

Hello everyone!

Welcome to the first part of a multi-part series of tutorials called "Hypervisor From Scratch". As the name implies, this course contains technical details to create a basic Virtual Machine based on hardware virtualization. If you follow the course, you'll be able to create your own virtual environment and you'll get an understanding of how VMWare, VirtualBox, KVM and other virtualization softwares use processors' facilities to create a virtual environment.

Introduction

Both Intel and AMD support virtualization in their modern CPUs. Intel introduced **(VT-x technology)** that previously codenamed "**Vanderpool**" on November 13, 2005, in Pentium 4 series. The CPU flag for **VT-x** capability is "**vmx**" which stands for **V**irtual **M**achine eXtension.

AMD, on the other hand, developed its first generation of virtualization extensions under the codename "**Pacifica**", and initially published them as AMD **Secure Virtual Machine (SVM)**, but later marketed them under the trademark *AMD Virtualization*, abbreviated *AMD-V*.

There two types of the hypervisor. The hypervisor type 1 called "bare metal hypervisor" or "native" because it runs directly on a bare metal physical server, a type 1 hypervisor has direct access to the hardware. With a type 1 hypervisor, there is no operating system to load as the hypervisor.

Contrary to a type 1 hypervisor, a type 2 hypervisor loads inside an operating system, just like any other application. Because the type 2 hypervisor has to go through the operating system and is managed by the OS, the type 2 hypervisor (and its virtual machines) will run less efficiently (slower) than a type 1 hypervisor.

Even more of the concepts about Virtualization is the same, but you need different considerations in **VT-x** and **AMD-V**. The rest of these tutorials mainly focus on **VT-x** because Intel CPUs are more popular and more widely used. In my opinion, AMD describes virtualization more clearly in its manuals but Intel somehow makes the readers confused especially in Virtualization documentation.

Hypervisor and Platform

These concepts are platform independent, I mean you can easily run the same code routine in both Linux or Windows and expect the same behavior from CPU but I prefer to use Windows as its more easily debuggable (at least for me.) but I try to give some examples for Linux systems whenever needed. Personally, as Linux kernel manages faults like #GP and other exceptions and tries to avoid kernel panic and keep the system up so it's better for testing something like hypervisor or any CPU-related affair. On the other hand, Windows never tries to manage any unexpected exception and it leads to a blue screen of death whenever you didn't manage your exceptions, thus you might get lots of BSODs.By the way, you'd better test it on both platforms (and other platforms too.).

At last, I might (and definitely) make mistakes like wrong implementation or misinformation or forget about mentioning some important description so I should say sorry in advance if I make any faults and I'll be glad for every comment that tells me my mistakes in the technical information or misinformation.

That's enough, Let's get started!

The Tools you'll need

You should have a Visual Studio with WDK installed. you can get Windows Driver Kit (WDK) here.

The best way to debug Windows and any kernel mode affair is using **Windbg** which is available in Windows SDK here. (If you installed WDK with default installing options then you probably install WDK and SDK together so you can skip this step.)

You should be able to debug your OS (in this case Windows) using Windbg, more information here.

Hex-rays IDA Pro is an important part of this tutorial.

OSR Driver Loader which can be downloaded here, we use this tools in order to load our drivers into the Windows machine.

SysInternals DebugView for printing the DbgPrint() results.



Almost all of the codes in this tutorial have to run in kernel level and you must set up either a Linux Kernel Module or Windows Driver Kit (WDK) module. As configuring VMM involves lots of assembly code, you should know how to run inline assembly within you kernel project. In Linux, you shouldn't do anything special but in the case of Windows, WDK no longer supports inline assembly in an x64 environment so if you didn't work on this problem previously then you might have struggle creating a simple x64 inline project but don't worry in one of my post I explained it step by step so I highly recommend seeing this topic to solve the problem before continuing the rest of this part.

Now its time to create a driver!

There is a good article here if you want to start with Windows Driver Kit (WDK).

The whole driver is this :

```
extern void inline AssemblyFunc1(void);
extern void inline AssemblyFunc2(void);
VOID DrvUnload(PDRIVER_OBJECT DriverObject);
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath);
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath)
 NTSTATUS NtStatus = STATUS_SUCCESS;
UINT64 uiIndex = 0;
PDEVICE_OBJECT pDeviceObject = NULL;
UNICODE_STRING usDriverName, usDosDeviceName;
 DbgPrint("DriverEntry Called.");
 RtlInitUnicodeString(&usDriverName, L"\Device\MyHypervisor");
 RtlInitUnicodeString(&usDosDeviceName, L"\DosDevices\MyHypervisor");
 NtStatus = IoCreateDevice(pDriverObject, 0, &usDriverName, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OP
 if (NtStatus == STATUS_SUCCESS)
 pDriverObject->DriverUnload = DrvUnload;
 pDeviceObject->Flags l= I0_TYPE_DEVICE;
 pDeviceObject->Flags &= (~DO_DEVICE_INITIALIZING);
               olicLink(&usDosDeviceName, &usDriverName);
 }
 return NtStatus;
VOID DrvUnload(PDRIVER_OBJECT DriverObject)
          STRING usDosDeviceName;
     Print("DrvUnload Called rn'
                       g(&usDosDeviceName, L"\DosDevices\MyHypervisor");
                        k(&usDosDeviceName);
    DeleteDevice(DriverObject->DeviceObject);
```

AssemblyFunc1 and AssemblyFunc2 are two external functions that defined as inline x64 assembly code.

Our driver needs to register a device so that we can communicate with our virtual environment from User-Mode code, on the hand, I defined **DrvUnload** which use PnP Windows driver feature and you can easily unload your driver and remove device then reload and create a new device.

The following code is responsible for creating a new device :



If you use Windows, then you should disable Driver Signature Enforcement to load your driver, that's because Microsoft prevents any not verified code to run in Windows Kernel (Ring 0).

To do this, press and hold the shift key and restart your computer. You should see a new Window, then

1. Click Advanced options.

- 2. On the new Window Click **Startup Settings**.
- 3. Click on **Restart**.
- 4. On the Startup Settings screen press 7 or F7 to disable driver signature enforcement.

Startup Settings
Press a number to choose from the options below
Use number keys or functions keys F1-F9.
1) Enable debugging
2) Enable boot logging
3) Enable low-resolution video
4) Enable Safe Mode
5) Enable Safe Mode with Networking
6) Enable Safe Mode with Command Prompt
7) Disable driver signature enforcement
8) Disable early launch anti-malware protection
9) Disable automatic restart after failure
Press F10 for more options
Press Enter to return to your operating system

The latest thing I remember is enabling Windows Debugging messages through registry, in this way you can get **DbgPrint()** results through **SysInternals DebugView**.

Just perform the following steps:

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.

Some thoughts before the start

There are some keywords that will be frequently used in the rest of these series and you should know about them (Most of the definitions derived from **Intel software developer's manual, volume 3C**).

Virtual Machine Monitor (VMM): VMM acts as a host and has full control of the processor(s) and other platform hardware. A VMM is able to retain selective control of processor resources, physical memory, interrupt management, and I/O.

Guest Software: Each virtual machine (VM) is a guest software environment.

VMX Root Operation and VMX Non-root Operation: A VMM will run in VMX root operation and guest software will run in VMX non-root operation.

VMX transitions: Transitions between VMX root operation and VMX non-root operation.

VM entries: Transitions into VMX non-root operation.

Extended Page Table (EPT): A modern mechanism which uses a second layer for converting the guest physical address to host physical address.

VM exits: Transitions from VMX non-root operation to VMX root operation.

Virtual machine control structure (VMCS): is a data structure in memory that exists exactly once per VM, while it is managed by the VMM. With every change of the execution context between different VMs, the VMCS is restored for the current VM, defining the state of the VM's virtual processor and VMM control Guest software using VMCS.

The VMCS consists of six logical groups:

- Guest-state area: Processor state saved into the guest state area on VM exits and loaded on VM entries.
- Host-state area: Processor state loaded from the host state area on VM exits.
- VM-execution control fields: Fields controlling processor operation in VMX non-root operation.
- VM-exit control fields: Fields that control VM exits.
- VM-entry control fields: Fields that control VM entries.
- VM-exit information fields: Read-only fields to receive information on VM exits describing the cause and the nature of the VM exit.

I found a great work which illustrates the VMCS, The PDF version is also available here.

GUEST STATE AREA								
CRO	CR3 CR4							
DR7								
RSP			RIP			RFLAGS		
CS	Selector	2	Ba	se Address	Se	gment Limit		Access Right
SS	Selector		Base Address		Se	Segment Limit		Access Right
DS	Selector		Base Address		Se	Segment Limit		Access Right
ES	Selector	×	Base Address		Se	Segment Limit		Access Right
FS	Selector		Base Address		Se	Segment Limit		Access Right
GS	Selector		Ba	se Address	Se	gment Limit		Access Right
LDTR	Selector	X	Ba	se Address	Se	gment Limit		Access Right
TR	Selector		Ba	se Address	Se	gment Limit		Access Right
GDTR	Selector		Ba	se Address	Se	gment Limit		Access Right
IDTR	Selector		Ba	se Address	Se	gment Limit		Access Right
IA32_DEBUGCTL	IA32_SY	SENTER	_CS	IA32_SYSE	INTER_ESI	2	IA32_SY	SENTER_EIP
IA32_PERF_GLOBAL_CT	RL IA32_PAT IA32_EFER IA32_BNDCFGS				BNDCFGS			
Activity state	SMBASE Astivity state							
Activity state Interruptionity state Pending dobug exceptions								
			VMC	5 link pointer	115			
		VM	1X-preer	mption timer v	alue	2		k
Page-directory-pointe	er-table entries PDPTE0 PDPTE1 PDPTE2 PDPTE3					PDPTE3		
			Guest i	nterrupt status	5			
			P					
		Н	OSTS	STATE AR	EA			
CRO	CR3 CR4							
	RSP RIP							
CS	Selector							
SS	Selector							
DS		Selector						
ES	Selector							
FS	Selector Base Address							
GS	Selector	ector Base Address						
TR	Selector Base Address							
GDTR	Base Address							
IDTR	Base Address							
IA32_SYSENTE	R_CS	S IA32_SYSENTER_ESP IA32_SYSENTER_EIP						
IA32_PERF_GLOB	AL_CTRL	IA32_PAT IA32_EFER						

CONTROL FIELDS								
Pin-Based VM-	External-interrupt exiting		NM	Al exiting Virtual NMIs				
Execution Controls	Activate VM	X-preemptic	-preemption timer		Process posted interrupts			
	Interrup	-window exiting			Use TSC offsetting			
Primary processor-	HLT exiting	IN	IVLPG ex	xiting	MWAIT exiting		ing	RDPMC exiting
based	RDTSC exiting	CR	3-load e	exiting		CR3-store exiting		CR8-load exiting
VM-execution	CR8-store exiting	Us	e TPR sh	nadow	NMI-window exiting		xiting	MOV-DR exiting
controls	Unconditional I/O exi	ting Us	e I/O bit	tmaps	Monitor trap flag		flag	Use MSR bitmaps
	MONITOR ex	iting		PAU	SE exiti	exiting Activa		te secondary controls
	Virtualize APIC acces	ses	Enable EPT		Des	Descriptor-table exiting		Enable RDTSCP
Cocondonu	Virtualize x2APIC mo	de l	Enable V	/PID		WBINVD exit	ting	Unrestricted guest
Secondary	APIC-register virt	ualization		Virtual-int	errupt	delivery	P	AUSE-loop exiting
VM execution	RDRAND exiting	En	able IN	VPCID	Er	hable VM fun	ctions	VMCS shadowing
controls	Enable ENCLS exitin	g RI	DSEED e	xiting		Enable PM	IL	EPT-violation #VE
controis	Conceal VMX non-r	oot operatio	on from l	Intel PT		Enab	ole XSAVES	S/XRSTORS
	Mode-based e	xecute conti	rol for El	PT			Use TSC s	caling
Excepti	on Bitmap	1/0	-Bitmap	Addresses	5		TSC	-offset
Guest/Host Masks f	or CR0 Guest/Ho	st Masks for	CR4	Read	Shadow	vs for CR0	Rea	ad Shadows for CR4
CR3-target value 0	CR3-target value	1 CR3	8-target	value 2	C	R3-target val	ue 3	CR3-target count
	APIC-access address Virtual-APIC address TPR threshold			TPR threshold				
APIC Virtualization	APIC Virtualization EOI-exit bitmap 0 EOI-exit bitmap 1 EOI-exit		OI-exit bitma	ap 2	EOI-exit bitmap 3			
6	Posted-inter	rupt notifica	tion vec	tor		Posted-int	errupt de	scriptor address
Read bitmap for low	v MSRs Read bitm	ap for high I	VISRs	Write bit	map fo	or low MSRs	Write	bitmap for low MSRs
Executive-	VMCS Pointer	Extend	ed-Page	-Table Poi	nter Virtual-Processor Identifier			
PLE_Gap	PLE_Window	VM-f	unction	controls	VMREAD bitmap VMWRITE bitmap			
E	ENCLS-exiting bitmap PML address							
Virtualization-except	ion information addres	S	EP	TP index			XSS-e	exiting bitmap
	N	/M-EXI7	r con	NTROL	FIEL	.DS		
	Save debug co	ntrols		Host add	ess spa	ace size	Load IA	32_PERF_GLOBAL_CTRL
VM-Exit Controls	Acknowledge interrup	ot on exit	Save IA3	2_PAT	Load IA	A32_PAT S	ave IA32_	EFER Load IA32_EFER
	Save VMX preemptio	n timer valu	e	Clear IA	32_BNI	DCFGS	Concea	l VM exits from Intel PT
VM-Exit Controls	VM-exit MSR-store co	MSR-store count VM-exit MSR-store address						
for MSRs	VM-exit MSR-load co	unt			VM-e	exit MSR-load	address	
VM-EXIT INFORMATION FIELDS								
Basic VM-Exit Exit reason Exit gualification					ication			
Information	formation Guest-linear address			Gu	est-physic	al address		
VM Exits Due to	VM Exits Due to Vectored Events VM-exit interruption information VM-exit interruption error code					erruption error code		
VM Exits That Occur	VM Exits That Occur During Event Delivery IDT-vectoring information IDT-vectoring error code					oring error code		
		VM	-exit ins	truction le	ngth	1V	M-exit inst	ruction information
VIVI Exits Due to Instruction Execution			RCX	1/	I/O RSI I/O RDI			I/O RIP
VM-instruction error field								

•	Natural-Width fields.
•	16-bits fields.
•	32-bits fields.
•	64-bits fields.

CopyLeft 2017, @Noteworthy (Intel Manuel of July 2017)

Don't worry about the fields, I'll explain most of them clearly in the later parts, just remember VMCS Structure varies between different version of a processor.

VMX Instructions

VMX introduces the following new instructions.

Intel/AMD Mnemonic	Description
INVEPT	Invalidate Translations Derived from EPT
INVVPID	Invalidate Translations Based on VPID
VMCALL	Call to VM Monitor

VMCLEAR	Clear Virtual-Machine Control Structure	
VMFUNC	Invoke VM function	
VMLAUNCH	Launch Virtual Machine	
VMRESUME	Resume Virtual Machine	
VMPTRLD	Load Pointer to Virtual-Machine Control Structure	
VMPTRST	Store Pointer to Virtual-Machine Control Structure	
VMREAD	Read Field from Virtual-Machine Control Structure	
VMWRITE	Write Field to Virtual-Machine Control Structure	
VMXOFF	Leave VMX Operation	
VMXON	Enter VMX Operation	

Life Cycle of VMM Software



- The following items summarize the life cycle of a VMM and its guest software as well as the interactions between them:
 - Software enters VMX operation by executing a VMXON instruction.
 - Using VM entries, a VMM can then turn guests into VMs (one at a time). The VMM effects a VM entry using instructions VMLAUNCH and VMRESUME; it regains control using VM exits.
 - VM exits transfer control to an entry point specified by the VMM. The VMM can take action appropriate to the cause of the VM exit and can then return to the VM using a VM entry.
 - Eventually, the VMM may decide to shut itself down and leave VMX operation. It does so by executing the VMXOFF instruction.

That's enough for now!

In this part, I explained about general keywords that you should be aware and we create a simple lab for our future tests. In the next part, I will explain how to enable VMX on your machine using the facilities we create above, then we survey among the rest of the virtualization so make sure to check the blog for the next part.

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] Hardware-assisted Virtualization (http://www.cs.cmu.edu/~412/lectures/L04_VTx.pdf)

[3] Writing Windows Kernel Driver (https://resources.infosecinstitute.com/writing-a-windows-kernel-driver/)

[4] What Is a Type 1 Hypervisor? (http://www.virtualizationsoftware.com/type-1-hypervisors/)

[5] Intel / AMD CPU Internals (https://github.com/LordNoteworthy/cpu-internals)

[6] Windows 10: Disable Signed Driver Enforcement (https://ph.answers.acer.com/app/answers/detail/a_id/38288/~/windows-10%3A-disable-signed-driverenforcement)

[7] Instruction Set Mapping » VMX Instructions (https://docs.oracle.com/cd/E36784_01/html/E36859/gntbx.html)

PAGES		
Blog Map		
Tools & Scripts		
Tutorials		
Judas tree , What kind of mystery is t	Sinaei this, that every spring, Comes with o be elate, You sing my unsang song	our hearts' mourning, Judas tree, You
Published in CPU, Hypervisor and Tuto	rials	
Create a virtual machine How to create	Virtual Machine Hypervisor fundam	nentals Hypervisor Tutorials
Intel Virtualization Intel VMX Intel V	VTX Tutorial using CPU Virtualization	n VMM Implementation

Sina & Shahriar's Blog An aggressive out-of-order blog...

The contents of this blog is licensed to the public under a **Creative Commons Attribution 4.0** license.