Hyper-V debugging for beginners

Author: Gerhart

Original article
https://hvinternals.blogspot.com/2015/10/hyper-v-debugging-for-beginners.html (blog article can contains some updates in future)

Great thanks to ERNW for the translation of the article!

The article presents a study hypervisor Hyper-V 3.0, which is part of Windows Server 2012. For the study was used the VMware Workstation 9, Windows Server 2012, Windows 7 x86, WinDBG 6.2 and IDA PRO. To create a VMware virtual machine, set the type of the guest OS to - Hyper-V and put the number of processors and cores to 1. Activate the Virtualize Intel VT-x / EPT, install Windows Server 2012 (or Hyper-V Server 2012) to activate the role of Hyper -V and install a guest in relation to the Hyper-V on Windows 7 x86.

1. Terms and definitions

– **The hypervisor –** component of Hyper-V, depending on the manufacturer of the processor (hvix64.exe for Intel and hvax64.exe for AMD). The article discusses the Intel hypervisor processor.

– **Hypercall (hypercall) –** call a given function in the hypervisor using the instructions vmcall

– **Root - partition (the rootpartition) –** Windows Server 2012 with the included component of Hyper-V.

– **VMCS (virtual-machine control structure) –** a structure that defines the logic of the hypervisor.

– **VMX root –** mode, which is running a hypervisor.

– **VMX non-root –** mode in which the running operating system and its client application software.

– **VM exit –** the transition of the **VMX non-root** into **VMX root.** Occurs when the execution of instructions or conditions specified in the VMCS incorporated directly into the logic of the processor.

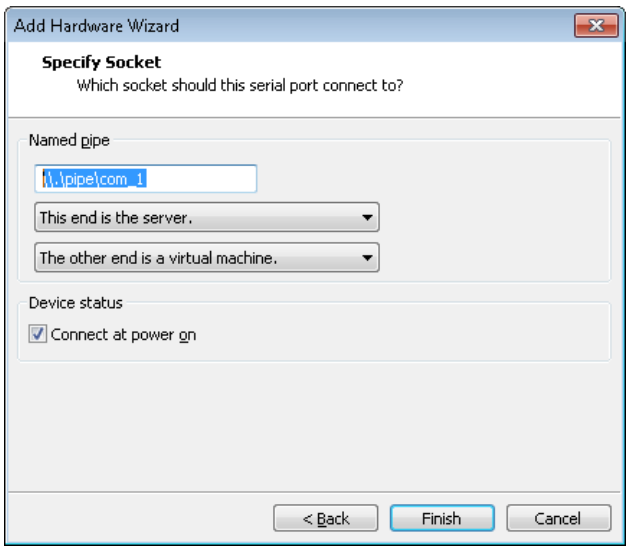
2. Debugging

Hyper- (V) consists of several components, a brief description can be found in (1). For debugging all components except the hypervisor you can use the standard methods, however, to connect to the hypervisor you have to perform a few extra steps to configure root-partition.

For debugging the hypervisor, Microsoft developed a special extension to WinDBG hvexts.dll, which, unfortunately, is not included in the distribution debugger and is available only to partners. Also in the catalog winxp, located in a folder with WinDBG, is an extension of nvkd.dll, which is intended for debugging extensions virtual switch Hyper-V.

The MSDN (2) and (3) is a description of debugging hypervisor via cable through the com-port, implying the presence of two physical machines. However, the hypervisor can be debugged, if you run it in VMware and use the com-port emulator Free Virtual Serial Ports Configuration Utility from the HHD-software (4). To do this:

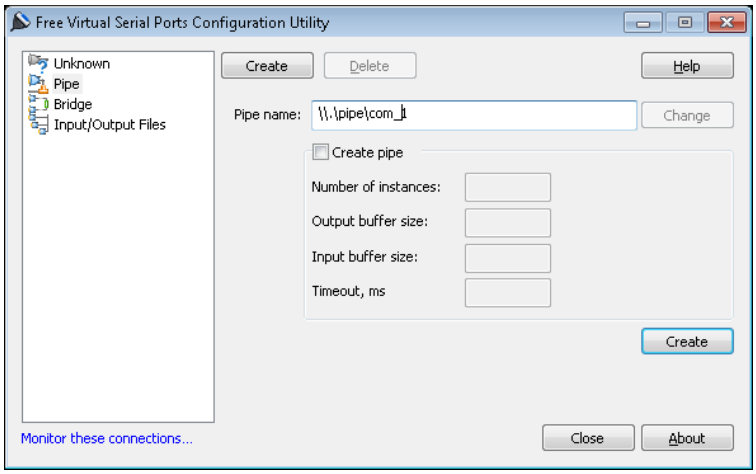
– create com-port for a virtual machine (Hardware->Add->Serial port->Output to a named pipe)



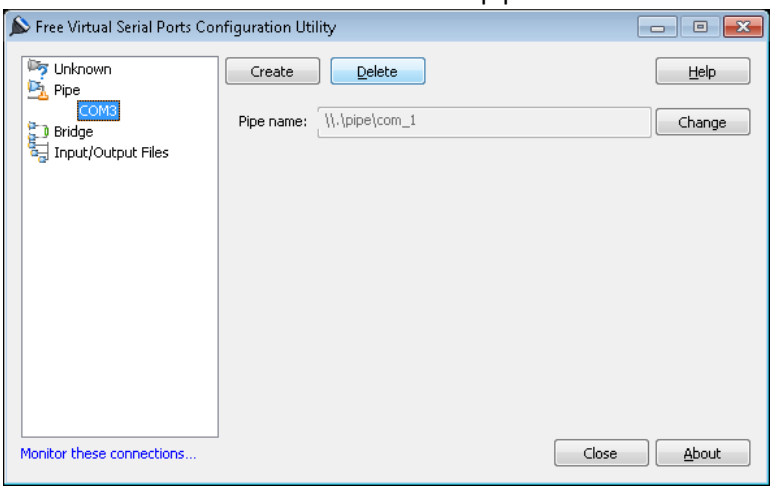
– to perform root-partition commands to configure debugging hypervisor and the OS:

bcdedit /hypervisorsettings serial DEBUGPORT:1 BAUDRATE:115200
bcdedit /set hypervisordebug on
bcdedit /set hypervisorlaunchtype auto
bcdedit /set dbgtransport kdhvcom.dll
bcdedit /dbgsettings serial DEBUGPORT:1 BAUDRATE:115200
bcdedit /debug on
Bcdedit /set bootdebug on (needed to study the process for loading the hypervisor)

– restart Windows Server 2012. pending connections will stop Loading the debugger.
– run Free Virtual Serial Ports Select **Pipe** and press **Create**. In the field of **Pipe name** specify the same value for a virtual machine- \\.\pipe\com_1. Press **Create .**



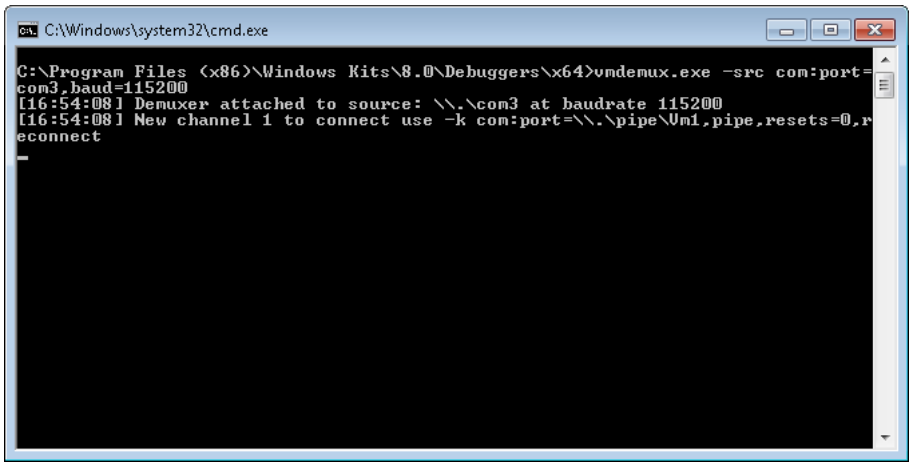
In the case of a successful connection to the named pipe it will create a virtual com -port



- Run vmdemux (located in the Setup directory of WinDBG), specifying the name of the port as one of the parameters:

vmdemux . exe - src com:port=com3,baud= 115200

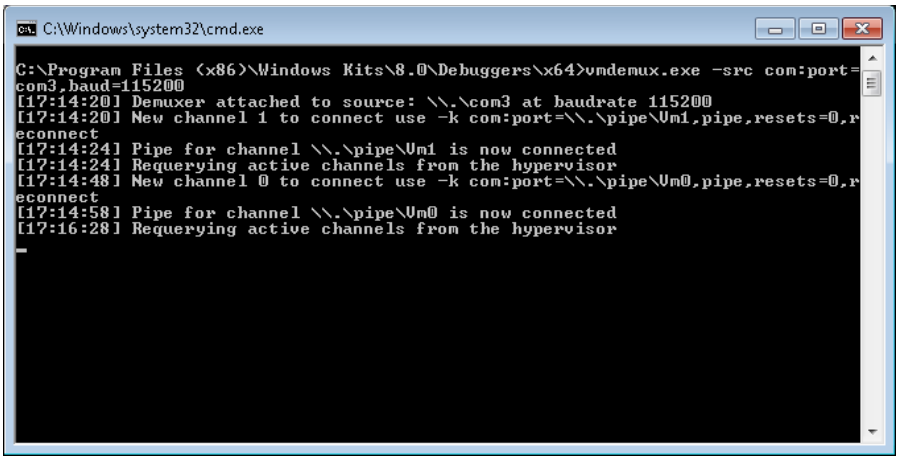
In case of a successful connection we get:



You created a named pipe \\.\pipe\Vm1 must be used to attach the debugger:

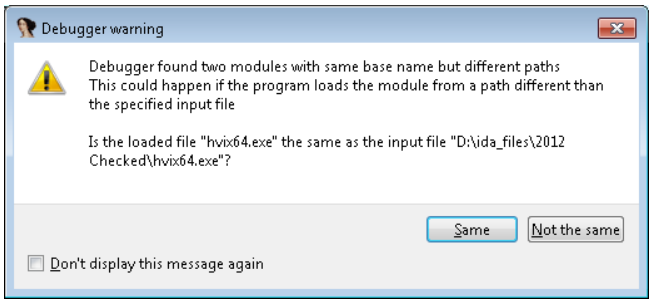
WinDBG.exe-b-k com: port = \\.\pipe\Vm1, pipe, reconnect, resets = 0

At the same time the debugger connects to the root-partition. Then you need to execute a command several times, then vmdemux shall issue:

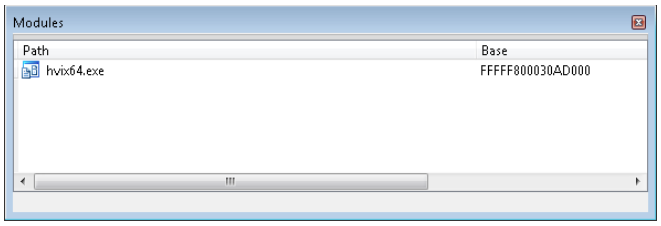


After that, with the help of IDA PRO, you can connect directly to the hypervisor via a named pipe\\.\Pipe\Vm0, choosing as WinDBG debugger and specifying process options in the connection string:**com: port = \\.\pipe\Vm0, pipe, resets = 0**

In case the following message appears choose **Same**.



The debugger will stop within the hypervisor:



However, the above method of debugging is quite slow and relatively unstable (at time of writing, the debugger when you connect to the hypervisor via com-port several times just hang). In Windows Server 2012 hypervisor an opportunity to debug the network, and even on MSDN at the time the article was no description of this method, however, a little digging in to help the team **bcdedit**, you can choose the options you want.
To do this in Windows Server 2012, it is necessary to write

bcdedit/set dbgtransport kdnet.dll
bcdedit/debug to yes
bcdedit/dbgsettings net hostip: 192.168.2.1 port: 50002
in response, the command will display the connection string of the root - partition
bcdedit/set hypervisordebug on
bcdedit/hypervisorsettings NET HOSTIP: 192.168.2.1 PORT: 50000
in response, the command will display the connection string of the hypervisor.

Inside the VMware virtual machine configuration for installing the Host Only adapter, go into the virtual network settings to configure DHCP for the adapter and make sure that Windows Server 2012 is normally assigned to this address, for example, by running the command ipconfig / renew.

Then run 2 instances of IDA PRO, set the debug type to KernelMode and specify the **Process Option->Connection string** to the following line from the command above:

net : port = 50002, Key =
2 ryd 8 (m) 5 mtthis . yomvgm 0 wtjzp 2. ip 83 bg 5 uczdf 1. ya 73 ieco 8 mhj -the rootpartition
net : port = 50000, Key = 2 10
ml 6 pt 2 onihj . hfak 67 vz 3 rei 14. kocxhm 1 ucio 2. lhd 41 tj 99 oa 2- hypervisor
thereby acquiring the ability to simultaneously debug root-partition and the hypervisor.
Option **bcdedit /dbgsettings nodhcp** allows the debugger to use network mode, use the ip-address of the rootpartition. In this case, configuring the DHCP in VMware is not necessary.

Debugging the guest against Hyper-V OS can be made either by the standard method via a virtual com-port or by using the debugging capabilities of the hypervisor. An example was given of a second embodiment is online OSR Online (5), and this is how you can set it up:

- copy the file kdvm.dll from the Windows 8 directory C:\Windows\system32\kdvm.dll same goes for Windows 7 (of course, the file must be identical to the 64-bit operating system). For Windows 8.1 \ Windows Server 2012 R2 kdvm.dll must be taken from preview-build, since the RTM versions of the file has been removed.

- in Windows 7 run following commands

bcdedit/set dbgtransport kdvm.dll

bcdedit/set {default} loadoptions = host_ip "1.2.3.4", host_port = 50005, "encryption_key =" 1.2.3.4 "

bcdedit / set debug on

- restart the OS.

- specify the parameters of the script hyperv-dbg.ps1 (the script in the archive has been adapted for Windows Server 2012 R2 \ Windows 8.1)"

```
#
# Argument initialization
#

$nextarg = "none"
$DebugPort = "50005" #port number (use in windbg connection string)
$targetcomputer = $env:COMPUTERNAME #name of host OS
$VMName = "Windows 7" #virtual machine name
$AutoAssign = "false"
$DebugOff = "false"
```

- Run the script hyperv-dbg.ps1 (run through the „Run as Administrator“, or disable UAC, run gpedit.msc and set Computer configuration \ Windows Settings \ Security Settings \ Local Policies \ Security Options \ User Account Control: Run All administrators in Admin Approval Mode to Disable) in the root-section
- start WinDBG:

  WinDBG -k net:port=50005,target=127.0.0.1,key=1.2.3.4
- execute the command **break,** then the debugger will stop inside the guest OS:

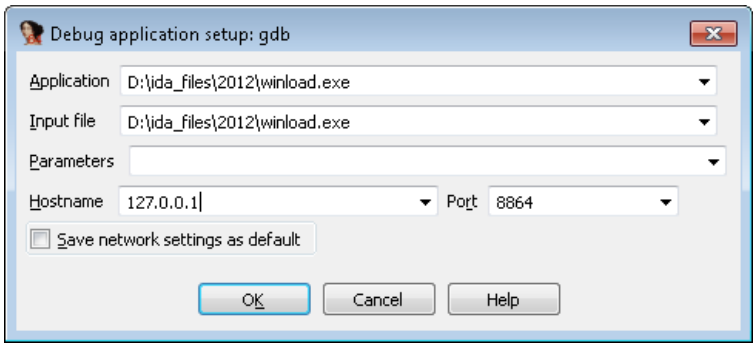
```
kd> vertarget
Windows 7 Kernel Version 7601 (Service Pack 1) MP (1 procs) Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 7601.17514.x86fre.win7sp1_rtm.101119-1850
Machine Name:
Kernel base = 0x82604000 PsLoadedModuleList = 0x8274e850
Debug session time: Mon Jun 10 15:36:14.382 2013 (UTC + 4:00)
System Uptime: 0 days 0:00:52.296
```

Also, for the virtual machine VMware, where Windows Server 2012 is installed on, the gdb-debugger must be enabled. To do this, vmx-file of this machine, you have to add the line

debugStub.listen.guest64 = "TRUE"
debugStub.hideBreakpoints= "TRUE"

3. Loading the hypervisor

The research used checked-file versions on hvloader.exe (6.2.9200.16384) and hvix64.exe (6.2.9200.16384). Before debugging load winload.exe into IDA PRO, choose Debugger -> Select Debugger -> GDB, in the **Process Options** to specify the **Host name** 127.0.0.1 and port 8864.



- Thanks to the previously installed boot loader options **bootdebug on** an early connection to download winload.exe, which produces the hypervisor launch this after the start of the OS, you need to:
- run WinDBG:
- **WinDBG.exe -b -k net:port=50002,key=2ryd8m5mtthis.yomvgm0wtjzp.2ip83bg5uczdf.1ya73ieco8mhj**

  These circumstances must occur within the function winload! DebugService2

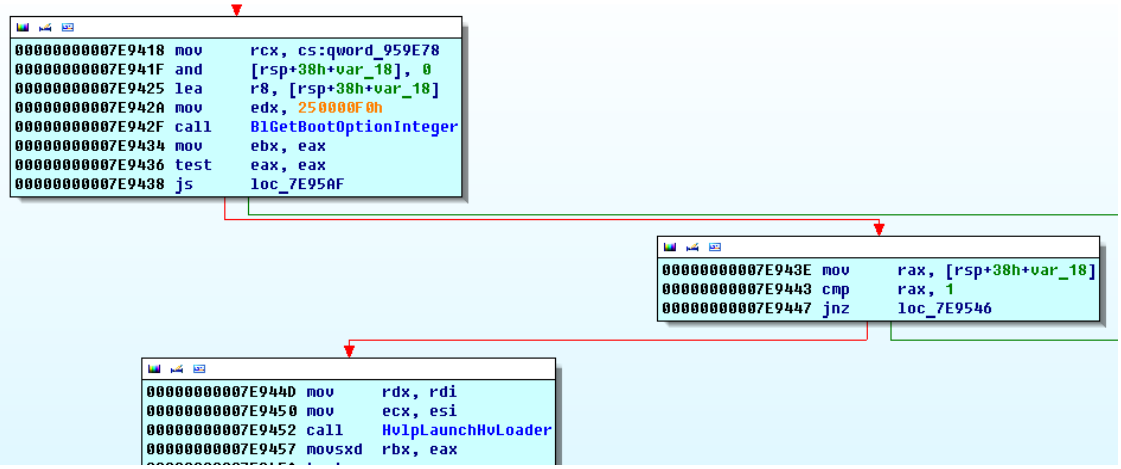
  - find download address of winload.exe

  kd> lm

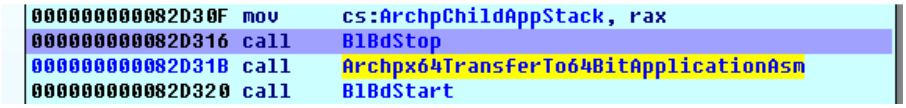
  start end module name
  **00000000`007df000** 00000000`00971000 winload (pdb symbols)
- run IDA PRO and load the previously analyzed module winload.exe, choose Debugger -> attach to process -> attach to process started on target, and after stopping run Edit -> Segments -> Rebase program, specified in the Image base load address winload.exe (0x007df000) and save it in IDA PRO. When loading winload.exe ASLR is not used, so the load address will not change when you restart the operating system and downloading to the IDA PRO winload.exe will be immediately posted to the correct address
  - put in IDA PRO a breakpoint on winload!OslArchHypervisorSetup and continue debugging (F9). Also continue debugging in WinDBG:

  kd> g

Winload checks whether the given parameter loader hypervisorlaunchtype (0x250000f0) is.

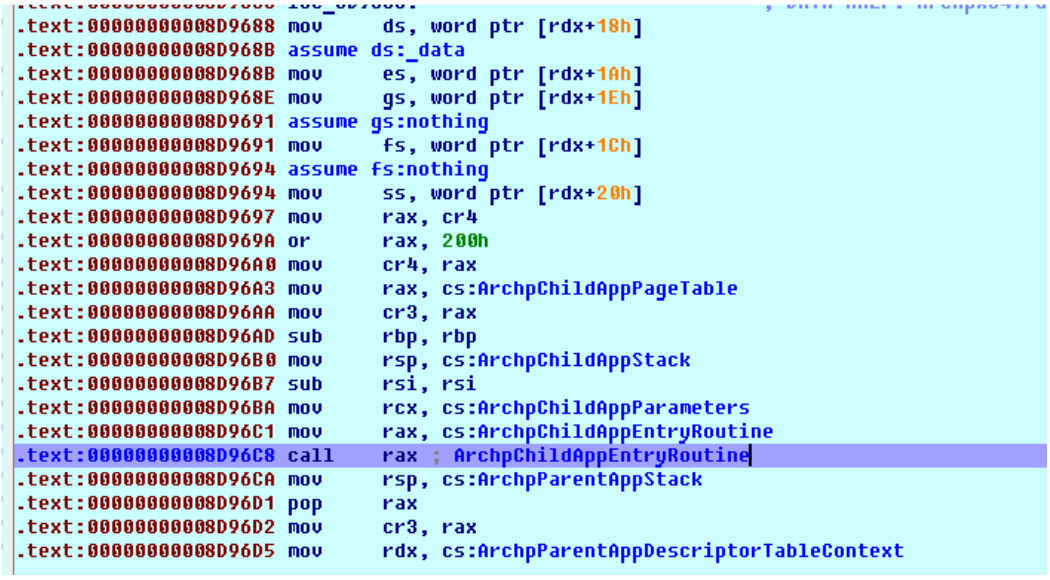


If the parameter is specified and its value is 1 (Auto), the function call HvlpLaunchHvLoader that loads and passes the control module hvloader.exe which will have to download the file of the hypervisor hvix 64.exe and prepare it for future work.



Function BlBdStop shuts off the WinDBG but you debug through gdb in Vmware which cannot be prevented.

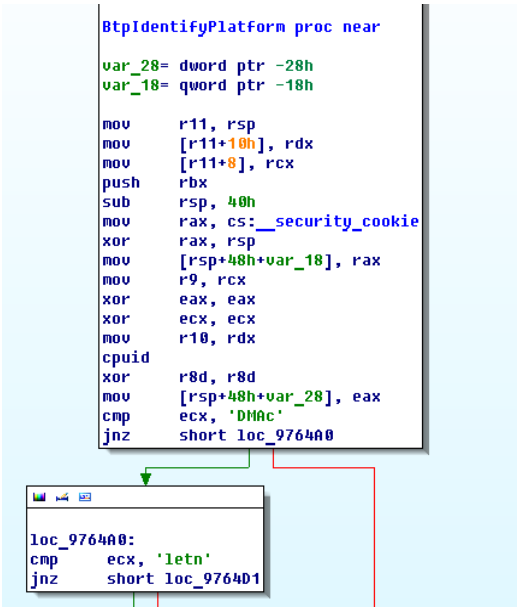
The function Archpx64TransferTo64BitApplicationAsm is used to give control to the hvlMain from hvloader.exe (the address of the functionhvlMain is in ArchpChildAppEntryRoutine).



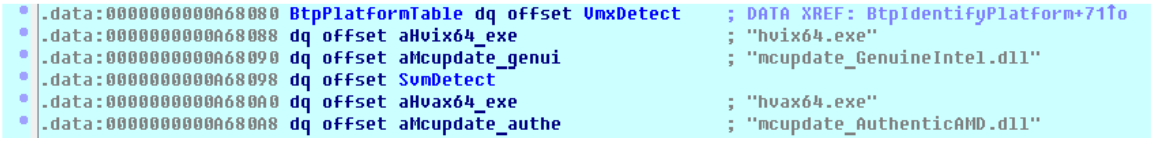
In order to properly debug hvloader.exe you can either load a previously created idb file, or cancel the current debugging session and reconnect. Uploading the File hvloader.idb hang IDA, so you will have to take advantage of the second option. This is done by replacing the first instruction of HvlMain to EB FE 90 that fixated the code and will provide an opportunity to restart IDA PRO, download hvloader.exe and reconnect gdb-debugger to VMware. You must then return the changed bytes in place and perform rebase module. To improve the speed of operations you can apply changes to code with simple scripts written in python (PatchHvLoader.py and RestoreHvLoader.py). Base load hvloader.exe does not change and always has been 0x971000, so that, by analogy with winload.exe once performed

rebase, the base remains, and on subsequent connections debugger module is located to the right address without performing additional operations.

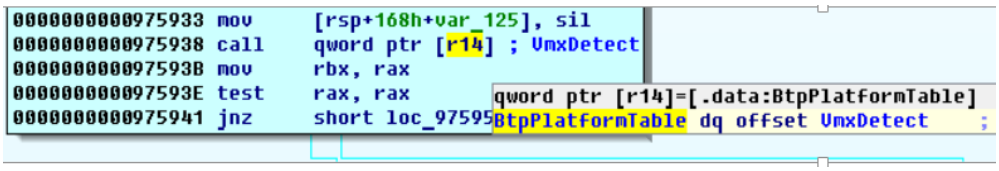
In hvloader.exe you should pay attention to the function BtPrepareHypervisorLaunch, which does basic operations for loading the hypervisor. Shortly before calling this function, you can see that the function BtLoadUpdateDll, which loads the library processor microcode updates mcupdate_GenuineIntel.dll. The functions BtLoadUpdateDll and BtPrepareHypervisorLaunch first performing BtpIdentityPlatform, which is determined by the manufacturer of the processor



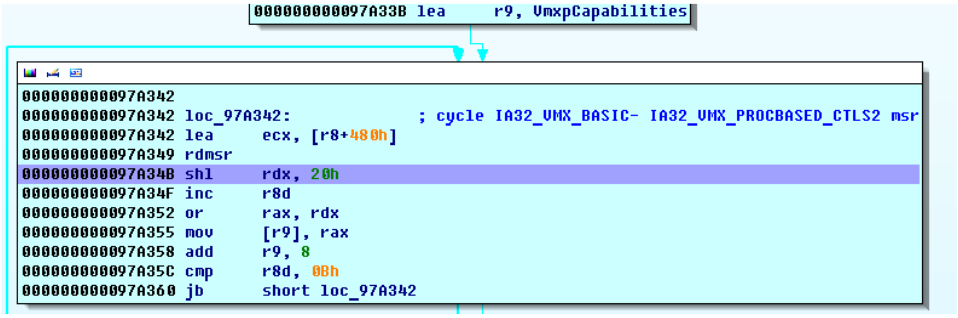
and returns a pointer to a structure BtpPlatformTable and the names of uploaded files.

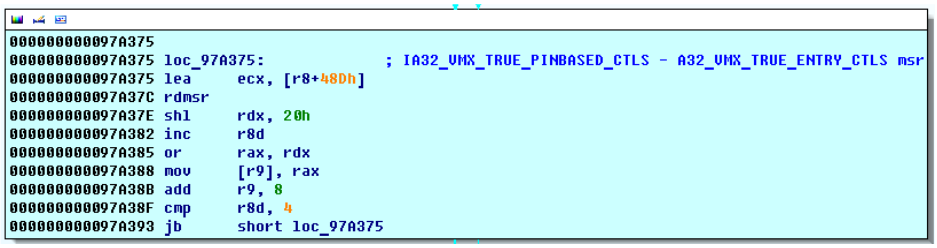


Pointers to function and VmxDetect SvmDetect needed only BtPrepareHypervisorLaunch. These functions are called immediately after BtpIdentityPlatform depending on the platform (VmxDetect for Intel and SvmDetect for AMD):



VmxDetect, for example, determines the capabilities of the processor





and returns a pointer to the next platform specific function VmxValidate (SvmDetect returns SvmValidate), etc.

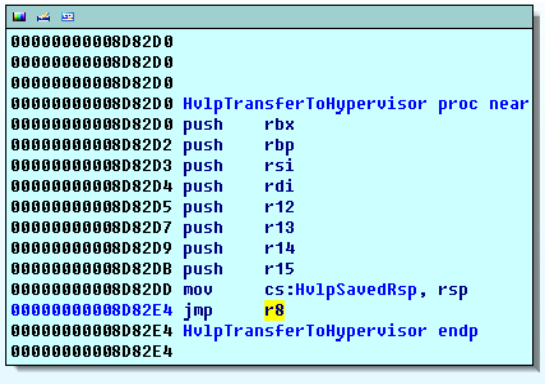
Additionally, attention may be drawn to the calculation of the random offset for the load adress of the hypervisor xFFFFF800 0000000000000000 and its subsequent displacement by calling BtpLayoutHvImage.

The structure BtpAllocateAndBuildLoaderBlock is filled with BtpLoaderBlockPages (aka HvlpLoaderBlock in winload.exe), which later will be used to transfer control to the start of the procedure hvix64.exe.

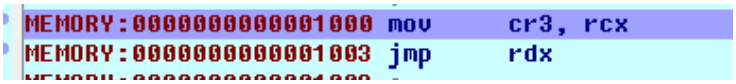
The Rebase Message Hv by: 6282000 shows the boot offset hypervisor on address 0xFFFFF800 0000000000000000. This shift will be needed at the moment we switch to IDA PRO debug with winload.exe on hvix64.exe

Back in the winload.exe

The function HvlpTransferToHypervisor made the transition to the start feature of hvix64.exe.



The Instruction jmp r8 transfers execution to the code located at the address specified in HvlpBelow1MbPage (0x1000)



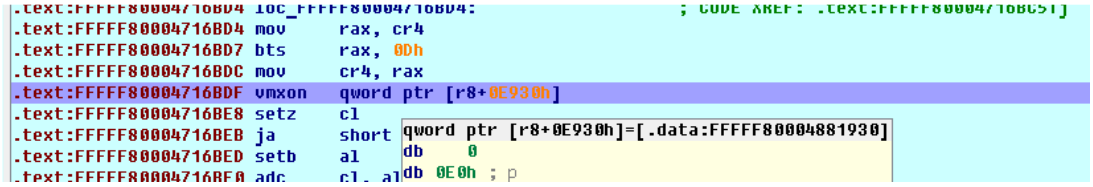
In a previous rdx the structure was placed by hvLoaderBlock address to the start of hvix64.exe

Later in IDA PRO you have to download hvix64.idb (similar to hvloader. exe), which works as follows:

- insert statement jmp $ (EB FE) at the start of the procedure start in hvix64. exe;
- completion debugging of winload.exe through the Debugger->the Detach from process;
- file download hvix64.exe in IDA PRO;
- connection to the gdb debugger vmware;
- restore the changed bytes to the original (0F 32);

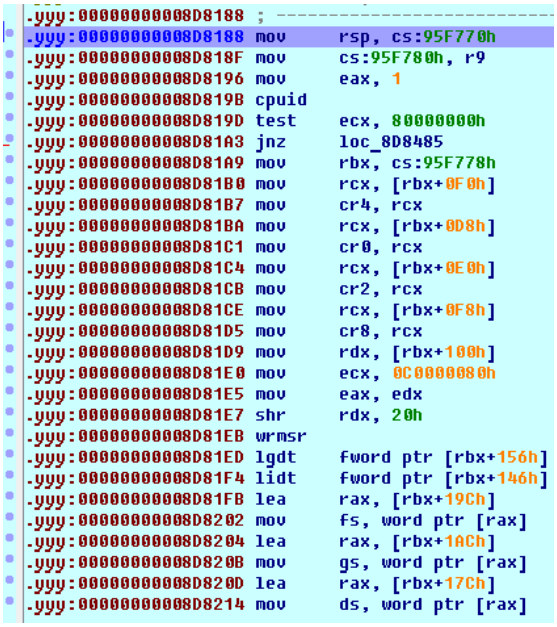
– performing the operation Edit -> Segment -> Rebase program indicating an Image Base 0xFFFFF0000000000000000 800 + value, which was issued by the debugger in the Rebase Hv by: 6282000.

Next quite a number of different operations as to be done in preparation for the execution of the vmxon hypervisor:

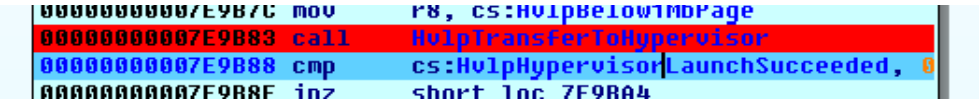


Then vmptrld, subsequent filling VMCS with necessary values and in the last instance it will start vmlaunch.

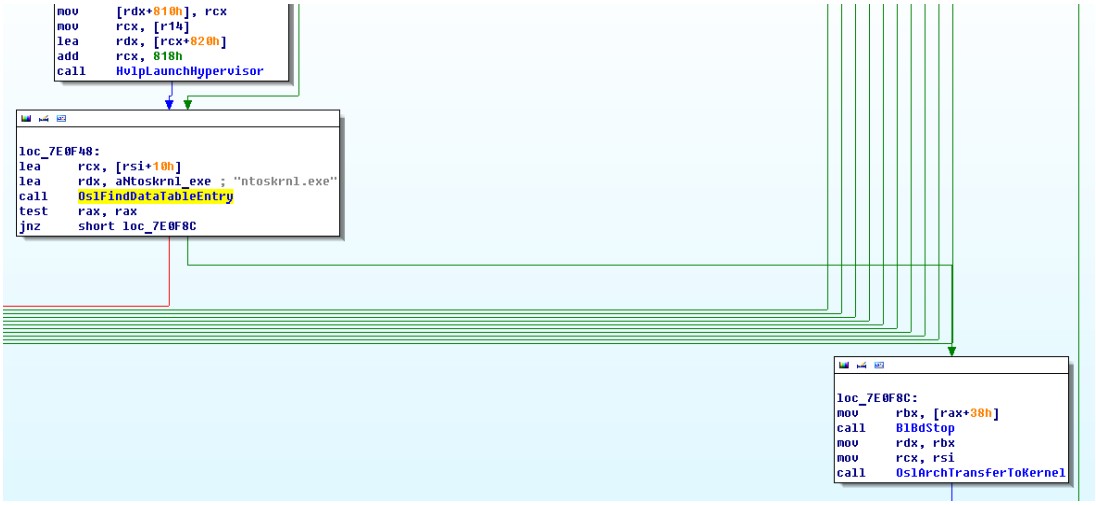
After vmlaunch gets into HvlpReturnFromHypervisor while debugging via GDB we will see that after the first instruction cpuid, calling VM exit, the transition is made directly to the HOST_RIP.



After Returning from the procedure, HvlpReturnFromHypervisor passes control to the next HvlpTransferToHypervisor for instructions.



at the end of the function HvlpLaunchHypervisor starts the kernel Windows through OslArchTransferToKernel.



If the Debugger is connected to the hypervisor, we can observe the following output(for the virtual system with two processors, each consists of two cores).

```
[0] Hypervisor initialized.
[0] Root Vp created.
MTRR map: number of ranges = 6 (default=UC)
Base=0x0000000000000000, Size=0x00000000000a0000, Type=WB, Synth=0
Base=0x00000000000a0000, Size=0x0000000000020000, Type=UC, Synth=0
Base=0x00000000000c0000, Size=0x000000000000c000, Type=WP, Synth=0
Base=0x00000000000cc000, Size=0x0000000000024000, Type=UC, Synth=0
Base=0x00000000000f0000, Size=0x0000000000010000, Type=WP, Synth=0
Base=0x0000000000100000, Size=0x00000000bff00000, Type=WB, Synth=0
----------------------
[0] Root Vp started.
[1] Root Vp created.
[1] Root Vp started.
[2] Root Vp created.
[2] Root Vp started.
[3] Root Vp created.
[3] Root Vp started.
MTRR map: number of ranges = 6 (default=UC)
Base=0x0000000000000000, Size=0x00000000000a0000, Type=WB, Synth=0
Base=0x00000000000a0000, Size=0x0000000000020000, Type=UC, Synth=0
Base=0x00000000000c0000, Size=0x000000000000c000, Type=WP, Synth=0
Base=0x00000000000cc000, Size=0x0000000000024000, Type=UC, Synth=0
Base=0x00000000000f0000, Size=0x0000000000010000, Type=WP, Synth=0
Base=0x0000000000100000, Size=0x00000000bff00000, Type=WB, Synth=0
```

It is worth mentioning that the process of loading a hypervisor in Windows Server 2012 differs significantly from Windows Server 2008R2, where the preparation and launch of the hypervisor directly produced by the hvboot.sys that run after loading the kernel Windows. This activation of the hypervisor instruction vmlaunch performed in the driver hvboot.sys and the next VM exit was processed in the hvix64.exe.

Find symbol information

When loading hvix64.exe in IDA PRO we get about three thousand functions with names like sub_FFFFF8000XXXXXbecause Microsoft, unfortunately, does not provide the symbol information for the hypervisor. facilitate the research of the hypervisor can first try to identify some of the functions without detailed study.

In the first place it is worth using bindiff (or diaphora) to compare the files hvix 64.exe, hvloader. exe and winload . exe where symbol information are provided. Comparison shows that the networking

function (e,1000_), USB , cryptography and some other features are exactly the same as the ones that are present in winload.exe. This will help set the appointment of 500 functions. The same bindiff allows you to move the names of matching functions from one database to another idb. However, this method should be taken with caution and do not move all fully matched functions. At least the result should be analyzed by Visual comparison graph matching functions (Ctrl + E).

Next, let's define exception/interrupt functions, which are standard for processor architecture x86. A little script is written in python (ParseIDT.py) to parse the IDT, which must be run in IDA PRO, beeing connected through a debugging module of WinDBG to the hypervisor.

In the case of ISR was not found, check the tab List of problems in IDA PRO, since these procedures can not be found in the automatic analysis code that IDA performs.

Next, you can define the exit procedure in VM after reading field values VMCS. This can be done after the procedure fill the VMCS at hvix64. exe or use this script display-vmcs.py, which in the context of the hypervisor reads all fields VMCS and prints their values.

Hypercall

Microsoft released document Hypervisor Top-Level Functional Specification: Windows Server 2012 (6), describes the principles of Hyper-V 3.0.

Each virtual machine, as well as directly with the OS component installed Hyper - (V) is presented in terms of the partition (partition). each section has its own identifier that must be unique to the host server.

For each section are given privileges to create (structure HV_PARTITION_PRIVILEGE_MASK), which determine the ability to perform specific hypercall.

Learn privileges by executing in the root-partition the following code in ring0:

```
WinHvGetPartitionId(&PartID);//PartID – ID section
WinHvGetPartitionProperty(PartID,HvPartitionPropertyPrivilegeFlags,&HvProp);// the result is returned in HvProp.
```

HvPartitionPropertyPrivilegeFlags– One of the enumeration values HV_PARTITION_PROPERTY_CODE, which operate functions exported driver winhv.sys.

```
HV_STATUS
WinHvGetPartitionProperty(
    __in  HV_PARTITION_ID    PartitionId,
    __in  HV_PARTITION_PROPERTY_CODE    PropertyCode,
    __out PHV_PARTITION_PROPERTY PropertyValue
);
```

Also, if necessary, these privileges can be changed, causing root-partition in the following function:

```
HV_STATUS
WinHvSetPartitionProperty(
    __in HV_PARTITION_ID    PartitionId,
    __in HV_PARTITION_PROPERTY_CODE    PropertyCode,
```

```
    __in HV_PARTITION_PROPERTY     PropertyValue
);
```

The value of HvPartitionPropertyPrivilegeFlags for the root partition: 000039FF00001FFF

AccessVpRunTimeMsr	AccessGuestIdleMsr	ConnectPort
AccessPartitionReferenceCounter	AccessFrequencyMsrs	AccessStats
AccessSynicMsrs	AccessDebugMsrs	Debugging
AccessSyntheticTimerMsrs	CreatePartitions	CpuManagement
AccessApicMsrs	AccessPartitionId	ConfigureProfiler
AccessHypercallMsrs	AccessMemoryPool	
AccessVpIndex	AdjustMessageBuffers	
AccessResetMsr	PostMessages	
AccessStatsMsr	SignalEvents	
AccessPartitionReferenceTsc	CreatePort	

The value of HvPartitionPropertyPrivilegeFlags for child partition 000008B000000E7F:

AccessVpRunTimeMsr	AccessPartitionReferenceTsc
AccessPartitionReferenceCounter	AccessGuestIdleMsr
AccessSynicMsrs	AccessFrequencyMsrs
AccessSyntheticTimerMsrs	PostMessages
AccessApicMsrs	SignalEvents
AccessHypercallMsrs	ConnectPort
AccessVpIndex	Debugging

In a Windows guest OS, privileges can be obtained by placing EAX 0x40000003 and following the instructions CPUID (in document Hypervisor Functional Specification top-level 3.0 a given interpretation of the results of the cpuid).

CPUID 40000003 called

EAX = 00000E7F (00001110 01111111)

Bit 0: VP Runtime (HV_X64_MSR_VP_RUNTIME)

Bit 1: Partition Reference Counter (HV_X64_MSR_TIME_REF_COUNT)

Bit 2: Basic SynIC MSRs (HV_X64_MSR_SCONTROL through HV_X64_MSR_EOM and HV_X64_MSR_SINT0 through HV_X64_MSR_SINT15)

Bit 3: Synthetic Timer MSRs (HV_X64_MSR_STIMER0_CONFIG through HV_X64_MSR_STIMER3_COUNT)

Bit 4: APIC access MSRs (HV_X64_MSR_EOI, HV_X64_MSR_ICR and HV_X64_MSR_TPR)

Bit 5: Hypercall MSRs (HV_X64_MSR_GUEST_OS_ID and HV_X64_MSR_HYPERCALL)

Bit 6: Access virtual processor index MSR (HV_X64_MSR_VP_INDEX)

EBX = 000008B0 (00001000 10110000)

Bit 4: PostMessages

Bit 5: SignalEvents

Bit 7: ConnectPort

Bit 11: Debugging

ECX = 00000002 (00000000 00000010)

Maximum Processor Power State is C2

EDX = 000007B2 (00000111 10110010)

Bit 1: Guest debugging support is available

Bit 4: Support for passing hypercall input parameter block via XMM registers is available

Bit 5: Support for a virtual guest idle state is available

The hypervisor privileges section, which carried out the operation that caused the VM exit, can be obtained by calculating the value of gs: 0, read the value of the field in the VMCS HOST_GS_BASE or IA32_GS_BASE MSR:

```
WINDBG>rdmsr 0xc0000101
msr[c0000101] = fffff800`05464000
```

then get the value pointed to gs: 82e8, and go to the offset 0xd8.

WINDBG>dc poi(fffff800`05464000+82e8)+0xd8
00000080`04dd70d8 **00001fff 000039ff** 00000000 ffffe800 .....9.....
00000080`04dd70e8 00000001 00000000 00000000 00000000 ....

In this case, the VM exit was made from root-partition.

The hypervisor in each section forms a special page to run hypercall. Its address can be obtained by reading MSR 0x40000001 (HV_X64_MSR_HYPERCALL):

kd> rdmsr 0x40000001

msr[40000001] = 00000000`1ffb1001

kd> !dc 00000000`1ffb1001

#1ffb1000 c3c1010f 90909090 90909090 90909090 ....
#1ffb1010 90909090 90909090 90909090 90909090 ....
As you can see, 0xc3c1010f - instructs opcodes to vmcall; ret

Windows Server 2012 following changes took place in the export of the driver winhv.sys in comparison with the Windows Server 2008 R2:

Добавлено	Удалено
WinHvAddLogicalProcessor	WinHvOnInterrupt
WinHvAttachDevice	WinHvReclaimInterruptVector
WinHvDetachDevice	WinHvSupplyInterruptVector
WinHvGetLogicalProcessorProperty	
WinHvGetLogicalProcessorRegisters	
WinHvGetNextQueuedPort	
WinHvGetSystemInformation	
WinHvInjectSyntheticMachineCheckEvent	
WinHvMapDeviceInterrupt	
WinHvPrepareForSleep	
WinHvProcessorIndexToLpIndex	
WinHvProcessorNumberToVpIndex	
WinHvRemoveLogicalProcessor	
WinHvSetLogicalProcessorProperty	
WinHvSetLogicalProcessorRegisters	
WinHvUnmapDeviceInterrupt	

In order to be able to use the export function winhv.sys can either dynamically calculate the addresses of the functions (7), or to create a lib-file (8). Consider the second option.

When you declare functions like stdcall (32-bit version of the driver) in the def-file, you must specify the ordinals of the functions or when loading the driver the imported functions will not be found (for some reason, the table import function hyperv3.sys driver gets a postfix @ number, even if the def-file register WinHvGetPartitionProperty @ 16 = WinHvGetPartitionProperty):

WinHvGetPartitionProperty@16 @42

To create a def-file using the output of dumpbin:

dumpbin /exports winhv.sys

(The Windows Server 2012 R2 is using a winhvr.sys driver root-section, so the def-file for the driver in the OS is necessary to form it).

To build a 64-bit driver you do not need to make any changes.

After editing the def-file it must be re-form the lib-file with the command (for x86):

"C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\bin\lib.exe" /def:D:\hyperv3\winhv.def /OUT:D:\ hyperv3\winhv.lib /machine:x86

Для x64 (выполняется 1 раз для конкретной версии winhv.sys):

"C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\bin\amd64\lib.exe" /def:D:\hyperv3\winhv64.def /OUT:D:\hyperv3\winhv64.lib /machine:x64

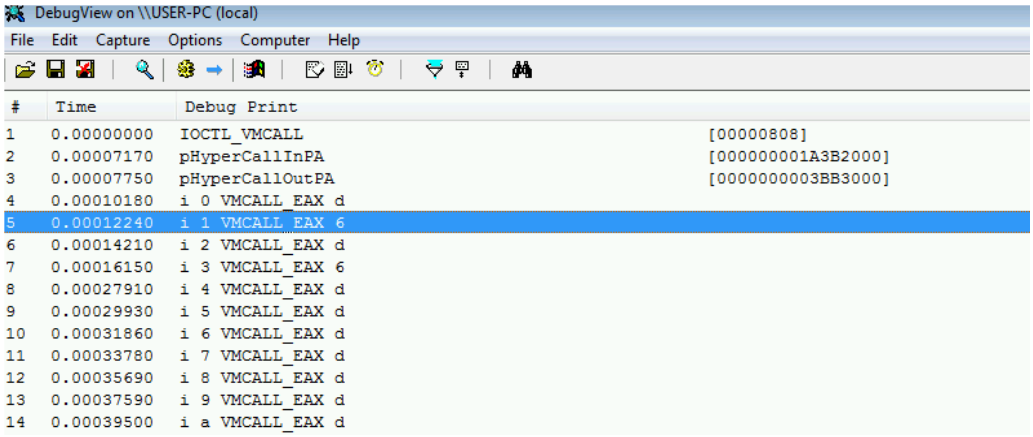
For x64 (performed 1 time for a specific version winhv.sys):

Let's try it in a loop from 0 to 0 x 100 consistently meet Hypercall 0 x 41 (HvInitializePartition), with the PartitionID in ECX, equal to the value of the loop iterator, with Fast bit (to pass parameters through the registers.) with EAX returns the output of the hypervisor.

```
for (i = 0x0; i <=0x100; i++)
{
	DbgPrintEx(DPFLTR_IHVDRIVER_ID, DBG_PRINT_LEVEL,"i %x VMCALL_EAX %x",i,ARCH_VMCALL_REG_MOD(i));
}
```

```
ARCH_VMCALL_REG_MOD PROC param1:DWORD
	push esi
	push edi
	push ebx
	xor edx,edx
	mov ecx, param1
	xor ebx,ebx
	xor esi,esi
	xor edi,edi
	mov eax, 10041h
	vmcall
	pop ebx
	pop edi
	pop esi
	ret
ARCH_VMCALL_REG_MOD ENDP
```

As a result, we obtain



In case if in the ecx was transferred to the active virtual machine PartitionID, the hypervisor returns-6 (HV_STATUS_ACCESS_DENIED), in other cases - d (HV_STATUS_INVALID_PARTITION_ID). Taking advantage of this fact, and the fact that the ID of each new section is calculated by simple adding 1 to the ID of the previous section, and the ID root-partition is always equal to 1, you can set the number of active virtual machines on the host. To do this, slightly modify the code for the driver:

```
for (i = 0x2; i <=0x10000; i++)
{
	res = ARCH_VMCALL_REG_MOD(i);
	if (res == HV_STATUS_INVALID_PARTITION_ID){
```

```
        DbgPrintEx(DPFLTR_IHVDRIVER_ID, DBG_PRINT_LEVEL,"PartitionID %x VMCALL_EAX %x \n",i,res);
        }
}

DbgPrintEx(DPFLTR_IHVDRIVER_ID,  DBG_PRINT_LEVEL,"Number  of  active  virtual  machines:  %x
\n",counter);
```

and get a list of active sections ID and number:

```
4    0.00017750   PartitionID 3 VMCALL_EAX 6
5    0.00019740   PartitionID 4 VMCALL_EAX 6
6    0.52050769   Number of active virtual machines: 2
```

The number of loop iterations must be greater than the number of running VMs + number of overloaded since the start VM hypervisor. After restarting the hypervisor numbering of all sections begins again.

These data are available for the following two reasons:

- The section PartitionID generated by simply adding 1 to the last used PartitionID.
- When processing a hypercall the hypervisor first checks the validity of the transferred PartitionID and just in case whats the referred PartitionID active partition, it checks the rights to perform hypercall.

This feature hypervisor can be used to determine the number of virtual machines running on a given host server. For the name of the host server, you can peek in the registry of the guest OS under HKLM \ Software \ Microsoft \ Virtual Machine \ Guest \ Parameter, which contains data on the host operating system, transmitted by Key Value Pair Integration Component, which is normally enabled by default. Also controlled restarting the virtual machine on the second Monday of the month and secure it PartitionID (there is quite a high probability that he will be the last in the list of active VM), you can determine whether a virtual neighbors on their servers coming out every second Tuesday security fixes. However, the reality is quite difficult to imagine that someone will need this information ...

This hypervisor behavior could be observed in the assembly 6.3.9431.0 (Windows Server 2012 R2 Preview), but Microsoft recognized this behavior as "unexpected behavior" and eliminated him in the assembly 6.3.9600.16384 ". the TLFS changes were made to allow for the enforcement of such hypercall behavior only from root-partition.

The Statement which is processing vmcall in the hypervisor runs roughly as follows:

- check ring protection in which the statement has been issued, if the statement was executed in ring 3, then processing stops;
- if the instruction is executed in ring0, it checks, whether at the same processor LongMode.
- depending on the operating mode of the processor to perform two different procedures, the logic is quite similar;
- each procedure loads a pointer to an array of structures that contain the parameters necessary for processing each of hypercall 0 to 8C (decryption codes listed in hypercall Hypervisor Top-Level Functional Specification: Windows Server 2012. Appendix B: Hypercall Code Reference). One of the elements of each structure is a pointer to a procedure for processing hypercall:

```
mVmcallHandlersTable dq offset HvCallReserved00
                                        ; DATA XREF: sub_FFFFF80006100074+F0D↑o
                                        ; mHandle_VMCALL+BA↑o
dw 0
dw 0
dw 0
dw 0
dw 0
dw 0
dw 40h
dw 0
dq offset HvSwitchVirtualAddressSpace   ; address of specific hypercall handler
dw 1                                    ; VMCALL ID
dw 0                                    ; REP CALL
dw 8                                    ; size of hypercall input param (in bytes)  without rep prefix
dw 0                                    ; size of hypercall input param with rep prefix
dw 0                                    ; hypercall output 1 element param size without rep prefix
dw 0                                    ; hypercall output 1 element param size with rep prefix
dw 43h                                  ; group number of hypercall (f.e Virtual Interrupt Interfaces)
                                        ; used like index in table of statistics of hypercall using
dw 0
dq offset HvFlushVirtualAddressSpace
dw 2
dw 0
dw 18h
```

- then there is a check which way the hypervisor have been transferred parameters through memory or through the registers (in this case, the fast call bit in EAX before hypercall should equal 1).
- then call the corresponding function..

For comparison, some of the important fields VMCS were obtained by using the script display - vmcs.py after VM exit:

Root partition	Child partition
CPU_BASED_VM_EXEC_CONTROL = 0xb6206dfa Use TSC offsetting HLT exiting MWAIT exiting RDPMC exiting Use TPR shadow Use I/O bitmaps Use MSR bitmaps MONITOR exiting Activate secondary controls IO_BITMAP_A = 0x4e06000 IO_BITMAP_A_HIGH = 0x0 IO_BITMAP_B = 0x4e07000 IO_BITMAP_B_HIGH = 0x0 EXCEPTION_BITMAP = 0x40000 MSR_BITMAP = 0x4e08000 MSR_BITMAP_HIGH = 0x0 PIN_BASED_VM_EXEC_CONTROL = 0x1f External-interrupt exiting NMI exiting SECONDARY_VM_EXEC_CONTROL = 0x2a Enable EPT Enable RDTSCP Enable VPID VM_ENTRY_CONTROLS = 0x13ff Load debug controls IA-32e mode guest VM_EXIT_CONTROLS = 0x3efff Save debug controls Host address space size Acknowledge interrupt on exit	CPU_BASED_VM_EXEC_CONTROL = 0xb5a06dfa Use TSC offsetting HLT exiting MWAIT exiting RDPMC exiting Use TPR shadow MOV-DR exiting Unconditional I/O exiting Use MSR bitmaps MONITOR exiting Activate secondary controls CR0_GUEST_HOST_MASK = 0xfffffe1 CR0_READ_SHADOW = 0x8001003b CR4_GUEST_HOST_MASK = 0xfffff874 CR4_READ_SHADOW = 0x406f8 EXCEPTION_BITMAP = 0x40000 GUEST_CR0 = 0x8001003b GUEST_CR3 = 0x185000 GUEST_CR4 = 0x426f9 GUEST_RIP = 0x839b1000 GUEST_RSP = 0x8870f8a4 HOST_CR0 = 0x80010031 PIN_BASED_VM_EXEC_CONTROL = 0x1f External-interrupt exiting NMI exiting SECONDARY_VM_EXEC_CONTROL = 0x62 Enable EPT Enable VPID WBINVD exiting VM_ENTRY_CONTROLS = 0x11ff Load debug controls VM_EXIT_CONTROLS = 0x3efff Save debug controls Host address space size Acknowledge interrupt on exit

For example , you can see, that for guest-partition the hypervisor handles all input/output (I/O exiting Unconditional), and for the root partition monitors only certain ports (Use I/O bitmaps).

```
WINDBG>!dc 0x4e06000 L250 - IO_BITMAP_A
# 4e06000 00000000 00000003 00000000 00000010 ................
# 4e06010 00000000 00000003 00000000 00000000 ................
# 4e06020 00000000 00000000 00000000 00000000 ................
……
# 4e06190 00000000 00000000 00000000 f1000000 ................
```

If I am not mistaken in the calculations, then the root-partition monitored ports are 20h, 21h, 44h, A0h, A1h, 1D5Fh, 1D64h, 1D65h, 1D66h, 1D67h.

closing

The article describes the steps that must be done to create a stand for the research of Hyper-V, and very briefly describes some aspects of the work of the hypervisor. I hope this information is useful for beginners in hypervisor security researcher at Microsoft.

Sources:

1. http://msdn.microsoft.com/en-us/library/Windows/hardware/ff540654(v=vs.85).aspx
2. http://msdn.microsoft.com/en-us/library/cc768520%28v=bts.10%29.aspx
3. http://en.community.dell.com/techcenter/virtualization/w/wiki/3029.aspx
4. http://www.hhdsoftware.com/Downloads/free-virtual-serial-ports
5. http://ww.osronline.com/showthread.cfm?link=234398
6. http://www.microsoft.com/en-us/download/details.aspx?id=39289
7. http://alter.org.ua/docs/nt_kernel/procaddr/
8. http://www.osronline.com/showthread.cfm?link=132065
9. http://blog.cr4.sh/2012/07/vmware-gdb-stub-ida.html