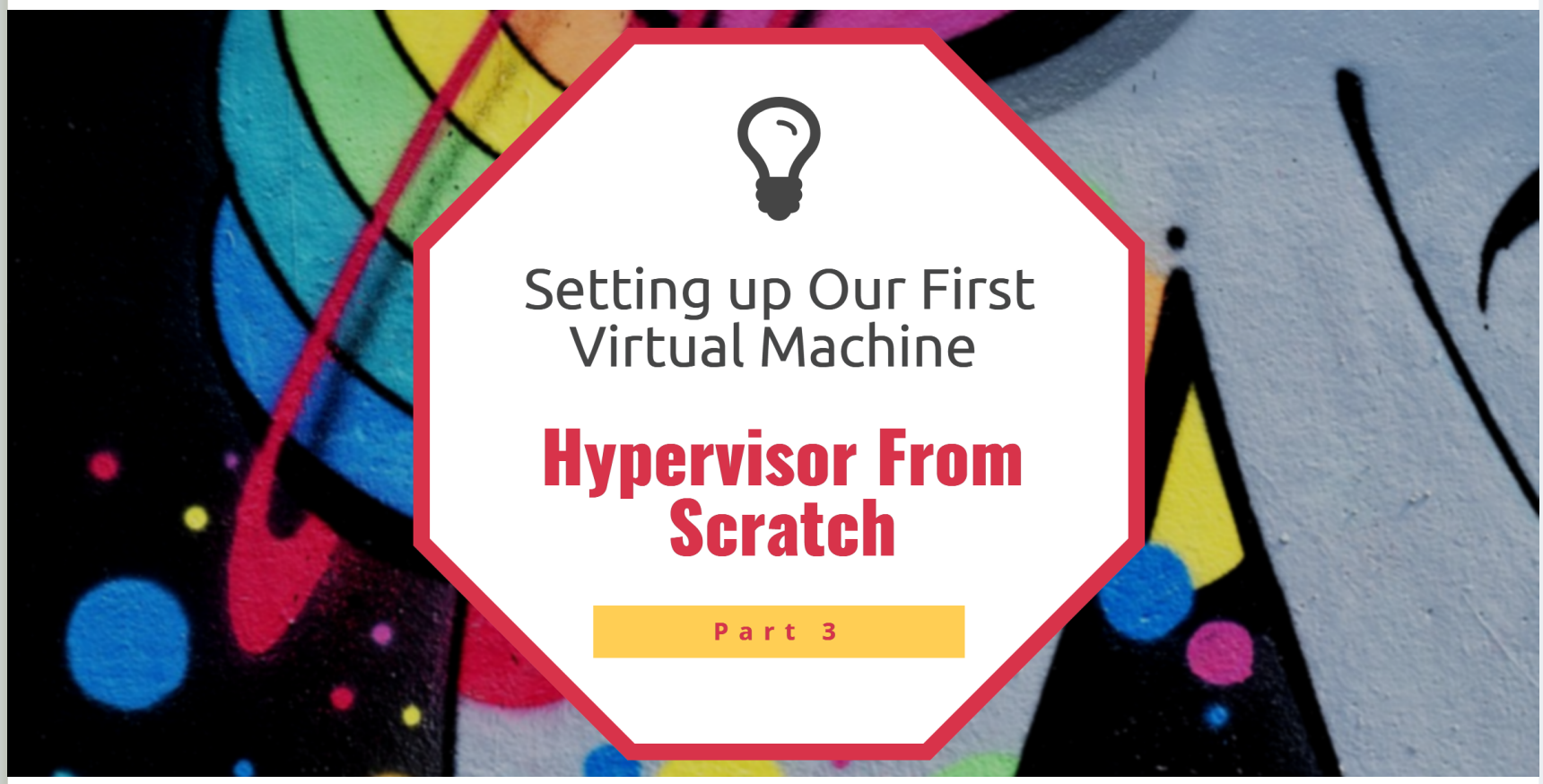
Hypervisor From Scratch – Part 3: Setting up Our First Virtual Machine

Published September 15, 2018 by Sinaei



Introduction

This is the third part of the tutorial "**Hypervisor From Scratch**". You may have noticed that the previous parts have steadily been getting more complicated. This part should teach you how to get started with creating your own VMM, we go to demonstrate how to interact with the VMM from Windows User-mode (**IOCTL Dispatcher**), then we solve the problems with the affinity and running code in a special core. Finally, we get familiar with initializing **VMXON Regions** and **VMCS Regions** then we load our hypervisor regions into each core and implement our custom functions to work with hypervisor instruction and many more things related to Virtual-Machine Control Data Structures (**VMCS**).

Some of the implementations derived from HyperBone (Minimalistic VT-X

hypervisor with hooks) and HyperPlatform by Satoshi Tanda and hvpp which is great work by my friend Petr Beneš the person who really helped me creating these series.

The full source code of this tutorial is available on :

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

Interacting with VMM Driver from User-Mode

The most important function in IRP MJ functions for us is **DrvIOCTLDispatcher (IRP_MJ_DEVICE_CONTROL)** and that's because this function can be called from user-mode with a special IOCTL number, it means you can have a special code in your driver and implement a special functionality corresponding this code, then by knowing the code (from user-mode) you can ask your driver to perform your request, so you can imagine that how useful this function would be.

Now let's implement our functions for dispatching IOCTL code and print it from our kernel-mode driver.

As long as I know, there are several methods by which you can dispatch IOCTL e.g **METHOD_BUFFERED, METHOD_NIETHER, METHOD_IN_DIRECT, METHOD_OUT_DIRECT**. These methods should be followed by the user-mode caller (the difference are in the place where buffers transfer between user-mode and kernel-mode or vice versa), I just copy the implementations with some minor modification form Microsoft's Windows Driver Samples, you can see the full code for user-mode and kernel-mode.

Imagine we have the following IOCTL codes:

```
//
// Device type           -- in the "User Defined" range."
//
#define SIOCTL_TYPE 40000

//
// The IOCTL function codes from 0x800 to 0xFFF are for customer use.
//
#define IOCTL_SIOCTL_METHOD_IN_DIRECT \
    CTL_CODE( SIOCTL_TYPE, 0x900, METHOD_IN_DIRECT, FILE_ANY_ACCESS  )

#define IOCTL_SIOCTL_METHOD_OUT_DIRECT \
    CTL_CODE( SIOCTL_TYPE, 0x901, METHOD_OUT_DIRECT , FILE_ANY_ACCESS  )
```

```
14
15 #define IOCTL_SIOCTL_METHOD_BUFFERED \
16     CTL_CODE( SIOCTL_TYPE, 0x902, METHOD_BUFFERED, FILE_ANY_ACCESS  )
17
18 #define IOCTL_SIOCTL_METHOD_NEITHER \
19     CTL_CODE( SIOCTL_TYPE, 0x903, METHOD_NEITHER , FILE_ANY_ACCESS  )
```

There is a convention for defining IOCTLs as it mentioned here,

The IOCTL is a 32-bit number. The first two low bits define the “transfer type” which can be METHOD_OUT_DIRECT, METHOD_IN_DIRECT, METHOD_BUFFERED or METHOD_NEITHER.

The next set of bits from 2 to 13 define the “Function Code”. The high bit is referred to as the “custom bit”. This is used to determine user-defined IOCTLs versus system defined. This means that function codes 0x800 and greater are customs defined similarly to how WM_USER works for Windows Messages.

The next two bits define the access required to issue the IOCTL. This is how the I/O Manager can reject IOCTL requests if the handle has not been opened with the correct access. The access types are such as FILE_READ_DATA and FILE_WRITE_DATA for example.

The last bits represent the device type the IOCTLs are written for. The high bit again represents user-defined values.

In IOCTL Dispatcher, The “**Parameters.DeviceIoControl.IoControlCode**” of the **IO_STACK_LOCATION** contains the IOCTL code being invoked.

For **METHOD_IN_DIRECT** and **METHOD_OUT_DIRECT**, the difference between IN and OUT is that with IN, you can use the output buffer to pass in data while the OUT is only used to return data.

The **METHOD_BUFFERED** is a buffer that the data is copied from this buffer. The buffer is created as the larger of the two sizes, the input or output buffer. Then the read buffer is copied to this new buffer. Before you return, you simply copy the return data into the same buffer. The return value is put into the **IO_STATUS_BLOCK** and the I/O Manager copies the data into the output buffer. The **METHOD_NEITHER** is the same.

Ok, let's see an example :

First, we declare all our needed variable.

Note that the **PAGED_CODE** macro ensures that the calling thread is running at an IRQL that is low enough to permit paging.

```
NTSTATUS DrvIOCTLDispatcher( PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
 PIO_STACK_LOCATION  irpSp;// Pointer to current stack location
 NTSTATUS            ntStatus = STATUS_SUCCESS;// Assume success
 ULONG               inBufLength; // Input buffer length
 ULONG               outBufLength; // Output buffer length
 PCHAR               inBuf, outBuf; // pointer to Input and output buffer
 PCHAR               data = "This String is from Device Driver !!!";
 size_t              datalen = strlen(data) + 1;//Length of data including
 PMDL                mdl = NULL;
 PCHAR               buffer = NULL;

 UNREFERENCED_PARAMETER(DeviceObject);

 PAGED_CODE();

 irpSp = IoGetCurrentIrpStackLocation(Irp);
 inBufLength = irpSp->Parameters.DeviceIoControl.InputBufferLength;
 outBufLength = irpSp->Parameters.DeviceIoControl.OutputBufferLength;

 if (!inBufLength || !outBufLength)
 {
 ntStatus = STATUS_INVALID_PARAMETER;
 goto End;
 }

...
```

Then we have to use switch-case through the IOCTLs (Just copy buffers and show it from **DbgPrint()**).

```
 switch (irpSp->Parameters.DeviceIoControl.IoControlCode)
 {
 case IOCTL_SIOCTL_METHOD_BUFFERED:

 DbgPrint("Called IOCTL_SIOCTL_METHOD_BUFFERED\n");
 PrintIrpInfo(Irp);
 inBuf = Irp->AssociatedIrp.SystemBuffer;
 outBuf = Irp->AssociatedIrp.SystemBuffer;
 DbgPrint("\tData from User :");
 DbgPrint(inBuf);
 PrintChars(inBuf, inBufLength);
 RtlCopyBytes(outBuf, data, outBufLength);
 DbgPrint(("\tData to User : "));
 PrintChars(outBuf, datalen);
 Irp->IoStatus.Information = (outBufLength < datalen ? outBufLength : data
 break;
```

```
17
18 ...
```

The **PrintIrpInfo** is like this :

```
1  VOID PrintIrpInfo(PIRP Irp)
2  {
3   PIO_STACK_LOCATION  irpSp;
4   irpSp = IoGetCurrentIrpStackLocation(Irp);
5
6   PAGED_CODE();
7
8   DbgPrint("\tIrp->AssociatedIrp.SystemBuffer = 0x%p\n",
9   Irp->AssociatedIrp.SystemBuffer);
10  DbgPrint("\tIrp->UserBuffer = 0x%p\n", Irp->UserBuffer);
11  DbgPrint("\tirpSp->Parameters.DeviceIoControl.Type3InputBuffer = 0x%p\n"
12  irpSp->Parameters.DeviceIoControl.Type3InputBuffer);
13  DbgPrint("\tirpSp->Parameters.DeviceIoControl.InputBufferLength = %d\n",
14  irpSp->Parameters.DeviceIoControl.InputBufferLength);
15  DbgPrint("\tirpSp->Parameters.DeviceIoControl.OutputBufferLength = %d\n"
16  irpSp->Parameters.DeviceIoControl.OutputBufferLength);
17  return;
18 }
```

Even though you can see all the implementations in my GitHub but that's enough, in the rest of the post we only use the **IOCTL_SIOCTL_METHOD_BUFFERED** method.

Now from user-mode and if you remember from the previous part where we create a handle (HANDLE) using **CreateFile**, now we can use the **DeviceIoControl** to call **DrvIOCTLDispatcher** (**IRP_MJ_DEVICE_CONTROL**) along with our parameters from user-mode.

```
1  char OutputBuffer[1000];
2  char InputBuffer[1000];
3  ULONG bytesReturned;
4  BOOL Result;
5
6  StringCbCopy(InputBuffer, sizeof(InputBuffer),
7  "This String is from User Application; using METHOD_BUFFERED");
8
9  printf("\nCalling DeviceIoControl METHOD_BUFFERED:\n");
10
11 memset(OutputBuffer, 0, sizeof(OutputBuffer));
12
13 Result = DeviceIoControl(handle,
14 (DWORD)IOCTL_SIOCTL_METHOD_BUFFERED,
15 &InputBuffer,
16 (DWORD)strlen(InputBuffer) + 1,
17 &OutputBuffer,
18 sizeof(OutputBuffer),
19 &bytesReturned,
20 NULL
21 );
```

```
22
23  if (!Result)
24  {
25  printf("Error in DeviceIoControl : %d", GetLastError());
26  return 1;
27
28  }
29  printf("    OutBuffer (%d): %s\n", bytesReturned, OutputBuffer);
```

There is an old, yet great topic here which describes the different types of IOCT dispatching.

I think we're done with WDK basics, its time to see how we can use Windows in order to build our VMM.

Per Processor Configuration and Setting Affinity

Affinity to a special logical processor is one of the main things that we should consider when working with the hypervisor.

Unfortunately, in Windows, there is nothing like **on_each_cpu** (like it is in Linux Kernel Module) so we have to change our affinity manually in order to run on each logical processor. In my **Intel Core i7 6820HQ** I have 4 physical cores and each core can run 2 threads simultaneously (due to the presence of hyper-threading) thus we have 8 logical processors and of course 8 sets of all the registers (including general purpose registers and MSR registers) so we should configure our VMM to work on 8

logical processors.

To get the count of logical processors you can use **KeQueryActiveProcessors()**, then we should pass a **KAFFINITY** mask to the **KeSetSystemAffinityThread** which sets the system affinity of the current thread.

KAFFINITY mask can be configured using a simple power function :

```
int ipow(int base, int exp) {
 int result = 1;
 for (;;)
 {
 if ( exp & 1)
 {
 result *= base;
 }
 exp >>= 1;
 if (!exp)
 {
 break;
 }
 base *= base;
 }
 return result;
}
```

then we should use the following code in order to change the affinity of the processor and run our code in all the logical cores separately:

```
 KAFFINITY kAffinityMask;
 for (size_t i = 0; i < KeQueryActiveProcessors(); i++)
 {
 kAffinityMask = ipow(2, i);
 KeSetSystemAffinityThread(kAffinityMask);
 DbgPrint("=====================================================");
 DbgPrint("Current thread is executing in %d th logical processor.",i);
 // Put you function here !

 }
```

Conversion between the physical and virtual addresses

VMXON Regions and VMCS Regions (see below) use physical address as the operand to VMXON and VMPTRLD instruction so we should create functions to convert Virtual Address to Physical address:

```
UINT64 VirtualAddress_to_PhysicallAddress(void* va)
{
 return MmGetPhysicalAddress(va).QuadPart;
```

```
}
```

And as long as we can't directly use physical addresses for our modifications in protected-mode then we have to convert physical address to virtual address.

```
UINT64 PhysicalAddress_to_VirtualAddress(UINT64 pa)
{
 PHYSICAL_ADDRESS PhysicalAddr;
 PhysicalAddr.QuadPart = pa;

 return MmGetVirtualForPhysical(PhysicalAddr);
}
```

Query about Hypervisor from the kernel

In the previous part, we query about the presence of hypervisor from user-mode, but we should consider checking about hypervisor from kernel-mode too. This reduces the possibility of getting kernel errors in the future or there might be something that disables the hypervisor using the **lock bit**, by the way, the following code checks **IA32_FEATURE_CONTROL** MSR (MSR address 3AH) to see if the **lock bit** is set or not.

```
BOOLEAN Is_VMX_Supported()
{
 CPUID data = { 0 };

 // VMX bit
 __cpuid((int*)&data, 1);
 if ((data.ecx & (1 << 5)) == 0)
 return FALSE;

 IA32_FEATURE_CONTROL_MSR Control = { 0 };
 Control.All = __readmsr(MSR_IA32_FEATURE_CONTROL);

 // BIOS lock check
 if (Control.Fields.Lock == 0)
 {
 Control.Fields.Lock = TRUE;
 Control.Fields.EnableVmxon = TRUE;
 __writemsr(MSR_IA32_FEATURE_CONTROL, Control.All);
 }
 else if (Control.Fields.EnableVmxon == FALSE)
 {
 DbgPrint("[*] VMX locked off in BIOS");
 return FALSE;
 }

 return TRUE;
}
```

The structures used in the above function declared like this:

```
typedef union _IA32_FEATURE_CONTROL_MSR
{
 ULONG64 All;
 struct
 {
 ULONG64 Lock : 1;                // [0]
 ULONG64 EnableSMX : 1;           // [1]
 ULONG64 EnableVmxon : 1;         // [2]
 ULONG64 Reserved2 : 5;           // [3-7]
 ULONG64 EnableLocalSENTER : 7;   // [8-14]
 ULONG64 EnableGlobalSENTER : 1;  // [15]
 ULONG64 Reserved3a : 16;         //
 ULONG64 Reserved3b : 32;         // [16-63]
 } Fields;
} IA32_FEATURE_CONTROL_MSR, *PIA32_FEATURE_CONTROL_MSR;

typedef struct _CPUID
{
 int eax;
 int ebx;
 int ecx;
 int edx;
} CPUID, *PCPUID;
```

VMXON Region

Before executing VMXON, software should allocate a naturally aligned 4-KByte region of memory that a logical processor may use to support VMX operation. This region is called the **VMXON region**. The address of the **VMXON region** (the VMXON pointer) is provided in an operand to VMXON.

A VMM can (should) use different VMXON Regions for each logical processor otherwise the behavior is “undefined”.

Note: The first processors to support VMX operation require that the following bits be 1 in VMX operation: CR0.PE, CR0.NE, CR0.PG, and CR4.VMXE. The restrictions on CR0.PE and CR0.PG imply that VMX operation is supported only in paged protected mode (including IA-32e mode). Therefore, the guest software cannot be run in unpaged protected mode or in real-address mode.

Now that we are configuring the hypervisor, we should have a global variable that describes the state of our virtual machine, I create the following structure for this purpose, currently, we just have two fields (**VMXON_REGION** and **VMCS_REGION**) but we will add new fields in this structure in the future parts.

```
typedef struct _VirtualMachineState
{
 UINT64 VMXON_REGION;                          // VMXON region
 UINT64 VMCS_REGION;                           // VMCS region
} VirtualMachineState, *PVirtualMachineState;
```

And of course a global variable:

```
extern PVirtualMachineState vmState;
```

I create the following function (in memory.c) to allocate VMXON Region and execute VMXON instruction using the allocated region's pointer.

```
BOOLEAN Allocate_VMXON_Region(IN PVirtualMachineState vmState)
{
 // at IRQL > DISPATCH_LEVEL memory allocation routines don't work
 if (KeGetCurrentIrql() > DISPATCH_LEVEL)
 KeRaiseIrqlToDpcLevel();


 PHYSICAL_ADDRESS PhysicalMax = { 0 };
 PhysicalMax.QuadPart = MAXULONG64;


 int VMXONSize = 2 * VMXON_SIZE;
 BYTE* Buffer = MmAllocateContiguousMemory(VMXONSize + ALIGNMENT_PAGE_SIZE

 PHYSICAL_ADDRESS Highest = { 0 }, Lowest = { 0 };
 Highest.QuadPart = ~0;

 //BYTE* Buffer = MmAllocateContiguousMemorySpecifyCache(VMXONSize + ALIGN

 if (Buffer == NULL) {
 DbgPrint("[*] Error : Couldn't Allocate Buffer for VMXON Region.");
 return FALSE;// ntStatus = STATUS_INSUFFICIENT_RESOURCES;
 }
 UINT64 PhysicalBuffer = VirtualAddress_to_PhysicallAddress(Buffer);

 // zero-out memory
 RtlSecureZeroMemory(Buffer, VMXONSize + ALIGNMENT_PAGE_SIZE);
 UINT64 alignedPhysicalBuffer = (BYTE*)((ULONG_PTR)(PhysicalBuffer + ALIGN

 UINT64 alignedVirtualBuffer = (BYTE*)((ULONG_PTR)(Buffer + ALIGNMENT_PAGE

 DbgPrint("[*] Virtual allocated buffer for VMXON at %llx", Buffer);
 DbgPrint("[*] Virtual aligned allocated buffer for VMXON at %llx", aligne
 DbgPrint("[*] Aligned physical buffer allocated for VMXON at %llx", align

 // get IA32_VMX_BASIC_MSR RevisionId

 IA32_VMX_BASIC_MSR basic = { 0 };


 basic.All = __readmsr(MSR_IA32_VMX_BASIC);

 DbgPrint("[*] MSR_IA32_VMX_BASIC (MSR 0x480) Revision Identifier %llx", b
```

```
44
45
46  //* (UINT64 *)alignedVirtualBuffer  = 04;
47
48  //Changing Revision Identifier
49  *(UINT64 *)alignedVirtualBuffer = basic.Fields.RevisionIdentifier;
50
51
52  int status = __vmx_on(&alignedPhysicalBuffer);
53  if (status)
54  {
55  DbgPrint("[*] VMXON failed with status %d\n", status);
56  return FALSE;
57  }
58
59  vmState->VMXON_REGION = alignedPhysicalBuffer;
60
61  return TRUE;
62 }
```

Let's explain the above function,

```
1  // at IRQL > DISPATCH_LEVEL memory allocation routines don't work
2  if (KeGetCurrentIrql() > DISPATCH_LEVEL)
3  KeRaiseIrqlToDpcLevel();
```

This code is for changing current **IRQL Level** to **DISPATCH_LEVEL** but we can ignore this code as long as we use **MmAllocateContiguousMemory** but if you want to use another type of memory for your VMXON region you should use **MmAllocateContiguousMemorySpecifyCache** (commented), other types of memory you can use can be found here.

Note that to ensure proper behavior in VMX operation, you should maintain the VMCS region and related structures in writeback cacheable memory. Alternatively, you may map any of these regions or structures with the UC memory type. Doing so is strongly discouraged unless necessary as it will cause the performance of transitions using those structures to suffer significantly.

Write-back is a storage method in which data is written into the cache every time a change occurs, but is written into the corresponding location in main memory only at specified intervals or under certain conditions. Being cachable or not cachable can be determined from the **cache disable bit** in paging structures (PTE).

By the way, we should allocate 8192 Byte because there is no guarantee that Windows allocates the aligned memory so we can find a piece of 4096 Bytes aligned in 8196 Bytes. (by aligning I mean, the physical address should be divisible by 4096

without any reminder).

In my experience, the **MmAllocateContiguousMemory** allocation is always aligned, maybe it is because every page in PFN are allocated by 4096 bytes and as long as we need 4096 Bytes, then it's aligned.

If you are interested in Page Frame Number (PFN) then you can read Inside Windows Page Frame Number (PFN) – Part 1 and Inside Windows Page Frame Number (PFN) – Part 2.

```
PHYSICAL_ADDRESS PhysicalMax = { 0 };
PhysicalMax.QuadPart = MAXULONG64;

int VMXONSize = 2 * VMXON_SIZE;
BYTE* Buffer = MmAllocateContiguousMemory(VMXONSize, PhysicalMax);  // Al
if (Buffer == NULL) {
DbgPrint("[*] Error : Couldn't Allocate Buffer for VMXON Region.");
return FALSE;// ntStatus = STATUS_INSUFFICIENT_RESOURCES;
}
```

Now we should convert the address of the allocated memory to its physical address and make sure it's aligned.

Memory that **MmAllocateContiguousMemory** allocates is uninitialized. A kernel-mode driver must first set this memory to zero. Now we should use **RtlSecureZeroMemory** for this case.

```
UINT64 PhysicalBuffer = VirtualAddress_to_PhysicallAddress(Buffer);

// zero-out memory
RtlSecureZeroMemory(Buffer, VMXONSize + ALIGNMENT_PAGE_SIZE);
UINT64 alignedPhysicalBuffer = (BYTE*)((ULONG_PTR)(PhysicalBuffer + ALIGN
UINT64 alignedVirtualBuffer = (BYTE*)((ULONG_PTR)(Buffer + ALIGNMENT_PAGE

DbgPrint("[*] Virtual allocated buffer for VMXON at %llx", Buffer);
DbgPrint("[*] Virtual aligned allocated buffer for VMXON at %llx", aligne
DbgPrint("[*] Aligned physical buffer allocated for VMXON at %llx", align
```

From Intel's manual (24.11.5 VMXON Region):

> Before executing VMXON, software should write the VMCS revision identifier to the VMXON region. (Specifically, it should write the 31-bit VMCS revision identifier to bits 30:0 of the first 4 bytes of the VMXON region; bit 31 should be cleared to 0.)

> It need not initialize the VMXON region in any other way. Software should use a separate region for each logical processor and should not access or modify the VMXON region of a logical processor between the execution of VMXON and VMXOFF on that logical processor. Doing otherwise may lead to unpredictable behavior.

So let's get the Revision Identifier from **IA32_VMX_BASIC_MSR** and write it to our VMXON Region.

```
// get IA32_VMX_BASIC_MSR RevisionId

IA32_VMX_BASIC_MSR basic = { 0 };


basic.All = __readmsr(MSR_IA32_VMX_BASIC);

DbgPrint("[*] MSR_IA32_VMX_BASIC (MSR 0x480) Revision Identifier %llx", b

//Changing Revision Identifier
*(UINT64 *)alignedVirtualBuffer = basic.Fields.RevisionIdentifier;
```

The last part is used for executing VMXON instruction.

```
int status = __vmx_on(&alignedPhysicalBuffer);
if (status)
{
DbgPrint("[*] VMXON failed with status %d\n", status);
return FALSE;
}

vmState->VMXON_REGION = alignedPhysicalBuffer;

return TRUE;
```

__vmx_on is the intrinsic function for executing VMXON. The status code shows diffrenet meanings.

Value	Meaning
0	The operation succeeded.
1	The operation failed with extended status available in the `VM-instruction error field` of the current VMCS.

2	The operation failed without status available.

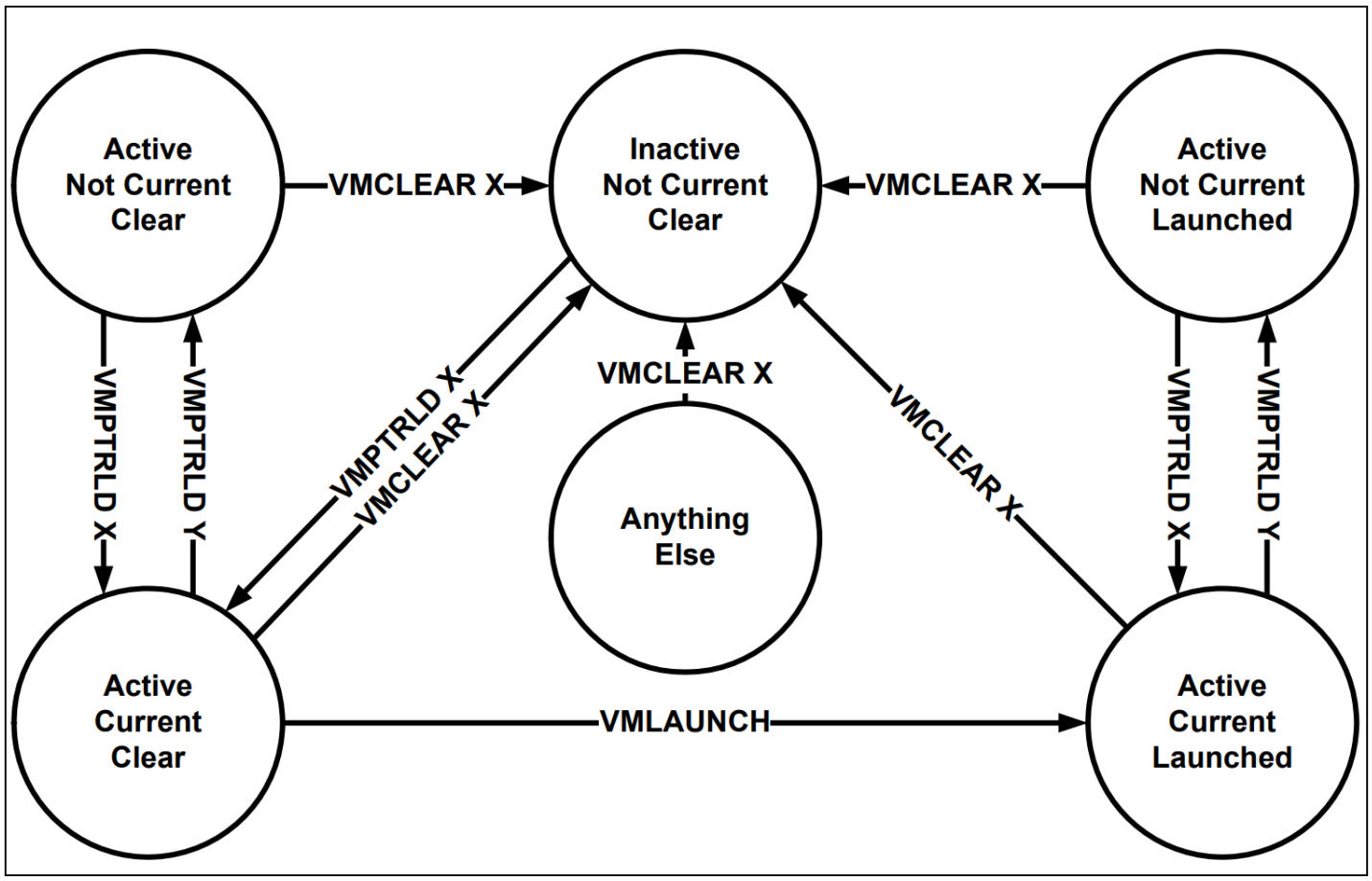
If we set the VMXON Region using VMXON and it fails then status = 1. If there isn't any VMCS the status =2 and if the operation was successful then status =0.

If you execute the above code twice without executing VMXOFF then you definitely get errors.

Now, our VMXON Region is ready and we're good to go.

Virtual-Machine Control Data Structures (VMCS)

A logical processor uses virtual-machine control data structures (VMCSs) while it is in VMX operation. These manage transitions into and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation. This structure is manipulated by the new instructions VMCLEAR, VMPTRLD, VMREAD, and VMWRITE.



The above picture illustrates the lifecycle VMX operation on VMCS Region.

Initializing VMCS Region

A VMM can (should) use different VMCS Regions so you need to set logical processor affinity and run you initialization routine multiple times.

The location where the VMCS located is called “VMCS Region”.

VMCS Region is a

- 4 Kbyte (bits 11:0 must be zero)
- Must be aligned to the 4KB boundary

This pointer must not set bits beyond the processor’s physical-address width (Software can determine a processor’s physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.)

There might be several VMCSs simultaneously in a processor but just one of them is currently active and the VMLAUNCH, VMREAD, VMRESUME, and VMWRITE instructions operate only on the current VMCS.

Using VMPTRLD sets the current VMCS on a logical processor.

The memory operand of the VMCLEAR instruction is also the address of a VMCS. After execution of the instruction, that VMCS is neither active nor current on the logical processor. If the VMCS had been current on the logical processor, the logical processor no longer has a current VMCS.

VMPTRST is responsible to give the current VMCS pointer it stores the value FFFFFFFFFFFFFFFFH if there is no current VMCS.

The launch state of a VMCS determines which VM-entry instruction should be used with that VMCS. The VMLAUNCH instruction requires a VMCS whose launch state is “clear”; the VMRESUME instruction requires a VMCS whose launch state is “launched”. A logical processor maintains a VMCS’s launch state in the corresponding VMCS region.

If the launch state of the current VMCS is “clear”, successful execution of the VMLAUNCH instruction changes the launch state to “launched”.

The memory operand of the VMCLEAR instruction is the address of a VMCS. After

execution of the instruction, the launch state of that VMCS is “clear”.

There are no other ways to modify the launch state of a VMCS (it cannot be modified using VMWRITE) and there is no direct way to discover it (it cannot be read using VMREAD).

The following picture illustrates the contents of a VMCS Region.

Byte Offset	Contents
0	Bits 30:0: VMCS revision identifier Bit 31: shadow-VMCS indicator (see Section 24.10)
4	VMX-abort indicator
8	VMCS data (implementation-specific format)

The following code is responsible for allocating VMCS Region :

```
BOOLEAN Allocate_VMCS_Region(IN PVirtualMachineState vmState)
{
 // at IRQL > DISPATCH_LEVEL memory allocation routines don't work
 if (KeGetCurrentIrql() > DISPATCH_LEVEL)
 KeRaiseIrqlToDpcLevel();


 PHYSICAL_ADDRESS PhysicalMax = { 0 };
 PhysicalMax.QuadPart = MAXULONG64;


 int VMCSSize = 2 * VMCS_SIZE;
 BYTE* Buffer = MmAllocateContiguousMemory(VMCSSize + ALIGNMENT_PAGE_SIZE,

 PHYSICAL_ADDRESS Highest = { 0 }, Lowest = { 0 };
 Highest.QuadPart = ~0;

 //BYTE* Buffer = MmAllocateContiguousMemorySpecifyCache(VMXONSize + ALIGN

 UINT64 PhysicalBuffer = VirtualAddress_to_PhysicallAddress(Buffer);
 if (Buffer == NULL) {
 DbgPrint("[*] Error : Couldn't Allocate Buffer for VMCS Region.");
 return FALSE;// ntStatus = STATUS_INSUFFICIENT_RESOURCES;
 }
 // zero-out memory
 RtlSecureZeroMemory(Buffer, VMCSSize + ALIGNMENT_PAGE_SIZE);
 UINT64 alignedPhysicalBuffer = (BYTE*)((ULONG_PTR)(PhysicalBuffer + ALIGN

 UINT64 alignedVirtualBuffer = (BYTE*)((ULONG_PTR)(Buffer + ALIGNMENT_PAGE



 DbgPrint("[*] Virtual allocated buffer for VMCS at %llx", Buffer);
 DbgPrint("[*] Virtual aligned allocated buffer for VMCS at %llx", aligned
 DbgPrint("[*] Aligned physical buffer allocated for VMCS at %llx", aligne

```

```
37  // get IA32_VMX_BASIC_MSR RevisionId
38
39  IA32_VMX_BASIC_MSR basic = { 0 };
40
41
42  basic.All = __readmsr(MSR_IA32_VMX_BASIC);
43
44  DbgPrint("[*] MSR_IA32_VMX_BASIC (MSR 0x480) Revision Identifier %llx", b
45
46
47  //Changing Revision Identifier
48  *(UINT64 *)alignedVirtualBuffer = basic.Fields.RevisionIdentifier;
49
50
51  int status = __vmx_vmptrld(&alignedPhysicalBuffer);
52  if (status)
53  {
54  DbgPrint("[*] VMCS failed with status %d\n", status);
55  return FALSE;
56  }
57
58  vmState->VMCS_REGION = alignedPhysicalBuffer;
59
60  return TRUE;
61 }
```

The above code is exactly the same as VMXON Region except for **__vmx_vmptrld** instead of **__vmx_on**, **__vmx_vmptrld** is the intrinsic function for VMPTRLD instruction.

In VMCS also we should find the **Revision Identifier** from **MSR_IA32_VMX_BASIC** and write in VMCS Region before executing VMPTRLD.

The MSR_IA32_VMX_BASIC is defined as below.

```
1  typedef union _IA32_VMX_BASIC_MSR
2  {
3   ULONG64 All;
4   struct
5   {
6   ULONG32 RevisionIdentifier : 31;   // [0-30]
7   ULONG32 Reserved1 : 1;             // [31]
8   ULONG32 RegionSize : 12;           // [32-43]
9   ULONG32 RegionClear : 1;           // [44]
10  ULONG32 Reserved2 : 3;             // [45-47]
11  ULONG32 SupportedIA64 : 1;         // [48]
12  ULONG32 SupportedDualMoniter : 1;  // [49]
13  ULONG32 MemoryType : 4;            // [50-53]
14  ULONG32 VmExitReport : 1;          // [54]
15  ULONG32 VmxCapabilityHint : 1;     // [55]
16  ULONG32 Reserved3 : 8;             // [56-63]
17   } Fields;
18 } IA32_VMX_BASIC_MSR, *PIA32_VMX_BASIC_MSR;
```

VMXOFF

After configuring the above regions, now its time to think about **DrvClose** when the handle to the driver is no longer maintained by the user-mode application. At this time, we should terminate VMX and free every memory that we allocated before.

The following function is responsible for executing VMXOFF then calling to **MmFreeContiguousMemory** in order to free the allocated memory :

```
1  void Terminate_VMX(void) {
2
3   DbgPrint("\n[*] Terminating VMX...\n");
4
5   KAFFINITY kAffinityMask;
6   for (size_t i = 0; i < ProcessorCounts; i++)
7   {
8   kAffinityMask = ipow(2, i);
9   KeSetSystemAffinityThread(kAffinityMask);
10  DbgPrint("\t\tCurrent thread is executing in %d th logical processor.", i
11
12
13  __vmx_off();
14  MmFreeContiguousMemory(PhysicalAddress_to_VirtualAddress(vmState[i].VMXON
15  MmFreeContiguousMemory(PhysicalAddress_to_VirtualAddress(vmState[i].VMCS_
16
17  }
18
19  DbgPrint("[*] VMX Operation turned off successfully. \n");
20
21 }
```

Keep in mind to convert VMXON and VMCS Regions to virtual address because **MmFreeContiguousMemory** accepts VA, otherwise, it leads to a BSOD.

Ok, It’s almost done!

Testing our VMM

Let’s create a test case for our code, first a function for Initiating VMXON and VMCS Regions through all logical processor.

```
PVirtualMachineState vmState;
int ProcessorCounts;

PVirtualMachineState Initiate_VMX(void) {

 if (!Is_VMX_Supported())
 {
 DbgPrint("[*] VMX is not supported in this machine !");
 return NULL;
 }

 ProcessorCounts = KeQueryActiveProcessorCount(0);
 vmState = ExAllocatePoolWithTag(NonPagedPool, sizeof(VirtualMachineState)


 DbgPrint("\n==================================================\n");

 KAFFINITY kAffinityMask;
 for (size_t i = 0; i < ProcessorCounts; i++)
 {
 kAffinityMask = ipow(2, i);
 KeSetSystemAffinityThread(kAffinityMask);
 // do st here !
 DbgPrint("\t\tCurrent thread is executing in %d th logical processor.", i

 Enable_VMX_Operation(); // Enabling VMX Operation
```

```
27  DbgPrint("[*] VMX Operation Enabled Successfully !");
28
29  Allocate_VMXON_Region(&vmState[i]);
30  Allocate_VMCS_Region(&vmState[i]);
31
32
33  DbgPrint("[*] VMCS Region is allocated at  ===============> %llx", vmStat
34  DbgPrint("[*] VMXON Region is allocated at ===============> %llx", vmStat
35
36  DbgPrint("\n=================================================\n");
37  }
38 }
```

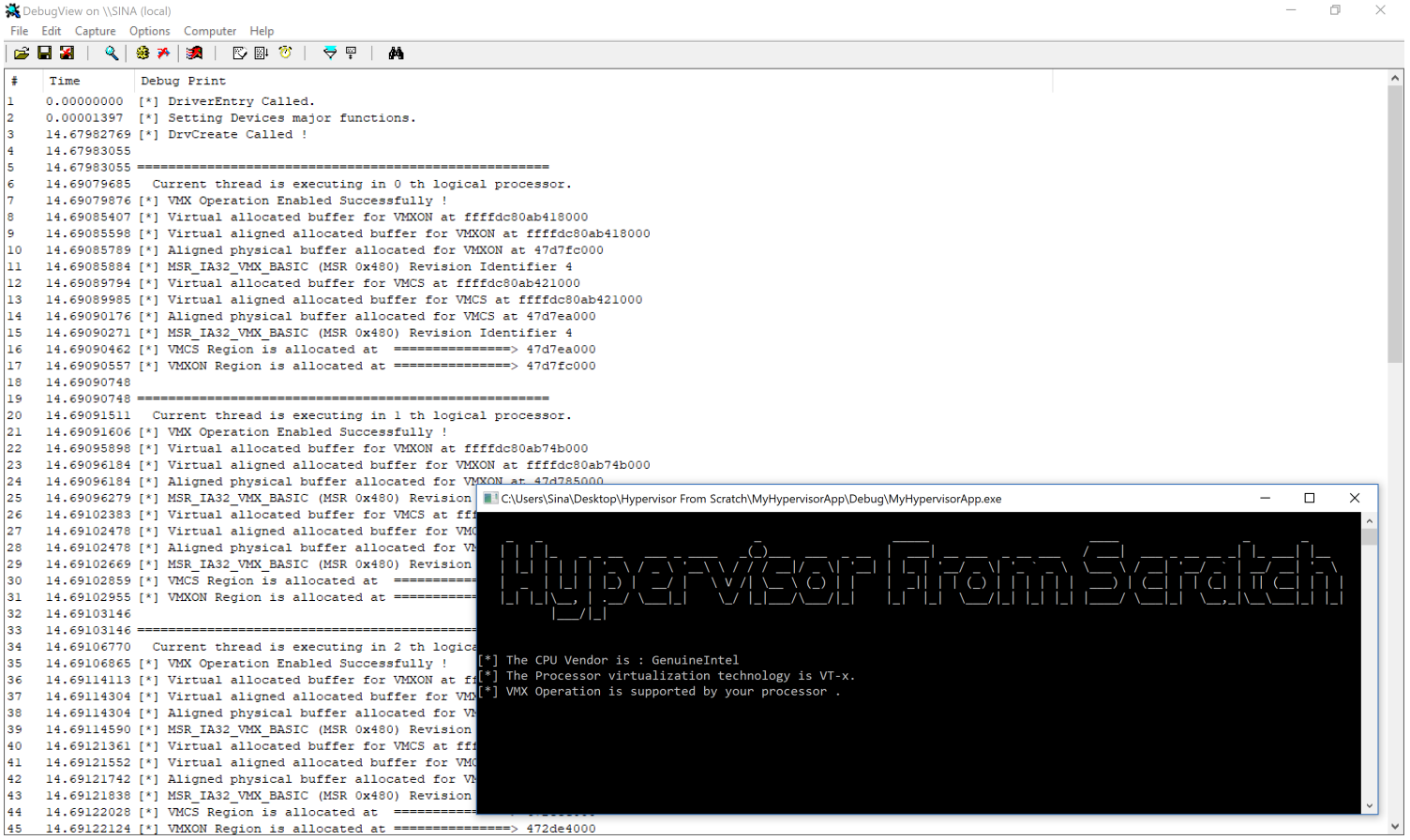
The above function should be called from IRP MJ CREATE so let's modify our **DrvCreate** to :

```
1  NTSTATUS DrvCreate(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
2  {
3
4   DbgPrint("[*] DrvCreate Called !");
5
6   if (Initiate_VMX()) {
7   DbgPrint("[*] VMX Initiated Successfully.");
8   }
9
10  Irp->IoStatus.Status = STATUS_SUCCESS;
11  Irp->IoStatus.Information = 0;
12  IoCompleteRequest(Irp, IO_NO_INCREMENT);
13
14  return STATUS_SUCCESS;
15 }
```

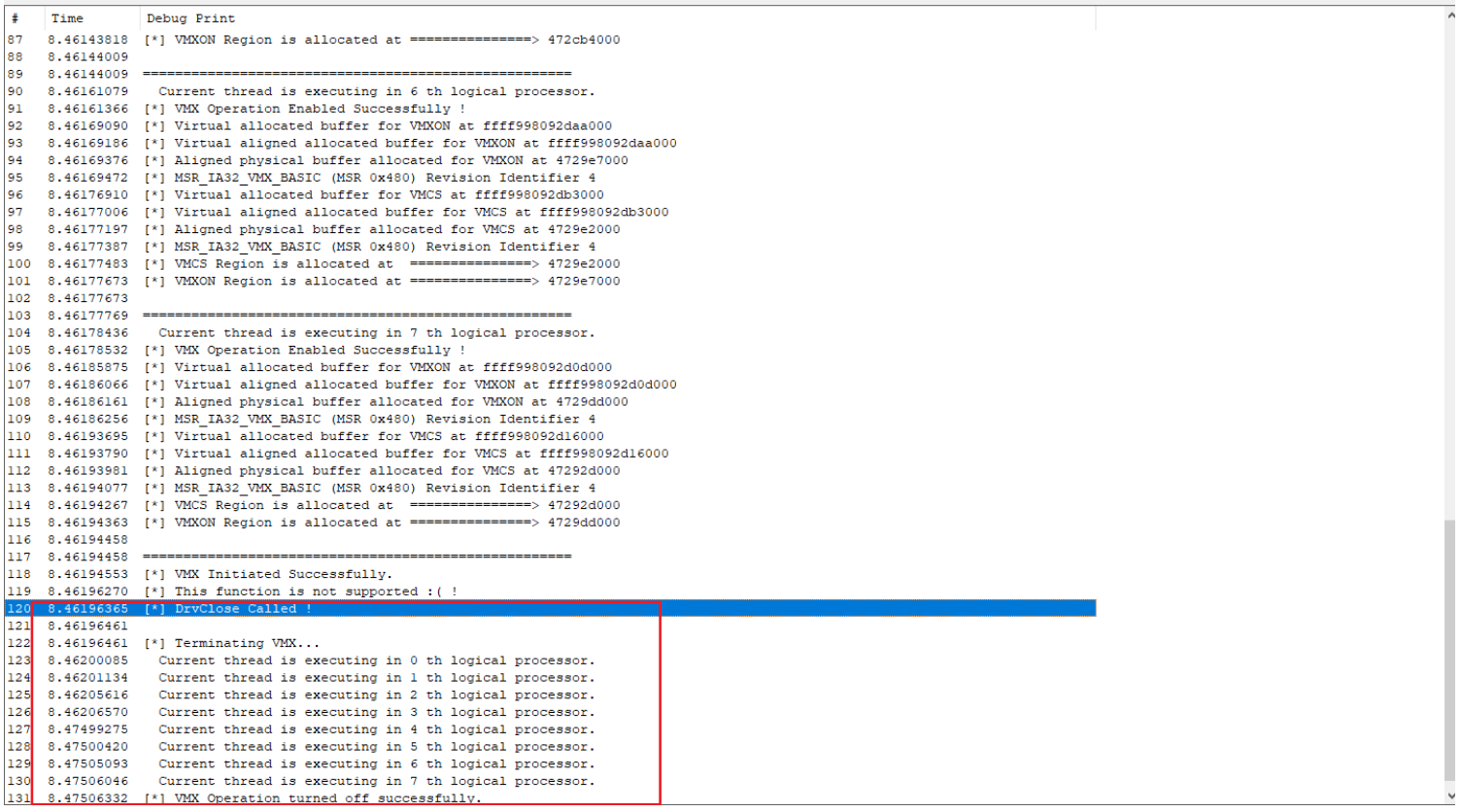
And modify DrvClose to :

```
1  NTSTATUS DrvClose(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
2  {
3   DbgPrint("[*] DrvClose Called !");
4
5   // executing VMXOFF on every logical processor
6   Terminate_VMX();
7
8   Irp->IoStatus.Status = STATUS_SUCCESS;
9   Irp->IoStatus.Information = 0;
10  IoCompleteRequest(Irp, IO_NO_INCREMENT);
11
12  return STATUS_SUCCESS;
13 }
```

Now, run the code, In the case of creating the handle (You can see that our regions allocated successfully).



And when we call **CloseHandle** from user mode:




```
#    Time         Debug Print
87   8.46143818   [*] VMXON Region is allocated at ================> 472cb4000
88   8.46144009
89   8.46144009   ======================================================
90   8.46161079     Current thread is executing in 6 th logical processor.
91   8.46161366   [*] VMX Operation Enabled Successfully !
92   8.46169090   [*] Virtual allocated buffer for VMXON at ffff998092daa000
93   8.46169186   [*] Virtual aligned allocated buffer for VMXON at ffff998092daa000
94   8.46169376   [*] Aligned physical buffer allocated for VMXON at 4729e7000
95   8.46169472   [*] MSR_IA32_VMX_BASIC (MSR 0x480) Revision Identifier 4
96   8.46176910   [*] Virtual allocated buffer for VMCS at ffff998092db3000
97   8.46177006   [*] Virtual aligned allocated buffer for VMCS at ffff998092db3000
98   8.46177197   [*] Aligned physical buffer allocated for VMCS at 4729e2000
99   8.46177387   [*] MSR_IA32_VMX_BASIC (MSR 0x480) Revision Identifier 4
100  8.46177483   [*] VMCS Region is allocated at  ================> 4729e2000
101  8.46177673   [*] VMXON Region is allocated at ================> 4729e7000
102  8.46177673
103  8.46177769   ======================================================
104  8.46178436     Current thread is executing in 7 th logical processor.
105  8.46178532   [*] VMX Operation Enabled Successfully !
106  8.46185875   [*] Virtual allocated buffer for VMXON at ffff998092d0d000
107  8.46186066   [*] Virtual aligned allocated buffer for VMXON at ffff998092d0d000
108  8.46186161   [*] Aligned physical buffer allocated for VMXON at 4729dd000
109  8.46186256   [*] MSR_IA32_VMX_BASIC (MSR 0x480) Revision Identifier 4
110  8.46193695   [*] Virtual allocated buffer for VMCS at ffff998092d16000
111  8.46193790   [*] Virtual aligned allocated buffer for VMCS at ffff998092d16000
112  8.46193981   [*] Aligned physical buffer allocated for VMCS at 47292d000
113  8.46194077   [*] MSR_IA32_VMX_BASIC (MSR 0x480) Revision Identifier 4
114  8.46194267   [*] VMCS Region is allocated at  ================> 47292d000
115  8.46194363   [*] VMXON Region is allocated at ================> 4729dd000
116  8.46194458
117  8.46194458   ======================================================
118  8.46194553   [*] VMX Initiated Successfully.
119  8.46196270   [*] This function is not supported :( !
120  8.46196365   [*] DrvClose Called !
121  8.46196461
122  8.46196461   [*] Terminating VMX...
123  8.46200085     Current thread is executing in 0 th logical processor.
124  8.46201134     Current thread is executing in 1 th logical processor.
125  8.46205616     Current thread is executing in 2 th logical processor.
126  8.46206570     Current thread is executing in 3 th logical processor.
127  8.47499275     Current thread is executing in 4 th logical processor.
128  8.47500420     Current thread is executing in 5 th logical processor.
129  8.47505093     Current thread is executing in 6 th logical processor.
130  8.47506046     Current thread is executing in 7 th logical processor.
131  8.47506332   [*] VMX Operation turned off successfully.
```

Source code

The source code of this part of the tutorial is available on my GitHub.

Conclusion

In this part we learned about different types of IOCTL Dispatching, then we see different functions in Windows to manage our hypervisor VMM and we initialized the VMXON Regions and VMCS Regions then we terminate them.

In the future part, we'll focus on VMCS and different actions that can be performed in VMCS Regions in order to control our guest software.

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] Windows Driver Samples (https://github.com/Microsoft/Windows-driver-samples)

[3] Driver Development Part 2: Introduction to Implementing IOCTLs (https://www.codeproject.com/Articles/9575/Driver-Development-Part-2-Introduction-to-Implemen)

[3] Hyperplatform (https://github.com/tandasat/HyperPlatform)

[4] PAGED_CODE macro (https://technet.microsoft.com/en-us/ff558773(v=vs.96))

[5] HVPP (https://github.com/wbenny/hvpp)

[6] HyperBone Project (https://github.com/DarthTon/HyperBone)

[7] Memory Caching Types (https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/ne-wdm-_memory_caching_type)

[8] What is write-back cache? (https://whatis.techtarget.com/definition/write-back)

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 VMM | Initiating VMX Operation | IRP_MJ_DEVICE_CONTROL | METHOD_BUFFERED | METHOD_IN_DIRECT | METHOD_NIETHER | METHOD_OUT_DIRECT | VMCS | VMCS Region | VMM | VMX Operation | VMXON | VMXON Region

IRQL_EQUALITY

Reply

Really great article, I’ve been diving into HV development and this certainly helped clarify some things. Looking forward to Part 4!

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name*

Jane Doe

Email*

name@email.com

Website

http://google.com

Post Comment

Search …

Search

RECENT POSTS

Hypervisor From Scratch – Part 4: Address Translation Using Extended Page Table (EPT)

Hypervisor From Scratch – Part 3: Setting up Our First Virtual Machine

Using Intel's Streaming SIMD Extensions 3 (MONITOR\MWAIT) As A Kernel Debugging Trick

Hypervisor From Scratch – Part 2: Entering VMX Operation

A Tour of Mount in Linux

RECENT COMMENTS

IRQL_EQUALITY on *Hypervisor From Scratch – Part 3: Setting up Our First Virtual Machine*

Kasbarg (Shayan) on *Hypervisor From Scratch – Part 1: Basic Concepts & Configure Testing Environment*

Sinaei on *Hypervisor From Scratch – Part 2: Entering VMX Operation*

Carl OS on *Hypervisor From Scratch – Part 1: Basic Concepts & Configure Testing Environment*

Necrolis on *Hypervisor From Scratch – Part 2: Entering VMX Operation*

ARCHIVES

October 2018

September 2018

August 2018

July 2018

June 2018

May 2018

April 2018

March 2018

January 2018

December 2017

November 2017

October 2017

September 2017

August 2017

April 2017

March 2017

CATEGORIES

.Net Framework

Android

Cisco

CPU

Debugging

Emulator

Hypervisor

Instrumentation

Kernel Mode

Linux

Malware

Network

Pentest

Programming

Ransomware

Security

Social

Software

SysAdmin

Tutorials

User Mode

Windows

TAGS

active directory Assembly x64 Visual Studio begining cache cisco Create a virtual machine debian debugging kernel mode debug virtual machine debug windows getting started with pykd helloworld How to create Virtual Machine Hypervisor fundamentals Hypervisor Tutorials Intel Virtualization Intel VMX Intel VTX Tutorial ios ipsec kernel-mode linux network opensource Page management in Windows PFN PFN Database proxy PyKD example PyKD sample PyKD scripts PyKD tutorial run PyKD command Setting up Virtual Machine Monitor start systemd tunnel using CPU Virtualization VMCS VMM Implementation VMM Tutorials VMWare and Windbg windows server x64 assembly in driver _MMPFN