Published September 3, 2018 by Sinaei

Hi guys,

It's the second part of a multiple series of a tutorial called "Hypervisor From Scratch", First I highly recommend to read the first part (Basic Concepts & Configure Testing Environment) before reading this part, as it contains the basic knowledge you need to know in order to understand the rest of this tutorial.

In this section, we will learn about **Detecting Hypervisor Support** for our processor, then we simply config the basic stuff to **Enable VMX** and **Entering VMX Operation** and a lot more thing about **Window Driver Kit (WDK)**.

Configuring Our IRP Major Functions

Beside our kernel-mode driver ("**MyHypervisorDriver**"), I created a user-mode application called "**MyHypervisorApp**", first of all (The source code is available in my GitHub), I should encourage you to write most of your codes in user-mode rather than kernel-mode and that's because you might not have handled exceptions so it leads to BSODs, or on the other hand, running less code in kernel-mode reduces the possibility of putting some nasty kernel-mode bugs.

If you remember from the previous part, we create some Windows Driver Kit codes, now we want to develop our project to support more IRP Major Functions.

IRP Major Functions are located in a conventional Windows table that is created for every device, once you register your device in Windows, you have to introduce these functions in which you handle these IRP Major Functions. That's like every device has a table of its Major Functions and everytime a user-mode application calls any of these functions, Windows finds the corresponding function (if device driver supports that MJ Function) based on the device that requested by the user and calls it then pass an IRP pointer to the kernel driver.

Now its responsibility of device function to check the privileges or etc.

The following code creates the device :

```
NTSTATUS NtStatus = STATUS_SUCCESS;
UINT64 uiIndex = 0;
PDEVICE_OBJECT pDeviceObject = NULL;
UNICODE_STRING usDriverName, usDosDeviceName;

DbgPrint("[*] DriverEntry Called.");

RtlInitUnicodeString(&usDriverName, L"\\Device\\MyHypervisorDevice");

RtlInitUnicodeString(&usDosDeviceName, L"\\DosDevices\\MyHypervisorDevice");

NtStatus = IoCreateDevice(pDriverObject, 0, &usDriverName, FILE_DEVICE_UNKNOWN,
NTSTATUS NtStatusSymLinkResult = IoCreateSymbolicLink(&usDosDeviceName, &usDrive
```

Note that our device name is "**\Device\MyHypervisorDevice**".

After that, we need to introduce our Major Functions for our device.

```
if (NtStatus == STATUS_SUCCESS && NtStatusSymLinkResult == STATUS_SUCCESS)
{
for (uiIndex = 0; uiIndex < IRP_MJ_MAXIMUM_FUNCTION; uiIndex++)
pDriverObject->MajorFunction[uiIndex] = DrvUnsupported;

DbgPrint("[*] Setting Devices major functions.");
pDriverObject->MajorFunction[IRP_MJ_CLOSE] = DrvClose;
pDriverObject->MajorFunction[IRP_MJ_CREATE] = DrvCreate;
pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DrvIOCTLDispatcher;
pDriverObject->MajorFunction[IRP_MJ_READ] = DrvRead;
pDriverObject->MajorFunction[IRP_MJ_WRITE] = DrvWrite;

pDriverObject->DriverUnload = DrvUnload;
}
else {
DbgPrint("[*] There was some errors in creating device.");
}
```

You can see that I put "**DrvUnsupported**" to all functions, this is a function to handle all MJ Functions and told the user that it's not supported. The main body of this function is like this:

```
NTSTATUS DrvUnsupported(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
 DbgPrint("[*] This function is not supported :( !");

 Irp->IoStatus.Status = STATUS_SUCCESS;
 Irp->IoStatus.Information = 0;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);

 return STATUS_SUCCESS;
}
```

We also introduce other major functions that are essential for our device, we'll complete the implementation in the future, let's just leave them alone.

```
NTSTATUS DrvCreate(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
 DbgPrint("[*] Not implemented yet :( !");

 Irp->IoStatus.Status = STATUS_SUCCESS;
 Irp->IoStatus.Information = 0;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);

 return STATUS_SUCCESS;
}

NTSTATUS DrvRead(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp)
{
 DbgPrint("[*] Not implemented yet :( !");

 Irp->IoStatus.Status = STATUS_SUCCESS;
 Irp->IoStatus.Information = 0;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);

 return STATUS_SUCCESS;
}

NTSTATUS DrvWrite(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
 DbgPrint("[*] Not implemented yet :( !");

 Irp->IoStatus.Status = STATUS_SUCCESS;
 Irp->IoStatus.Information = 0;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);

 return STATUS_SUCCESS;
}

NTSTATUS DrvClose(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
 DbgPrint("[*] Not implemented yet :( !");

 Irp->IoStatus.Status = STATUS_SUCCESS;
 Irp->IoStatus.Information = 0;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);

 return STATUS_SUCCESS;
}
```

Now let's see IRP MJ Functions list and other types of Windows Driver Kit handlers routine.

IRP Major Functions List

This is a list of IRP Major Functions which we can use in order to perform different operations.

```
#define IRP_MJ_CREATE                   0x00
#define IRP_MJ_CREATE_NAMED_PIPE        0x01
#define IRP_MJ_CLOSE                    0x02
#define IRP_MJ_READ                     0x03
#define IRP_MJ_WRITE                    0x04
#define IRP_MJ_QUERY_INFORMATION        0x05
#define IRP_MJ_SET_INFORMATION          0x06
#define IRP_MJ_QUERY_EA                 0x07
#define IRP_MJ_SET_EA                   0x08
#define IRP_MJ_FLUSH_BUFFERS            0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION   0x0b
#define IRP_MJ_DIRECTORY_CONTROL        0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL      0x0d
#define IRP_MJ_DEVICE_CONTROL           0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL  0x0f
#define IRP_MJ_SHUTDOWN                 0x10
#define IRP_MJ_LOCK_CONTROL             0x11
#define IRP_MJ_CLEANUP                  0x12
#define IRP_MJ_CREATE_MAILSLOT          0x13
#define IRP_MJ_QUERY_SECURITY           0x14
#define IRP_MJ_SET_SECURITY             0x15
#define IRP_MJ_POWER                    0x16
#define IRP_MJ_SYSTEM_CONTROL           0x17
#define IRP_MJ_DEVICE_CHANGE            0x18
#define IRP_MJ_QUERY_QUOTA              0x19
#define IRP_MJ_SET_QUOTA                0x1a
#define IRP_MJ_PNP                      0x1b
#define IRP_MJ_PNP_POWER                IRP_MJ_PNP      // Obsolete....
#define IRP_MJ_MAXIMUM_FUNCTION         0x1b
```

Every major function will only trigger if we call its corresponding function from user-mode. For instance, there is a function (in user-mode) called **CreateFile** (And all its variants like **CreateFileA** and **CreateFileW** for **ASCII** and **Unicode**) so everytime we call **CreateFile** the function that registered as **IRP_MJ_CREATE** will be called or if we call **ReadFile** then **IRP_MJ_READ** and **WriteFile** then **IRP_MJ_WRITE** will be called. You can see that

Windows treats its devices like files and everything we need to pass from user-mode to kernel-mode is available in **PIRP Irp** as a buffer when the function is called.

In this case, Windows is responsible to copy user-mode buffer to kernel mode stack.

Don't worry we use it frequently in the rest of the project but we only support **IRP_MJ_CREATE** in this part and left others unimplemented for our future parts.

IRP Minor Functions

IRP Minor functions are mainly used for PnP manager to notify for a special event, for example, The PnP manager sends **IRP_MN_START_DEVICE** after it has assigned hardware resources, if any, to the device or The PnP manager sends **IRP_MN_STOP_DEVICE** to stop a device so it can reconfigure the device's hardware resources.

We will need these minor functions later in these series.

A list of IRP Minor Functions is available below:

```
IRP_MN_START_DEVICE
IRP_MN_QUERY_STOP_DEVICE
IRP_MN_STOP_DEVICE
IRP_MN_CANCEL_STOP_DEVICE
IRP_MN_QUERY_REMOVE_DEVICE
IRP_MN_REMOVE_DEVICE
IRP_MN_CANCEL_REMOVE_DEVICE
IRP_MN_SURPRISE_REMOVAL
IRP_MN_QUERY_CAPABILITIES
IRP_MN_QUERY_PNP_DEVICE_STATE
IRP_MN_FILTER_RESOURCE_REQUIREMENTS
IRP_MN_DEVICE_USAGE_NOTIFICATION
IRP_MN_QUERY_DEVICE_RELATIONS
IRP_MN_QUERY_RESOURCES
IRP_MN_QUERY_RESOURCE_REQUIREMENTS
IRP_MN_QUERY_ID
IRP_MN_QUERY_DEVICE_TEXT
IRP_MN_QUERY_BUS_INFORMATION
IRP_MN_QUERY_INTERFACE
IRP_MN_READ_CONFIG
IRP_MN_WRITE_CONFIG
IRP_MN_DEVICE_ENUMERATED
IRP_MN_SET_LOCK
```

Fast I/O

For optimizing VMM, you can use **Fast I/O** which is a different way to initiate I/O operations that are faster than IRP. Fast I/O operations are always synchronous.

According to MSDN:

Fast I/O is specifically designed for rapid synchronous I/O on cached files. In fast I/O operations, data is transferred directly between user buffers and the system cache, bypassing the file system and the storage driver stack. (Storage drivers do not use fast I/O.) If all of the data to be read from a file is resident in the system cache when a fast I/O read or write request is received, the request is satisfied immediately.

When the I/O Manager receives a request for synchronous file I/O (other than paging I/O), it invokes the fast I/O routine first. If the fast I/O routine returns **TRUE**, the operation was serviced by the fast I/O routine. If the fast I/O routine returns **FALSE**, the I/O Manager creates and sends an IRP instead.

The definition of Fast I/O Dispatch table is:

```
typedef struct _FAST_IO_DISPATCH {
  ULONG                                   SizeOfFastIoDispatch;
  PFAST_IO_CHECK_IF_POSSIBLE              FastIoCheckIfPossible;
  PFAST_IO_READ                           FastIoRead;
  PFAST_IO_WRITE                          FastIoWrite;
  PFAST_IO_QUERY_BASIC_INFO               FastIoQueryBasicInfo;
  PFAST_IO_QUERY_STANDARD_INFO            FastIoQueryStandardInfo;
  PFAST_IO_LOCK                           FastIoLock;
  PFAST_IO_UNLOCK_SINGLE                  FastIoUnlockSingle;
  PFAST_IO_UNLOCK_ALL                     FastIoUnlockAll;
  PFAST_IO_UNLOCK_ALL_BY_KEY              FastIoUnlockAllByKey;
  PFAST_IO_DEVICE_CONTROL                 FastIoDeviceControl;
  PFAST_IO_ACQUIRE_FILE                   AcquireFileForNtCreateSection;
  PFAST_IO_RELEASE_FILE                   ReleaseFileForNtCreateSection;
  PFAST_IO_DETACH_DEVICE                  FastIoDetachDevice;
  PFAST_IO_QUERY_NETWORK_OPEN_INFO        FastIoQueryNetworkOpenInfo;
  PFAST_IO_ACQUIRE_FOR_MOD_WRITE          AcquireForModWrite;
  PFAST_IO_MDL_READ                       MdlRead;
  PFAST_IO_MDL_READ_COMPLETE              MdlReadComplete;
  PFAST_IO_PREPARE_MDL_WRITE              PrepareMdlWrite;
  PFAST_IO_MDL_WRITE_COMPLETE             MdlWriteComplete;
  PFAST_IO_READ_COMPRESSED                FastIoReadCompressed;
  PFAST_IO_WRITE_COMPRESSED               FastIoWriteCompressed;
  PFAST_IO_MDL_READ_COMPLETE_COMPRESSED   MdlReadCompleteCompressed;
  PFAST_IO_MDL_WRITE_COMPLETE_COMPRESSED  MdlWriteCompleteCompressed;
  PFAST_IO_QUERY_OPEN                     FastIoQueryOpen;
  PFAST_IO_RELEASE_FOR_MOD_WRITE          ReleaseForModWrite;
  PFAST_IO_ACQUIRE_FOR_CCFLUSH            AcquireForCcFlush;
  PFAST_IO_RELEASE_FOR_CCFLUSH            ReleaseForCcFlush;
} FAST_IO_DISPATCH, *PFAST_IO_DISPATCH;
```

Defined Headers

I created the following headers (source.h) for my driver.

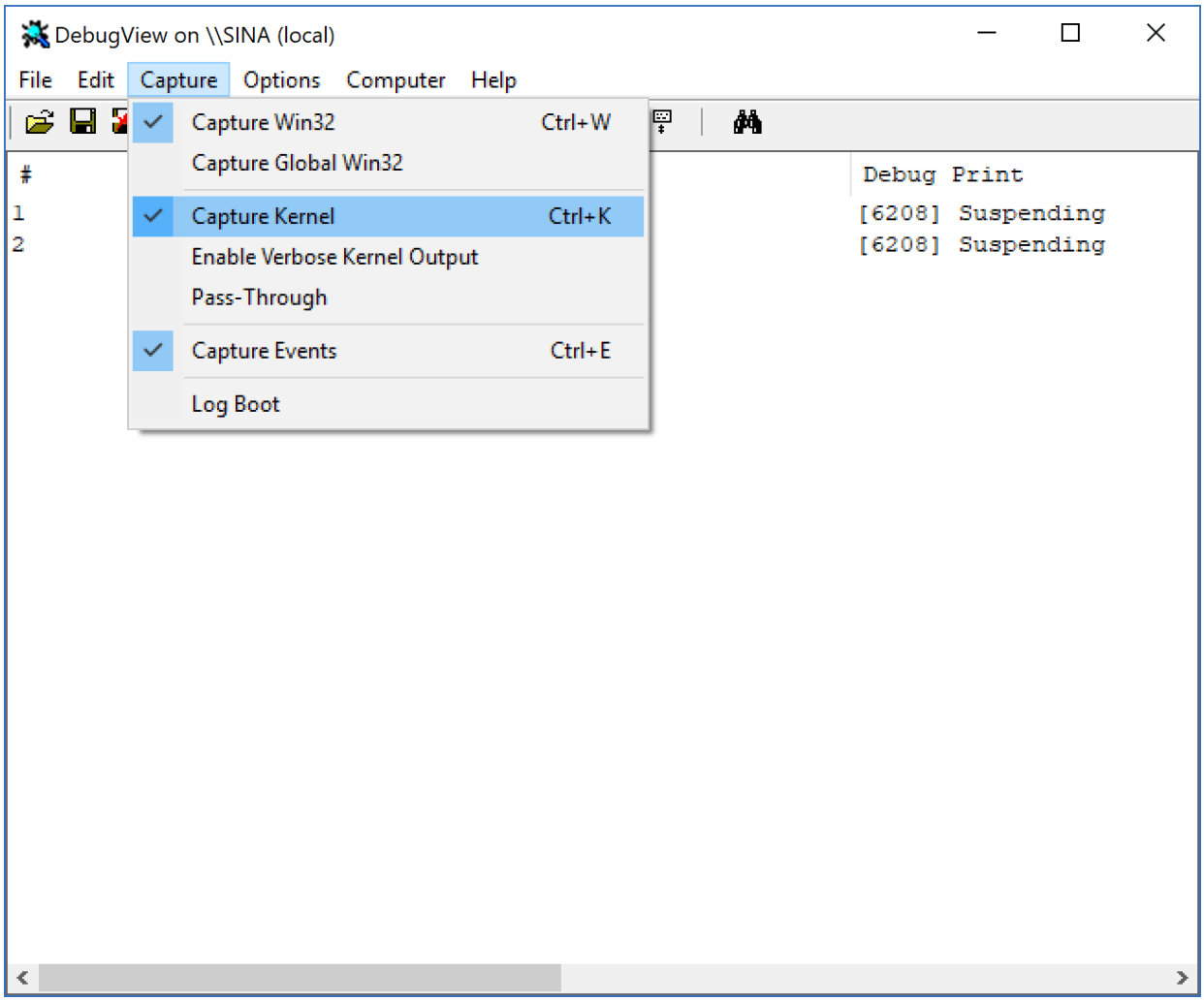
```
#pragma once
#include <ntddk.h>
#include <wdf.h>
#include <wdm.h>

extern void inline Breakpoint(void);
extern void inline Enable_VMX_Operation(void);


NTSTATUS DriverEntry(PDRIVER_OBJECT  pDriverObject, PUNICODE_STRING  pRegistryPat
VOID DrvUnload(PDRIVER_OBJECT  DriverObject);
NTSTATUS DrvCreate(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
NTSTATUS DrvRead(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
NTSTATUS DrvWrite(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
NTSTATUS DrvClose(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
NTSTATUS DrvUnsupported(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
NTSTATUS DrvIOCTLDispatcher(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);

VOID PrintChars(_In_reads_(CountChars) PCHAR BufferAddress, _In_ size_t CountChar
VOID PrintIrpInfo(PIRP Irp);

#pragma alloc_text(INIT, DriverEntry)
#pragma alloc_text(PAGE, DrvUnload)
#pragma alloc_text(PAGE, DrvCreate)
#pragma alloc_text(PAGE, DrvRead)
```

```
#pragma alloc_text(PAGE, DrvWrite)
#pragma alloc_text(PAGE, DrvClose)
#pragma alloc_text(PAGE, DrvUnsupported)
#pragma alloc_text(PAGE, DrvIOCTLDispatcher)



// IOCTL Codes and Its meanings
#define IOCTL_TEST 0x1 // In case of testing
```

Now just compile your driver.

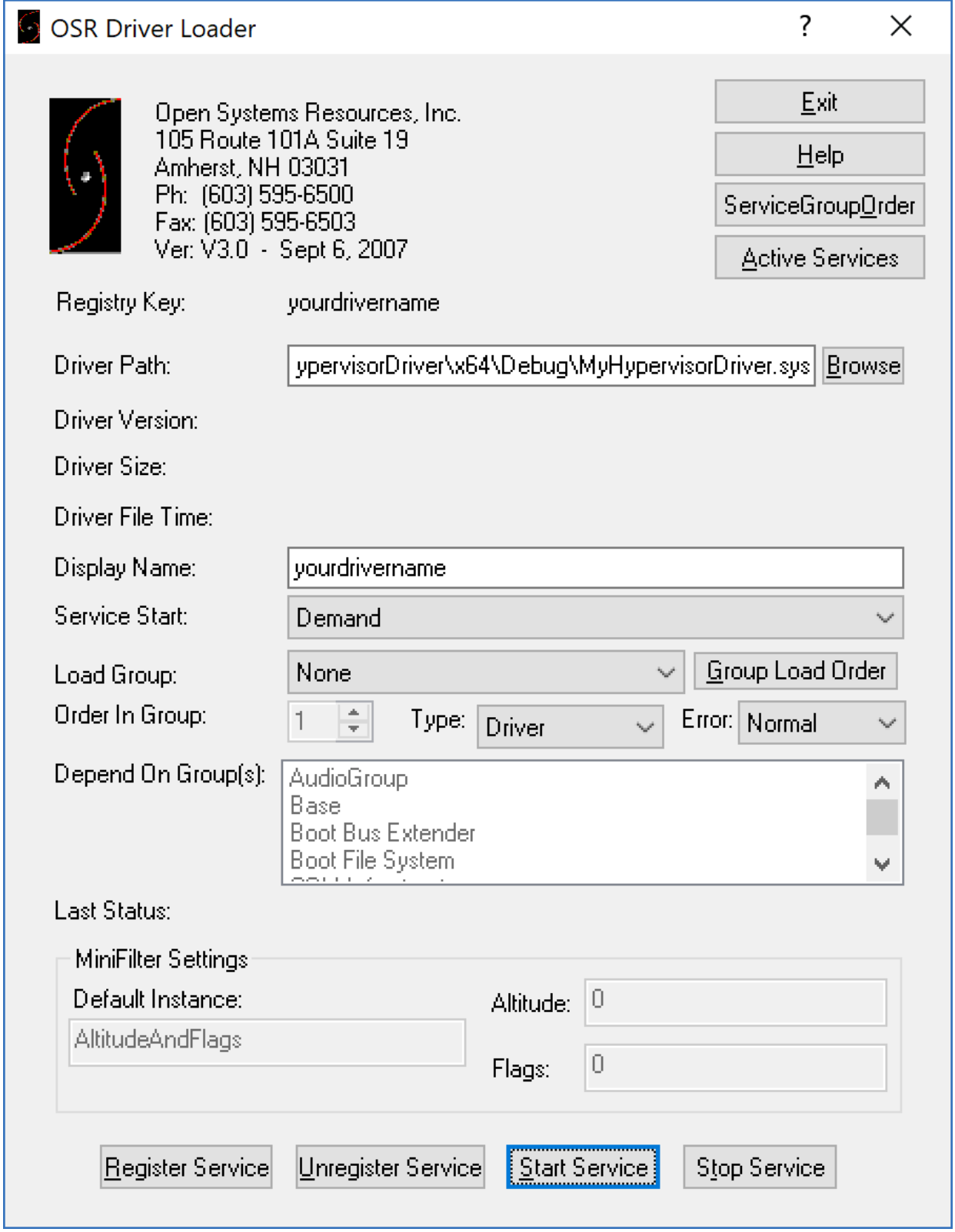
Loading Driver and Check the presence of Device

In order to load our driver (MyHypervisorDriver) first download OSR Driver Loader, then run Sysinternals DbgView as administrator make sure that your DbgView captures the kernel (you can check by going Capture -> Capture Kernel).



After that open the OSR Driver Loader (go to OsrLoader -> kit-> WNET -> AMD64 -> FRE) and open OSRLOADER.exe (in an x64 environment). Now if you built your driver, find .sys file (in MyHypervisorDriver\x64\Debug\ should be a file named: “MyHypervisorDriver.sys”), in OSR Driver Loader click to browse and select (MyHypervisorDriver.sys) and then click to “Register Service” after the message box that shows your driver registered successfully, you should click on “Start Service”.

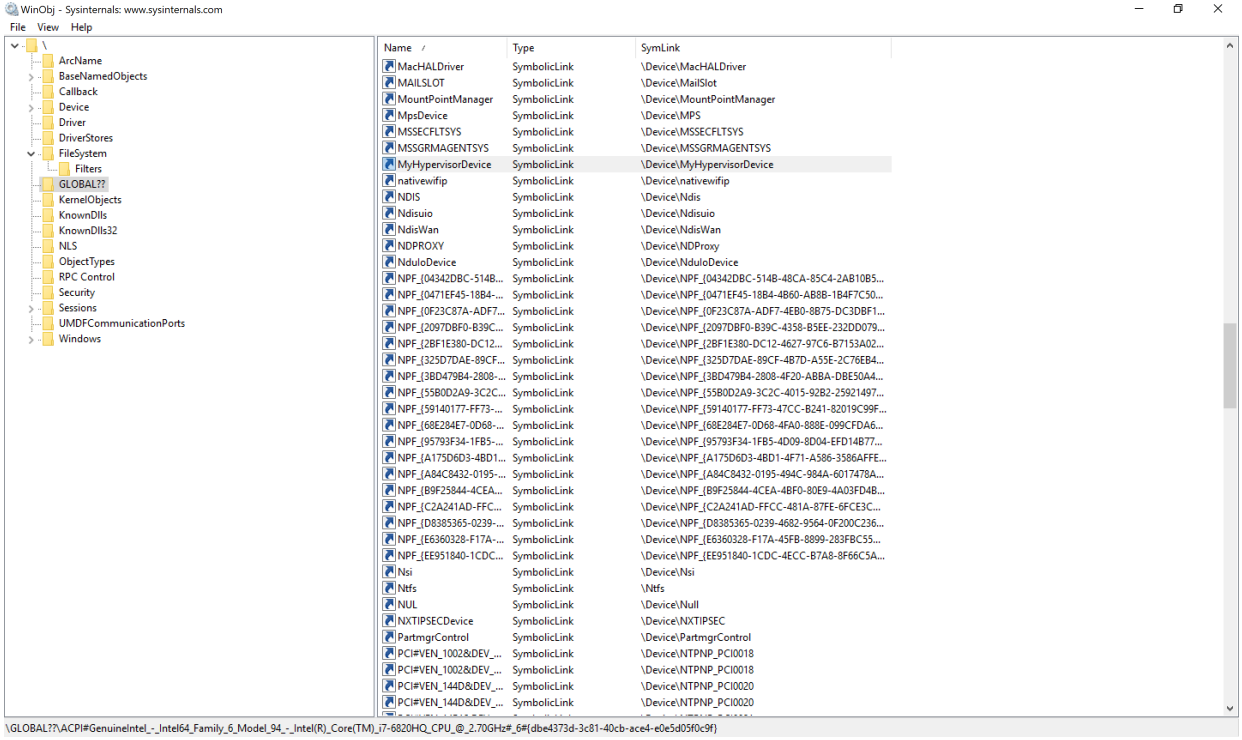
Please note that you should have WDK installed for your Visual Studio in order to be able building your project.



Now come back to DbgView, then you should see that your driver loaded successfully and a message "**[*] DriverEntry Called.** " should appear.

If there is no problem then you're good to go, otherwise, if you have a problem with DbgView you can check the next step.

Keep in mind that now you registered your driver so you can use **SysInternals WinObj** in order to see whether "**MyHypervisorDevice**" is available or not.



The Problem with DbgView

Unfortunately, for some unknown reasons, I'm not able to view the result of DbgPrint(), If you can see the result then you can skip this step but if you have a problem, then perform the following steps:

As I mentioned in part 1:

> In regedit, add a key:
>
> HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter
>
> Under that , add a DWORD value named IHVDRIVER with a value of 0xFFFF
>
> Reboot the machine and you'll good to go.

It always works for me and I tested on many computers but my MacBook seems to have a problem.

In order to solve this problem, you need to find a Windows Kernel Global variable called, **nt!Kd_DEFAULT_Mask,** this variable is responsible for showing the results in DbgView, it has a mask that I'm not aware of so I just put a 0xffffffff in it to simply make it shows everything!

To do this, you need a Windows Local Kernel Debugging using Windbg.

1. Open a Command Prompt window as Administrator. Enter **bcdedit /debug on**
2. If the computer is not already configured as the target of a debug transport, enter **bcdedit /dbgsettings local**

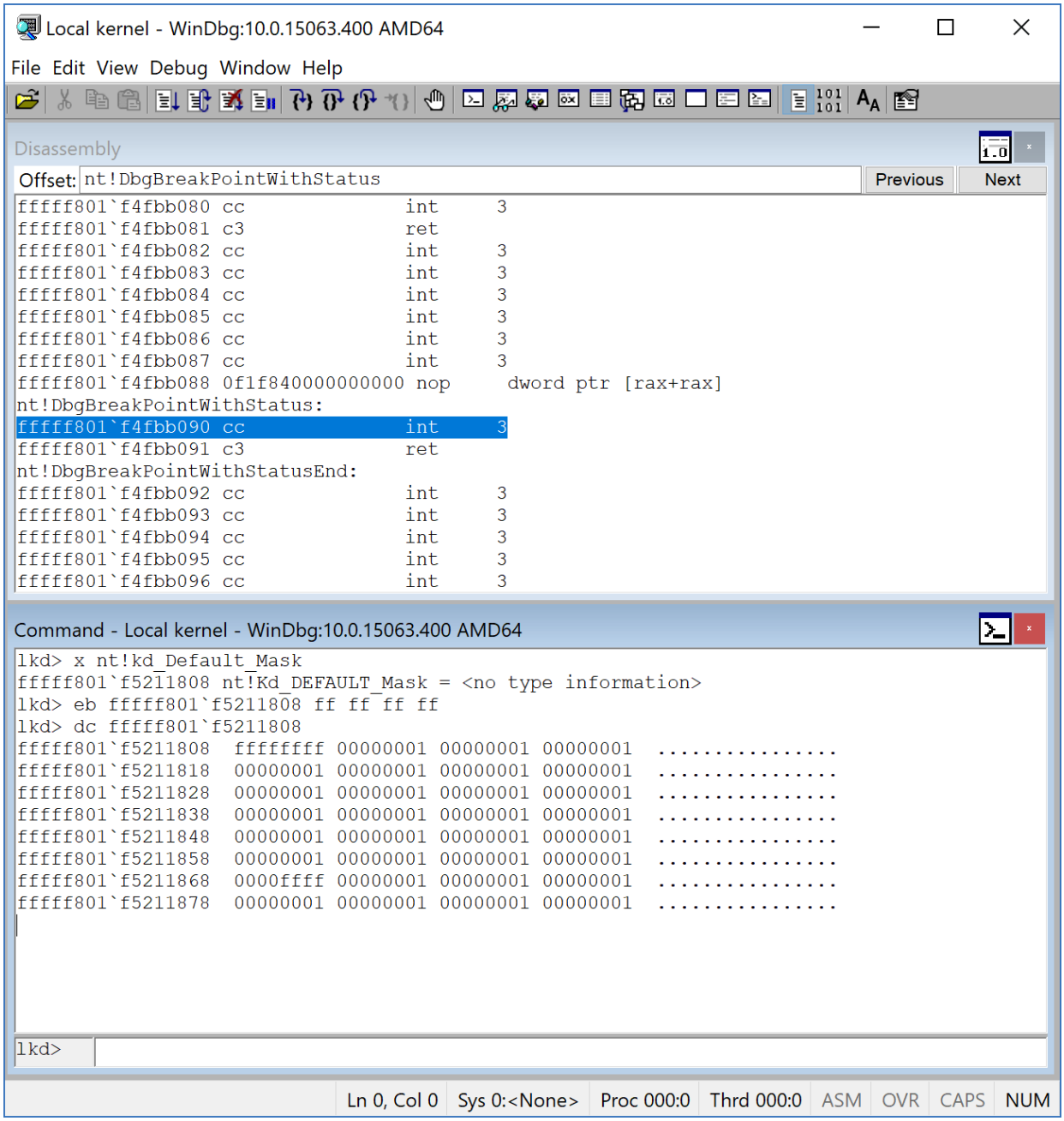
3. Reboot the computer.

After that you need to open Windbg with UAC Administrator privilege, go to File > Kernel Debug > Local > press OK and in you local Windbg find the **nt!Kd_DEFAULT_Mask** using the following command :

```
prlkd> x nt!kd_Default_Mask
fffff801`f5211808 nt!Kd_DEFAULT_Mask = <no type information>
```

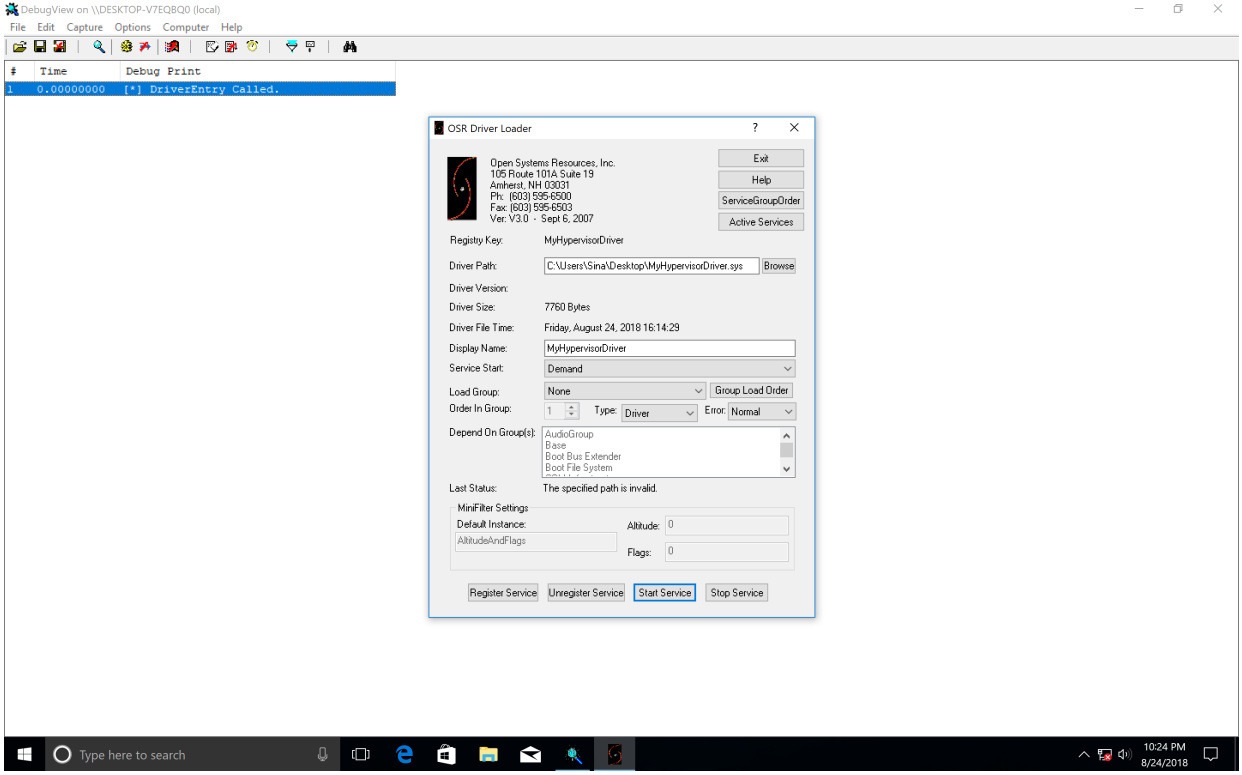
Now change it value to 0xffffffff.

```
lkd> eb fffff801`f5211808 ff ff ff ff
```



After that, you should see the results and now you'll good to go.

Remember this is an essential step for the rest of the topic, because if we can't see any kernel detail then we can't debug.



Detecting Hypervisor Support

Discovering support for **vmx** is the first thing that you should consider before enabling **VT-x**, this is covered in **Intel Software Developer's Manual volume 3C** in section **23.6 DISCOVERING SUPPORT FOR VMX**.

You could know the presence of VMX using **CPUID** if **CPUID.1:ECX.VMX[bit 5] = 1**, then VMX operation is supported.

First of all, we need to know if we're running on an Intel-based processor or not, this can be understood by checking the CPUID instruction and find vendor string "**GenuineIntel**".

The following function returns the vendor string form CPUID instruction.

```
string GetCpuID()
{
 //Initialize used variables
 char SysType[13]; //Array consisting of 13 single bytes/characters
 string CpuID; //The string that will be used to add all the characters to
   //Starting coding in assembly language
 _asm
 {
 //Execute CPUID with EAX = 0 to get the CPU producer
 XOR EAX, EAX
 CPUID
 //MOV EBX to EAX and get the characters one by one by using shift out right bitw
 MOV EAX, EBX
 MOV SysType[0], al
 MOV SysType[1], ah
 SHR EAX, 16
 MOV SysType[2], al
 MOV SysType[3], ah
```

```
//Get the second part the same way but these values are stored in EDX
MOV EAX, EDX
MOV SysType[4], al
MOV SysType[5], ah
SHR EAX, 16
MOV SysType[6], al
MOV SysType[7], ah
//Get the third part
MOV EAX, ECX
MOV SysType[8], al
MOV SysType[9], ah
SHR EAX, 16
MOV SysType[10], al
MOV SysType[11], ah
MOV SysType[12], 00
}
CpuID.assign(SysType, 12);
return CpuID;
}
```

The last step is checking for the presence of VMX, you can check it using the following code :

```
bool VMX_Support_Detection()
{

 bool VMX = false;
 __asm {
 xor    eax, eax
 inc    eax
 cpuid
 bt     ecx, 0x5
 jc     VMXSupport
 VMXNotSupport :
 jmp     NopInstr
 VMXSupport :
 mov    VMX, 0x1
 NopInstr :
 nop
 }

 return VMX;
}
```

As you can see it checks CPUID with EAX=1 and if the 5th (6th) bit is 1 then the VMX Operation is supported. We can also perform the same thing in Kernel Driver.

All in all, our main code should be something like this:

```
int main()
{
 string CpuID;
 CpuID = GetCpuID();
 cout << "[*] The CPU Vendor is : " << CpuID << endl;
 if (CpuID == "GenuineIntel")
 {
 cout << "[*] The Processor virtualization technology is VT-x. \n";
 }
 else
 {
 cout << "[*] This program is not designed to run in a non-VT-x environemnt !\n";
 return 1;
 }

 if (VMX_Support_Detection())
 {
 cout << "[*] VMX Operation is supported by your processor .\n";
 }
 else
```

```
21  {
22  cout << "[*] VMX Operation is not supported by your processor .\n";
23  return 1;
24  }
25  _getch();
26     return 0;
27  }
```

The final result:

Enabling VMX Operation

If our processor supports the VMX Operation then its time to enable it. As I told you above, **IRP_MJ_CREATE** is the first function that should be used to start the operation.

Form Intel Software Developer's Manual (**23.7 ENABLING AND ENTERING VMX OPERATION**):

Before system software can enter VMX operation, it enables VMX by setting CR4.VMXE[bit 13] = 1. VMX operation is then entered by executing the VMXON instruction. VMXON causes an invalid-opcode exception (#UD) if executed with CR4.VMXE = 0. Once in VMX operation, it is not possible to clear CR4.VMXE. System software leaves VMX operation by executing the VMXOFF instruction. CR4.VMXE can be cleared outside of VMX operation after executing of VMXOFF.
VMXON is also controlled by the IA32_FEATURE_CONTROL MSR (MSR address 3AH). This MSR is cleared to zero when a logical processor is reset. The relevant bits of the MSR are:

- Bit 0 is the lock bit. If this bit is clear, VMXON causes a general-protection exception. If the lock bit is set, WRMSR to this MSR causes a general-protection exception; the MSR cannot be modified until a power-up reset condition. System BIOS can use this bit to

provide a setup option for BIOS to disable support for VMX. To enable VMX support in a platform, BIOS must set bit 1, bit 2, or both, as well as the lock bit.

- Bit 1 enables VMXON in SMX operation. If this bit is clear, execution of VMXON in SMX operation causes a general-protection exception. Attempts to set this bit on logical processors that do not support both VMX operation and SMX operation cause general-protection exceptions.
- Bit 2 enables VMXON outside SMX operation. If this bit is clear, execution of VMXON outside SMX operation causes a general-protection exception. Attempts to set this bit on logical processors that do not support VMX operation cause general-protection exceptions.

Setting CR4 VMXE Bit

Do you remember the previous part where I told you how to create an inline assembly in Windows Driver Kit x64?

Now you should create some function to perform this operation in assembly.

Just in Header File (in my case **Source.h**) declare your function:

```
extern void inline Enable_VMX_Operation(void);
```

Then in assembly file (in my case SourceAsm.asm) add this function (Which set the 13th (14th) bit of Cr4).

```
Enable_VMX_Operation PROC PUBLIC
push rax ; Save the state

xor rax,rax ; Clear the RAX
mov rax,cr4
or rax,02000h          ; Set the 14th bit
mov cr4,rax

pop rax ; Restore the state
ret
Enable_VMX_Operation ENDP
```

Also, declare your function in the above of SourceAsm.asm.

```
PUBLIC Enable_VMX_Operation
```

The above function should be called in **DrvCreate**:

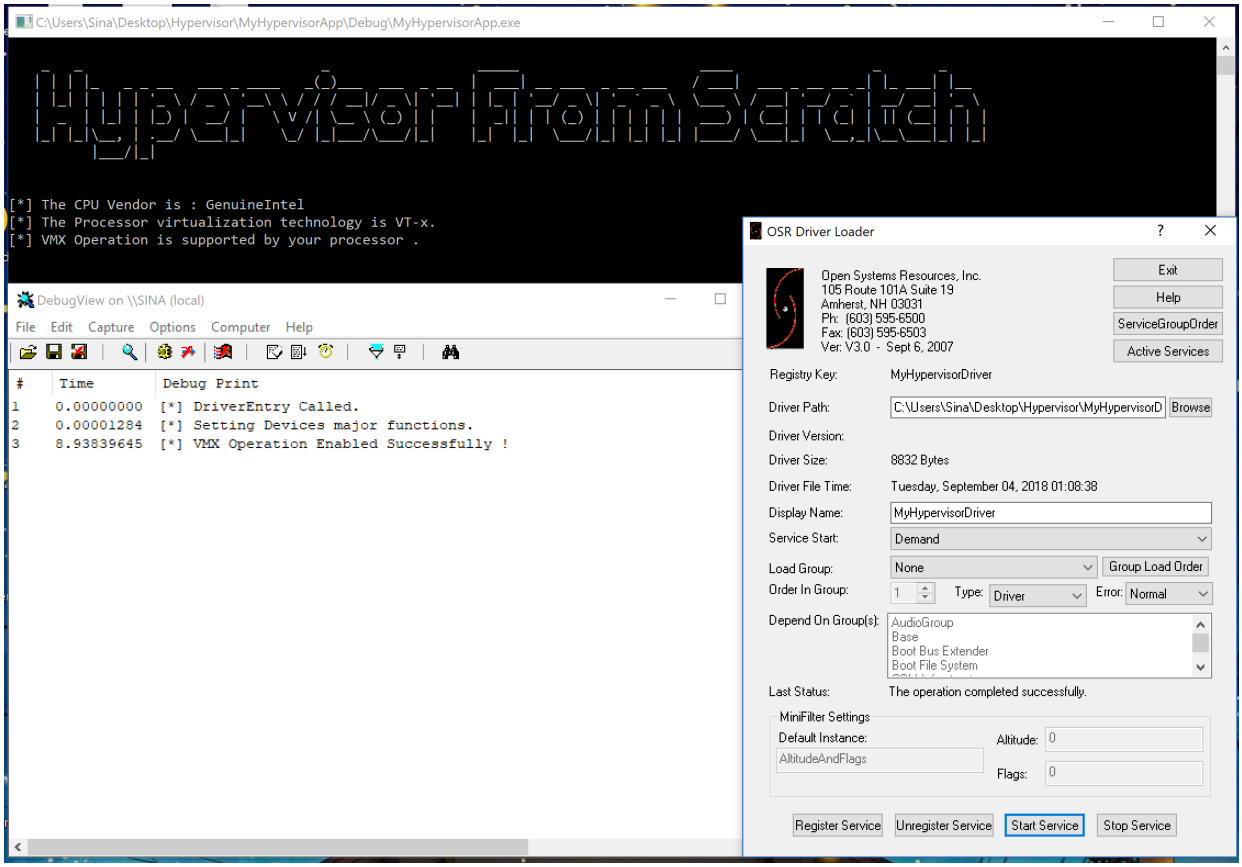
```
NTSTATUS DrvCreate(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
 Enable_VMX_Operation(); // Enabling VMX Operation
 DbgPrint("[*] VMX Operation Enabled Successfully !");
 return STATUS_SUCCESS;
}
```

At last, you should call the following function from the user-mode:

```
 HANDLE hWnd = CreateFile(L"\\\\.\\MyHypervisorDevice",
 GENERIC_READ | GENERIC_WRITE,
```

```
3   FILE_SHARE_READ |
4   FILE_SHARE_WRITE,
5   NULL, /// lpSecurityAttirbutes
6   OPEN_EXISTING,
7   FILE_ATTRIBUTE_NORMAL |
8   FILE_FLAG_OVERLAPPED,
9   NULL); /// lpTemplateFile
```

If you see the following result, then you completed the second part successfully.



Important Note: Please consider that your .asm file should have a different name from your driver main file (.c file) for example if your driver file is “Source.c” then using the name “Source.asm” causes weird linking errors in Visual Studio, you should change the name of you .asm file to something like “SourceAsm.asm” to avoid these kinds of linker errors.

Conclusion

In this part, you learned about basic stuff you to know in order to create a Windows Driver Kit program and then we entered to our virtual environment so we build a cornerstone for the rest of the parts.

In the third part, we’re getting deeper with Intel VT-x and make our driver even more advanced so wait, it’ll be ready soon!

The source code of this topic is available at :

[https://github.com/SinaKarvandi/Hypervisor-From-Scratch/]

References

[1] Intel® 64 and IA-32 architectures software developer's manual combined volumes 3 (https://software.intel.com/en-us/articles/intel-sdm)

[2] IRP_MJ_DEVICE_CONTROL (https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-device-control)

[3] Windows Driver Kit Samples (https://github.com/Microsoft/Windows-driver-samples/blob/master/general/ioctl/wdm/sys/sioctl.c)

[4] Setting Up Local Kernel Debugging of a Single Computer Manually (https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-local-kernel-debugging-of-a-single-computer-manually)

[5] Obtain processor manufacturer using CPUID (https://www.daniweb.com/programming/software-development/threads/112968/obtain-processor-manufacturer-using-cpuid)

[6] Plug and Play Minor IRPs (https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/plug-and-play-minor-irps)

[7] _FAST_IO_DISPATCH structure (https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/ns-wdm-_fast_io_dispatch)

[8] Filtering IRPs and Fast I/O (https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/filtering-irps-and-fast-i-o)

[9] Windows File System Filter Driver Development (https://www.apriorit.com/dev-blog/167-file-system-filter-driver)

PAGES

Blog Map

Tools & Scripts

Tutorials

Sinaei

Judas tree , What kind of mystery is this, that every spring, Comes with our hearts' mourning, Judas tree, You be elate, You sing my unsang song...

Published in **CPU**, **Hypervisor** and **Tutorials**

Creating Virtual Machine | Hypervisor Tutorials | Intel VT-x Tutorial | Setting up Virtual Machine Monitor | VMM Tutorials | VMX Implementation | VMX Tutorials