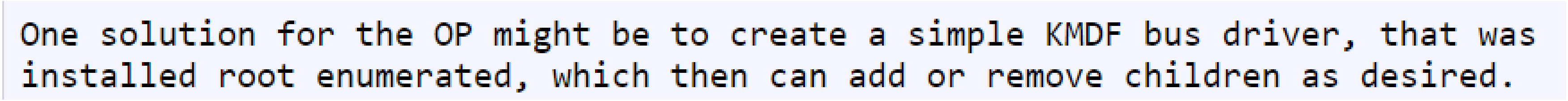
It's also *much* better to do it in WDF if at ALL possible. Writing a simple KMDF bus driver is a BREEZE. Writing a WDM bus driver pretty much sucks. Even



Doron Holan
xxxxxx@microsoft.com

WDM - Toaster - Driverentry() is called by NtWriteFile. Why?

Don't waste your time with the wdm version, just use the kmdf version. There is never a reason to use wdm to write a bus driver again. As for NtWriteFile in the



One solution for the OP might be to create a simple KMDF bus driver, that was installed root enumerated, which then can add or remove children as desired.

Kernel Mode Driver Framework

Writing a “Software” PnP Driver

It turns out that since NT 4 didn’t actually have Plug-n-Play, there was a completely different way of accessing hardware resources

- There were no “PDOs” back then

Each driver ran on its own, scanned the bus, found what it needed, and claimed it from the operating system

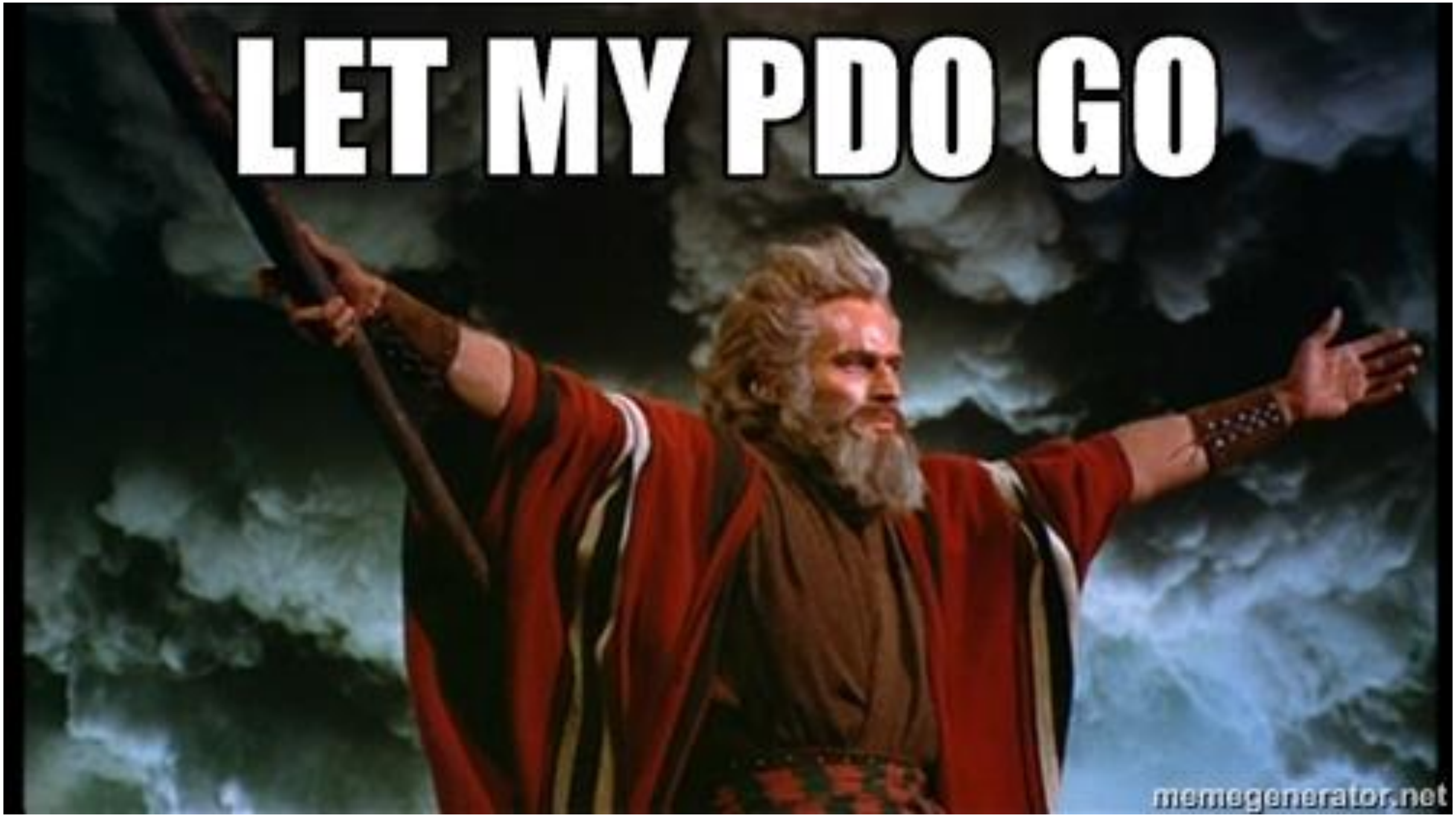
- *IoAssignResources*
- *HalGetInterruptVector*
- Other APIs which are all now marked as “deprecated/legacy/dontuse”

Windows 2000’s WDM/PnP model broke all these legacy drivers since they didn’t have a PDO

Windows introduced some legacy APIs for backward compatibility:

- *IoReportDetectedDevice*
- *IoReportResourceForDetection*

IoReportDetectedDevice



Claiming APIC/MSI Interrupts…

With those APIs, we can now take our software driver, become a PDO, follow all the right PnP procedures, and register our interrupt!

- Not so fast
- In order to get the right to request an interrupt, the IRQ/GSIV resource must be associated with our PDO

We can "fake" an interrupt resource by using the *IoReportResourceForDetection* API…

- But this sets an internal flag and doesn't populate an undocumented registry key in the HARDWARE hive…
- When we later try to connect to the interrupt, the ACPI IRQ Arbiter sees that, and refuses our attempt
  - Because APIC/MSI-X interrupts are modern – why would a "legacy" driver need to claim them?

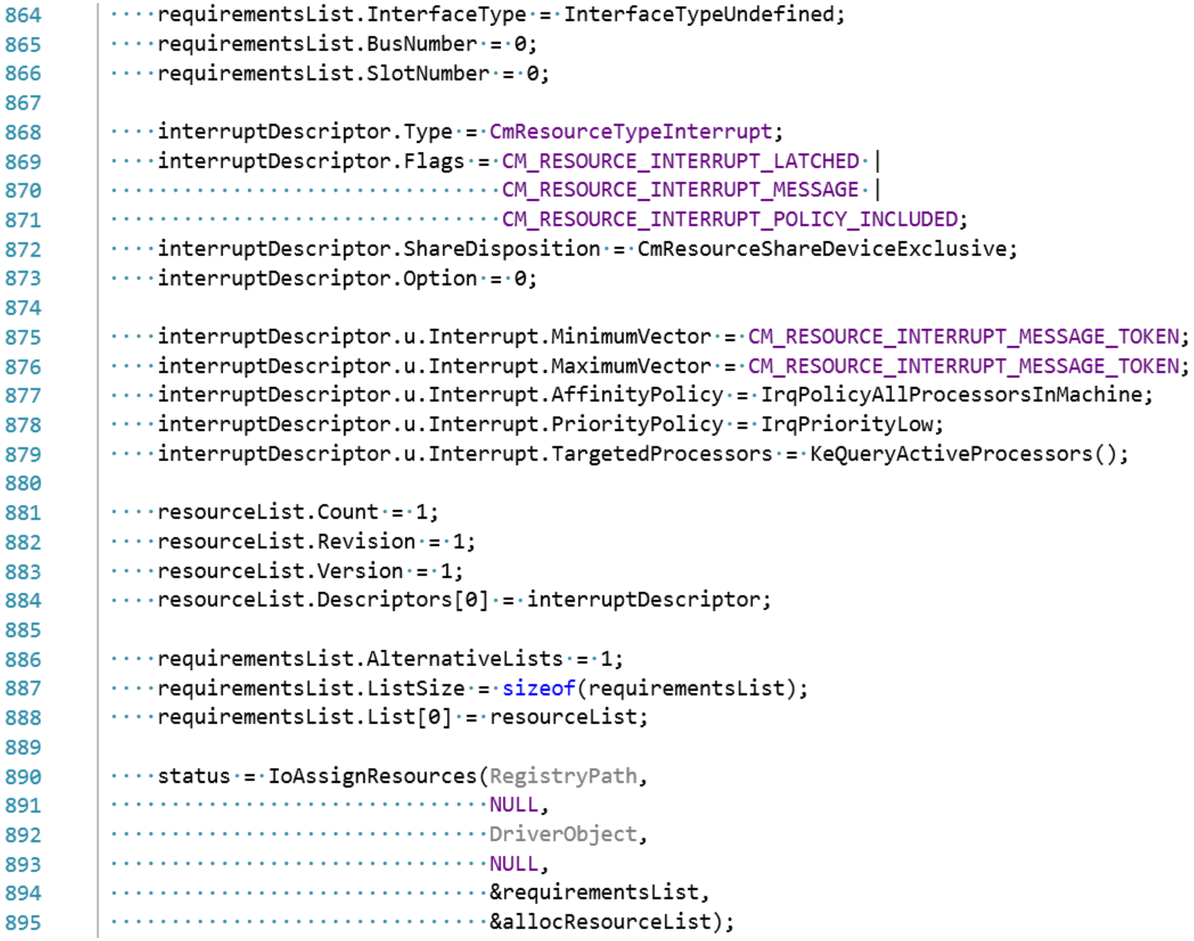
Repeated OSR posts from experts all claim you need an INF file…

- Or a root-bus-enumerated driver using KMDF

INF Files Require Manual Install

Unless… There’s Another Way

IoAssignResources



```
    requirementsList.InterfaceType = InterfaceTypeUndefined;
    requirementsList.BusNumber = 0;
    requirementsList.SlotNumber = 0;

    interruptDescriptor.Type = CmResourceTypeInterrupt;
    interruptDescriptor.Flags = CM_RESOURCE_INTERRUPT_LATCHED |
                                CM_RESOURCE_INTERRUPT_MESSAGE |
                                CM_RESOURCE_INTERRUPT_POLICY_INCLUDED;
    interruptDescriptor.ShareDisposition = CmResourceShareDeviceExclusive;
    interruptDescriptor.Option = 0;

    interruptDescriptor.u.Interrupt.MinimumVector = CM_RESOURCE_INTERRUPT_MESSAGE_TOKEN;
    interruptDescriptor.u.Interrupt.MaximumVector = CM_RESOURCE_INTERRUPT_MESSAGE_TOKEN;
    interruptDescriptor.u.Interrupt.AffinityPolicy = IrqPolicyAllProcessorsInMachine;
    interruptDescriptor.u.Interrupt.PriorityPolicy = IrqPriorityLow;
    interruptDescriptor.u.Interrupt.TargetedProcessors = KeQueryActiveProcessors();

    resourceList.Count = 1;
    resourceList.Revision = 1;
    resourceList.Version = 1;
    resourceList.Descriptors[0] = interruptDescriptor;

    requirementsList.AlternativeLists = 1;
    requirementsList.ListSize = sizeof(requirementsList);
    requirementsList.List[0] = resourceList;

    status = IoAssignResources(RegistryPath,
                               NULL,
                               DriverObject,
                               NULL,
                               &requirementsList,
                               &allocResourceList);
```

Problem with Assigned Resources

Now that you’ve assigned resources, you “own” them and you can go ahead and claim them

But, due to hardware architecture reasons, the resources that a device sees and the underlying hardware resources are not always the same

- Windows implements a “translation” and “arbitration” process to resolve these issues
- The ultimate point of this is that the *IoConnectInterrupt* call expects to receive the final, translated & assigned resources
- While *IoAssignRessources* only provides an intermediate step

Real Plug-and-Play drivers will receive the final copy of these resources in a special PIRP called IRP_MN_START_DEVICE

- But self-reported “fake” PDOs do not receive IRP_MN_START_DEVICE…
- … until at the next reboot!

I want IRP_MN_START_DEVICE

Forcing IRP_MN_START_DEVICE

This last hurdle is solved by reading a wonderful blog post from one of the principal Microsoft developers in the PnP World

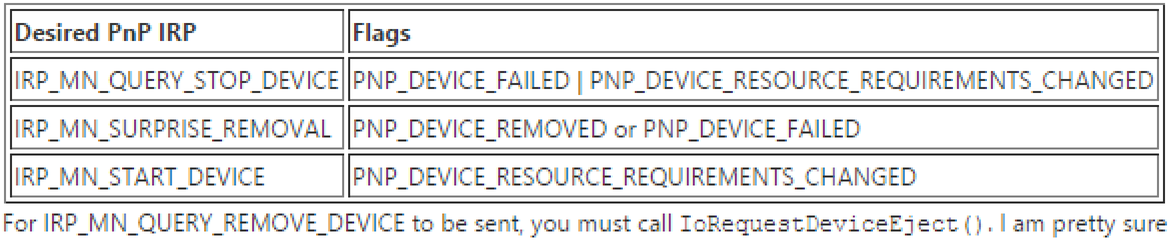
- “How to test PnP state changes in your driver”

Requires usage of the *IoInvalidateDeviceState* API

- Which we can call because we are a PDO now!

This sends another PIRP: IRP_MN_QUERY_PNP_DEVICE_STATE

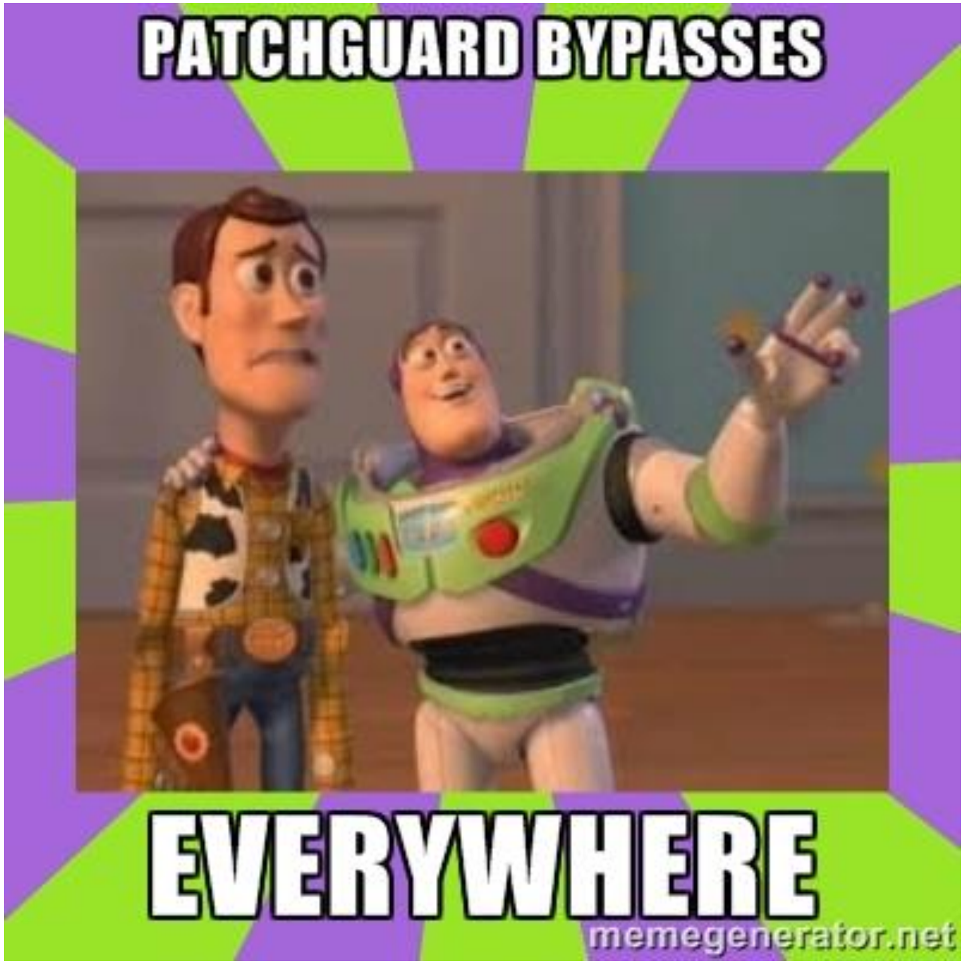
- Our response to this, with certain flags, will force one of the desired PIRPs to occur



Desired PnP IRP	Flags
IRP_MN_QUERY_STOP_DEVICE	PNP_DEVICE_FAILED | PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED
IRP_MN_SURPRISE_REMOVAL	PNP_DEVICE_REMOVED or PNP_DEVICE_FAILED
IRP_MN_START_DEVICE	PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED

For IRP_MN_QUERY_REMOVE_DEVICE to be sent, you must call IoRequestDeviceEject(). I am pretty sure

Fake PDOs can Register Interrupts!



CHILD->ROOT MESSAGE DEMO

ROOT->CHILD EVENT DEMO

I DON'T ALWAYS SIGNAL EVENTS
BUT WHEN I DO, I PICK THE WRONG SINT
memegenerator.net

Interesting Hyper-V Behaviors

Overlay pages are allocated as **executable** and at easy to guess virtual/physical locations

- Significantly increases attack surface risk if there is a bug in message passing, for example
- Fixed in Windows 10 Build 10041

VMCALL instruction is at fixed, executable address (by design).

- Interesting for ROP

No real validation of port/connection IDs is done

- Can “free” a port ID even when it’s in use
- Can “free” more port IDs than allocated, causing unsigned overflow, and confusion in the future allocation (massive amount of port IDs is allocated)

Undocumented hypercalls exist, which can generate memory dumps, turn off the hypervisor, and more

- Reverse-engineer the hvix64.exe binary to find these
- Most are also visible in WinHV.sys

The Future

Hyper-V will be heavily used in Windows 10 to provide increased platform security and isolation

Pass-the-hash attacks will be mitigated with the introduction of Virtual Trust Levels / Virtual Secure Machine (VTL/VSM)

- Secure Kernel Mode (SKM) and Isolated User Mode (IUM) will provide security boundary even against Root Partition Ring 0 Attackers
- Other ‘trustlets’ may be written over time to also isolate in the same way

Rumors are that Docker-type applications will also use Hyper-V

- Part of codenamed “Pico” APIs in the kernel (SKM runs as a PicoProcess)
- May be called “Chambers”?

Look for a talk on this at a future conference

Full whitepaper will be upcoming in a new Phrack issue (yes, Phrack!)

QUESTIONS?

SEE YOU AT THE NEXT SYSCAN

MARCH 2016 SWISSOTEL MERCHANT COURT

#SYSCAN16