Exploiting Reversing (ER) series: Article 05 | Hyper-V (part 01)

(a step-by-step security research series on Win, macOS, hypervisors, browsers, and others)

by Alexandre Borges
release date: MAR/12/2025 | rev: A.1

0. Quote

"Remember Red, hope is a good thing, maybe the best of things, and no good thing ever dies."
(Andy Dufresne played by Tim Robbins | "The Shawshank Redemption" movie - 1994)

1. Introduction

Welcome to the fifth article of **Exploiting Reversing (ER) series**, **a step-by-step vulnerability research series on Windows, macOS, hypervisors, browsers, and others**, where we will review concepts, architecture and practical steps related to vulnerability research. My last articles are listed below:

- **ERS_04:** https://exploitreversing.com/2025/02/04/exploiting-reversing-er-series-article-04/
- **ERS_03:** https://exploitreversing.com/2025/01/22/exploiting-reversing-er-series-article-03/
- **ERS_02:** https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/
- **ERS_01:** https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/
- **MAS_10**: https://exploitreversing.com/2025/01/15/malware-analysis-series-mas-article-10/
- **MAS_09:** https://exploitreversing.com/2025/01/08/malware-analysis-series-mas-article-09/
- **MAS_08:** https://exploitreversing.com/2024/08/07/malware-analysis-series-mas-article-08/
- **MAS_07:** https://exploitreversing.com/2023/01/05/malware-analysis-series-mas-article-7/
- **MAS_06:** https://exploitreversing.com/2022/11/24/malware-analysis-series-mas-article-6/
- **MAS_05:** https://exploitreversing.com/2022/09/14/malware-analysis-series-mas-article-5/
- **MAS_04:** https://exploitreversing.com/2022/05/12/malware-analysis-series-mas-article-4/
- **MAS_03:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/
- **MAS_02:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_01**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/

This is the first article of a multi-part series of hypervisors' articles. Hypervisors are really wide scope, this text is quite an **introductory article,** so nothing new will be presented and the most basic goal is getting a better understanding of hypervisor's concepts at this moment. Therefore, I have chosen Hyper-V, from Microsoft, because it is pretty accessible for any professional who runs Windows systems.

2. Acknowledgments

The year is 2025. Four years ago, I started drafting articles with the sole purpose of helping the hacking and information security community, and as I could already imagine at that time, it would be challenging to find time to continue producing content, and indeed this side effect has been confirmed over time. As I have been using IDA Pro for a long time, I needed a license of my own, and that was when **Ilfak Guilfanov (@ilfak)** and **Hex-Rays SA (@HexRaysSA)** decided to help me, and since then they have provided continuous and decisive support to write this **Malware Analysis Series (MAS)**, which is focused on malware analysis, and the **Exploiting Reversing series (ERs)**, which is my **current and long-term series** on internals, vulnerability research and, eventually, exploitation in critical topics such as Windows, kernel drivers, macOS, browser and hypervisors.

Time flies, and companies around the world have become more demanding in terms of technical skills, but I still believe that one of the most effective ways to help these professionals is to write articles because such content can offer a solid method to learn details that would be a bit more difficult in live conferences or even other media. I still face serious time constraints to write, but I keep trying to do it because, in some way, I know that these series have been important for people's careers.

Life may be short, but every moment is worth it. Enjoy the journey and keep exploiting it!

3. Lab infrastructure

This article demands the following environment:

- A **physical and/or a virtual machine running Windows 11**, where this physical or virtual machine might have one or two network interface cards.
- **IDA Pro or IDA Home version (@HexRaysSA):** https://hex-rays.com/ida-pro/ . Readers might use Binary Ninja, Ghidra and other ones, but I will be using IDA Pro and its decompiler in this article.
- As plugins for **IDA Pro**, reader can install two products to analyze binary patches, but we will not necessarily use them right now:
    - **BinDiff:** https://github.com/google/bindiff/releases/tag/v8
    - **Diaphora:** https://github.com/joxeankoret/diaphora
- Install **Windows SDK + Visual Studio + Windows Development Kit (optionally)** in both host and virtual machines:
    - **Visual Studio:** https://visualstudio.microsoft.com/downloads/. During the installation, don't forget to install "Desktop development with C++" set.
    - **Windows SDK:** https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/
    - **Windows Development Kit (WDK):** https://learn.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk

4. Lab configuration

One of the first steps for researching Hyper-V is to set up a functional debugging environment and even though it is not a challenging task, there are many intricate details that might be important before obtaining a working environment, which will be useful for correlating dynamic execution with static analysis. In this article I will be using VMware Workstation 17.6, which is a product from Broadcom, and is offered in versions for home users and commercial use.

4.1 Kernel Debugging

First of all, readers can skip this section. My only intention here is to show a simple kernel debugging environment and whether everything goes well here then we will work with Kernel and Hyper-V debugging at the same time, even though we will use a different strategy and approach.

These steps are valid for an environment composed of either one physical system (host) communicating with a virtual machine (target) or two virtual machines, which one of them acts as a host and the other one is the target. In my specific case, I am adopting the second scenario (a physical host debugging a virtual machine), and both systems are running Windows 11 Pro edition. Obviously, there are subtleties that distinguish one case from another, and may make the difference at the end. Windows kernel can be debugged through a network connection, USB and serial connection and as expected, the network approach is usually the preferred way. As reference, the network configuration involved in this text is:

- **host:** 192.168.0.96
- **virtual machine (Ethernet_01):** IP address from DHCP (NAT interface)
- **virtual machine (Ethernet_02):** IP address from DHCP (Host only interface)

The IP addresses of network interfaces can be retrieved from a PowerShell terminal:

- **Get-NetIPAddress -AddressFamily IPv4 | Select-Object IPAddress, InterfaceAlias | ft -AutoSize**

```
PS C:\> Get-NetIPAddress -AddressFamily IPv4 | Select-Object IPAddress, InterfaceAlias | ft -AutoSize

IPAddress       InterfaceAlias
---------       --------------
172.28.80.1     vEthernet (Default Switch)
192.168.159.131 Ethernet01
127.0.0.1       Loopback Pseudo-Interface 1
```

[Figure 01]: Network Interfaces' IP addresses

it is quite important to highlight these points:

- Use an **NAT type interface** and not Bridge type to avoid any communication issue.
- Use **DHCP address** and not fixed address.

You should try to ping from host (debugger) to virtual machine (debugged), and vice-versa, to guarantee that everything is working well. Additionally, two further details can be relevant on both systems:

- Check whether the **Windows Firewall** is blocking the connection.

- Optionally, go to **Settings > Network & Internet > Advanced network settings > Advanced sharing settings**, and enable **Network Discovery**.

There are multiple ways to setup kernel debugging through network, and all of them do the job. Probably using KDNET is the easiest and most recommended way, and can be performed by executing the following steps:

On the target

- **mkdir C:\kdnet**
- **copy "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\kdnet.exe" C:\kdnet**
- **copy "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\VerifiedNICList.xml" C:\kdnet**
- **cd C:\kdnet**
- **kdnet (***check supported network interfaces)*
- **bcdedit /set key 1.2.3.4**
- **kdnet 192.168.0.96 50001 -k**

One of great advantages of KDNET is that it adjusts the debugging settings with busparams automatically. Another especially useful feature of KDNET is that it offers options to debug the **kernel (k), hypervisor (h), bootmanager (b)** and **winload (w).** Thus, we can enable more than one option at the same time, like **-kh** that enabled kernel and hypervisor debugging at the same time. One disadvantage is that it does not allow to set directly the key to authorize the communication, and if you have never executed other **bcdedit** command, it is likely that kdnet will return a long key, which is not necessary for our tests, and that is the reason why I used **bcdedit** command to set it explicitly, which reaches the same results and requires almost identical settings. You will need to get the network hardware information:

- PowerShell: **Get-NetAdapterHarwareInfo**
- Executing **kdnet.exe**
- Checking **Device Manager**

Using PowerShell, the network hardware information can be viewed below:

```
PS C:\> Get-NetAdapterHardwareInfo

Name                           Segment Bus Device Function Slot NumaNode PcieLinkSpeed PcieLinkWidth Version
----                           ------- --- ------ -------- ---- -------- ------------- ------------- -------
Ethernet00                           0   3      0        0  160                5.0 GT/s            32 1.1+
Ethernet02                           0  27      0        0  256                5.0 GT/s            32 1.1+
```

[Figure 02]: PowerShell: getting network adapter information

To set up kernel debugging using **bcdedit**, run the following steps:

- **bcdedit /debug on**
- **bcdedit /dbgsettings net hostip:192.168.0.96 port:50001 busparams:3.0.0 key:1.2.3.4**
- **bcdedit /dbgsettings** *(check the changes)*

As readers can realize, it is almost identical to the previous procedure, but we do not have “ready” options to setup **kernel, winload** or **bootmanager** debugging, and if it was needed then we should do it manually

too. On both systems (host and target), I strongly recommend you configure the following system environment variable:

- **_NT_SYMBOL_PATH=srv*c:\symbols*https://msdl.microsoft.com/download/symbols**

Create this system environment variable by setting it at **Advanced Windows Setting > Environment Variables** and creating the **_NT_SYMBOL_PATH** as explained above.

- **On the host: windbg -k net:port=50001,key=1.2.3.4**

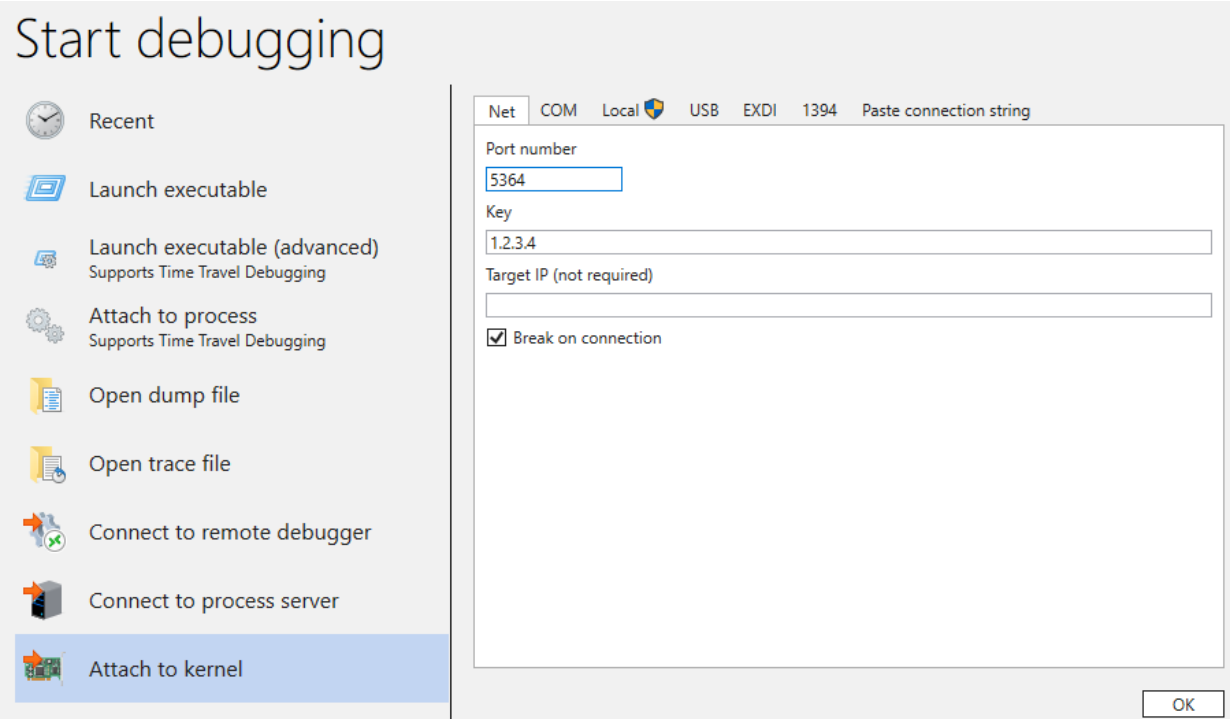
Finally, reboot the target (virtual machine) by running **shutdown -r -t 0.** If everything went well, readers should see an output similar to the following one and, if the debugger does not stop, you can pick up to **Debug > Break** :

```
Microsoft (R) Windows Debugger Version 10.0.26100.1 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target 192.168.0.96 on port 50001 on local IP 192.168.0.96.
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 26100 x64 target at (Thu Sep 12 21:47:50.771 2024 (UTC - 3:00)), ptr64 TRUE
Kernel Debugger connection established.

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*C:\symbols*https://msdl.microsoft.com/download/symbols
Symbol search path is: srv*C:\symbols*https://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 26100 MP (2 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Edition build lab: 26100.1.amd64fre.ge_release.240331-1435
Kernel base = 0xfffff802`e0a00000 PsLoadedModuleList = 0xfffff802`e18f4730
Debug session time: Thu Sep 12 21:47:50.611 2024 (UTC - 3:00)
System Uptime: 0 days 0:11:07.532
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run console kernel debugger) or,                       *
*       CTRL+BREAK (if you run GUI kernel debugger),                          *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!DbgBreakPointWithStatus:
fffff802`e0eb9990 cc              int     3
```

[Figure 03]: Kernel debugging session

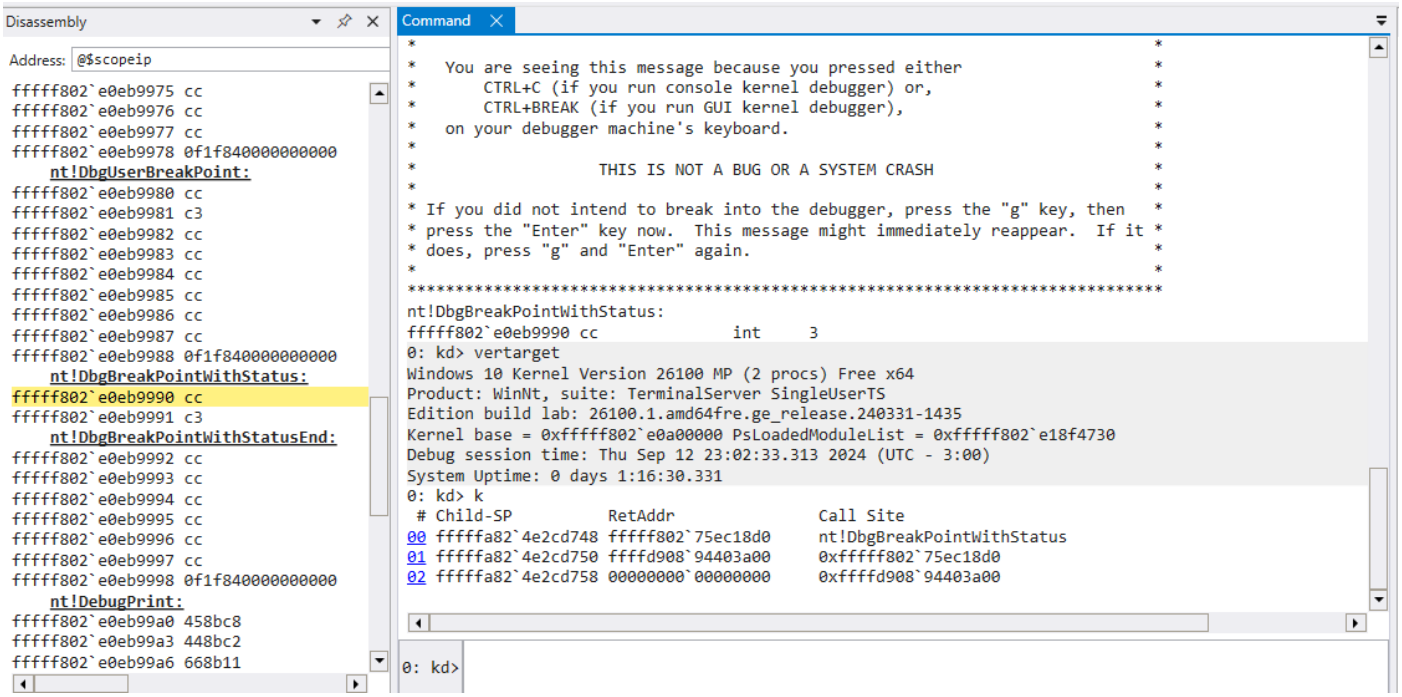
That has worked! To resume the virtual machine execution and exit from WinDbg, type **qd**. Probably WinDbg will exit, and you will have to log on to the virtual machine again, but everything should be working well. If you receive an error by executing the command above, try to change the used port (in this case 5364) because there could be something already running on this specific port. If there is any communication between the host and the target, check firewall rules and, in special, search for “Windows GUI Symbolic Debugger” and/or “Windows Kernel Debugger” in Inbound Rules. If it is blocked, allow it. Personally, I prefer to use the new version of WinDbg (previously named “WinDbg Preview”), which can be retrieved from https://aka.ms/windbg/download or easily by executing **winget install Microsoft.WinDbg**

Once you open WinDbg, go to **File | Attach to Kernel | Net** tab and type the port (50001), key (1.2.3.4) and, optionally, the target IP address (192.168.0.96 in my case):



[Figure 04]: WinDbg setup

Likely Windows Firewall will pop up a message asking authorization to connect and, as readers already know, the recent version of WinDbg is absurdly better than the previous one:



[Figure 05]: WinDbg: remote kernel debugging

At the same way, we can detach and resume the target virtual machine execution by executing **g** (go) and closing the WinDbg. Now that everything is working well, it is time to move on and configure the Hyper-V debug environment.

4.2 Kernel and Hyper-V debugging (hvix64)

There are multiple methods to debug Hyper-V, and choosing a preferred method depends on the context and professional point of view. Anyway, this setup is completely independent of the previous section, and you can assume that it is a different experiment.

In terms of hardware configuration, our options are:

- **Use two physical hosts.**
- **Use a physical host and one virtual machine.**
- **Use two virtual machines.**

The first option, which uses two physical hosts running Windows 11, would be the best choice, by far. However, as many readers might not have two physical systems available only for performing these tests, I am taking the second option and using a physical host (running Windows 11 Pro) and a virtual machine (also using a Windows 11 Pro).

Everything could seem too easy, but as we learn from the popular quote, evil is in the details. If your system is running Hyper-V virtual machines, your only alternative would be to establish **serial communication** between the physical host and the virtual machine. While it works really well, it could not be as fast as you would like. Actually, I rarely configure serial debugging.

At the time I am drafting this article, Broadcom company made VMware Workstation Pro freely available for home users, and I will be using **VMware Workstation Pro 17.5** for running the target virtual machine. In this case, **network debugging between the physical host and the virtual machine is available and it is much better,** so all next steps will be using this option. In a simplified way, we need to build the following environment:

- **physical host: L0**
  - **first virtual machine running Windows 11 (VMware): L1**
    - **second virtual machine running Windows 11 (Hyper-V): L2**

We will have **Windows 11** installed at all levels, where (L1 -- level one) is a VMware virtual machine and within it there is a Hyper-V virtual machine running (L2 – level two). The full list of requirements follows:

- **physical host system (L0)**:
  - Windows 11 with VMware Workstation 17.x.x installed.
  - IP address 192.168.0.96 (your IP address is obviously different).

- **first virtual machine (VMware):**

    - Windows 11, with 8 or 16 GB RAM, 4 processors and TPM.
    - Virtualization enabled: mark Virtualize Intel VT-x/EPT or AMD-V/RVI option.
    - Secure Boot disabled (it is a particularly crucial step)
    - first network interface: NAT (DHCP)
    - second network interface: Host-Only (DHCP)

- **second virtual machine (Hyper-V, nested in the VMware virtual machine)**:

    - Windows 11, with 4 GB RAM and 1 processor.

In the **first virtual machine (L1)**, which will run the second virtual machine inside (**L2** -- nested) and to make it work and depending on your system you could need to edit its configuration file (**.vmx** extension) to include one or more of the following lines to guarantee that the nested virtual machine (**L2** – Hyper-V) will be able to start:

- **hypervisor.cpuid.v0 = "FALSE"**
- **vhv.enable = "TRUE"**

Now it is time to configure the debug environment and, as we will use **bcdedit.exe** command multiple times, and its syntax is the following one:

bcdedit /hypervisorsettings [<debugtype> [DEBUGPORT:<port>] [BAUDRATE:<baud>] [CHANNEL:<channel>] [HOSTIP:<ip>] [PORT:<port>] [BUSPARAMS:<Bus.Device.Function>]]

From the syntax above, we have the following settings:

- **debugtype:** net (for network connection)
- **hostip:** specify the Ipv4 address from the host debugger.
- **port:** specify the TCP port to communicate with host debugger.
- **busparams:** this setting defines the PCI bus, device, and function number of the debugging device.

Once again, the network interfaces' configuration is:

- **physical host (L0):** 192.168.0.96
    - **first virtual machine (L1 | VMware):** IP address from DHCP (NAT)
    - **first virtual machine (L1 | VMware):** IP address from DHCP (Host-Only)
        - **second virtual machine (L2 | Hyper-V):** IP address from DHCP

Readers should be careful here because network adapters from VMware virtual machine must be configured to get IP address through DHCP (do not use fixed IP) and, as suggested, connect adapters to NAT and Host-Only networks, respectively. Please, do not configure any of these network adapters to use Bridge.

To keep separate debugging sessions, the first network adapter will be used for kernel debugging and the second one for hypervisor debugging, and we will use the following ports:

- **kernel port:** 50020
- **hypervisor port:** 50021

The next step is to discover the **PCI parameters** of both virtual machine's network interfaces. Although it could be done graphically, it is easier to do it on PowerShell:

- **Get-NetAdapterHardwareInfo | Format-Table**

```
PS C:\> Get-NetAdapterHardwareInfo | Format-Table

Name                      Segment Bus Device Function Slot NumaNode PcieLinkSpeed PcieLinkWidth Version
----                      ------- --- ------ -------- ---- -------- ------------- ------------- -------
Ethernet00                      0   3      0        0  160                5.0 GT/s            32 1.1+
Ethernet1                       0   4      0        0  161                5.0 GT/s            32 1.1+
```

[Figure 06]: Network Interfaces | hardware properties

Once again, do not forget to create the following environment variable **_NT_SYMBOL_PATH=srv*c:\symbols*https://msdl.microsoft.com/download/symbols** on physical and virtual environments.

Proceeding with our configuration, we can set up the configuration for both kernel and hypervisor debugging by executing the following steps:

- **bcdedit /debug on**
- **bcdedit /dbgsettings net hostip:192.168.0.96 port:50020 key:1.2.3.4**
- **bcdedit /set {dbgsettings} busparams 3.0.0**
- **bcdedit /hypervisorsettings net hostip:192.168.0.96 port:50021 busparams:4.0.0 key:6.7.8.9**
- **bcdedit /set hypervisordebug on**
- **bcdedit /set hypervisorlaunchtype auto**
- **bcdedit /dbgsettings**
- **bcdedit /hypervisorsettings**

If you make any mistake and set a parameter with a wrong value, you can reset It by removing the value using ***bcdedit /deletevalue <param>*** or ***bcdedit /deletevalue {hypervisorsettings} <param>*.**

As a side note, kdnet might have been one option to set up this same configuration because, as explained previously, it can be used to enable distinct kinds of debugging, and hypervisor debugging is one of them. In this case, to configure. In this case, the easiest way to do it using kdnet.exe is executing the following command:

- **kdnet /busparams 3.0.0 192.168.0.96 50020 -k**
- **kdnet /busparams 4.0.0 192.168.0.96 50021 -h**

Or alternatively both in a single command, which automatically will assign the kernel debug port to 50020 and the hypervisor debug port to 50021, but it can bring issues when debugging both at the same time:

- **kdnet /busparams 3.0.0 192.168.0.96 50020 -kh**

To establish a connection, execute either the windbg command ("C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe") as shown below or through the graphical mode as explained previously (make sure you are using the new WinDbg). Anyway, execute the following commands:

- **windbg.exe -d -k net:port=50020,key=1.2.3.4**
- **windbg.exe -d -k net:port=50021,key=6.7.8.9**

The -k option is used to specify kernel or hypervisor debugging, and the **-d** option is used to force the debugger to break into the target computer after the reboot. Now, you should reboot the VMware virtual machine running **shutdown /r /t 0**.

Therefore, starting up the target virtual machine (L1 | VMware), you will see the following:

```
Opened WinSock 2.0
Waiting to reconnect...
Connected to target 192.168.0.96 on port 50021 on local IP 192.168.0.96.
You can get the target MAC address by running .kdtargetmac command.
Connected to Microsoft Hypervisor 26100 x64 target at (Wed Nov 20 15:04:19.117
2024 (UTC - 3:00)), ptr64 TRUE
Kernel Debugger connection established.

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*C:
\symbols*https://msdl.microsoft.com/download/symbols
Symbol search path is: srv*C:\symbols*https://msdl.microsoft.com/download/symbols
Executable search path is:
PEB is paged out (Peb.Ldr = 00000000`00262018).  Type ".hh dbgerr001" for details
ReadVirtual() failed in GetXStateConfiguration() first read attempt (error == 0.)
Microsoft Hypervisor Kernel Version 26100 MP (1 procs) Free x64
Edition build lab: 26100.0.GitEnlistment(IMProdBldB).241031-1202
Primary image base = 0xfffff800`78c00000 Loaded module list = 0xfffff800`78cdd1a0
System Uptime: not available
Unable to add extension DLL: hvexts
PEB is paged out (Peb.Ldr = 00000000`00262018).  Type ".hh dbgerr001" for details
hv+0x3a3ebf:
fffff800`78fa3ebf c3              ret
kd> k
 # Child-SP          RetAddr               Call Site
00 fffff878`5c5c7ac8 fffff800`78e4fa3a     hv+0x3a3ebf
01 fffff878`5c5c7ad0 00000000`0000000a     hv+0x24fa3a
02 fffff878`5c5c7ad8 00000000`00000000     0xa
kd> lm
start             end                 module name
fffff800`78c00000 fffff800`79015000   hv         (no symbols)

kd> lmDvmhv
Browse full module list
start             end                 module name
fffff800`78c00000 fffff800`79015000   hv         (no symbols)
    Loaded symbol image file: hvix64.exe
    Image path: hvix64.exe
    Image name: hvix64.exe
    Browse all global symbols  functions  data
    Image was built with /Brepro flag.
    Timestamp:        5B4DA72A (This is a reproducible build file hash, not a
timestamp)
    CheckSum:         001F84A1
    ImageSize:        00415000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
    Information from resource tables:
```

[Figure 07]: Hypervisor Debugging -- part 1

There are a few observations that we can make:

- We see the message *"Microsoft Hypervisor Kernel Version 26100 MP (1 procs) Free x64"* indicating that the hypervisor is there.
- The stack shows indication that the **hv module** is being called.
- Unfortunately, **there is no symbol for hv module**.
- The image path is **hvix64.exe**.
- It has been unable to add an extension DLL named **hvexts**.

Additionally, I run a few instructions to show to readers such instructions and, eventually, to help them with a guideline to check whether they are repeating steps correctly or not, as shown below:

```
kd> p
hv+0x24fa3a:
fffff800`78e4fa3a 4883c458        add     rsp,58h
kd> p
hv+0x24fa3e:
fffff800`78e4fa3e c3              ret
kd> p
hv+0x24b17d:
fffff800`78e4b17d 488b152c23e9ff  mov     rdx,qword ptr [hv+0xdd4b0 (fffff800`
78cdd4b0)]
kd> p
hv+0x24b184:
fffff800`78e4b184 488d0d1dfddbff  lea     rcx,[hv+0xaea8 (fffff800`78c0aea8)]
kd> p
hv+0x24b18b:
fffff800`78e4b18b e850480000      call    hv+0x24f9e0 (fffff800`78e4f9e0)
kd> p
PEB is paged out (Peb.Ldr = 00000000`00262018).  Type ".hh dbgerr001" for details
hv+0x24b190:
fffff800`78e4b190 40382d2ad2e5ff  cmp     byte ptr [hv+0xa83c1 (fffff800`
78ca83c1)],bpl
kd> p
hv+0x24b197:
fffff800`78e4b197 7413            je      hv+0x24b1ac (fffff800`78e4b1ac)
kd> p
hv+0x24b199:
fffff800`78e4b199 e88a530000      call    hv+0x250528 (fffff800`78e50528)
kd> p
hv+0x24b19e:
fffff800`78e4b19e 84c0            test    al,al
kd> p
hv+0x24b1a0:
fffff800`78e4b1a0 740a            je      hv+0x24b1ac (fffff800`78e4b1ac)
kd> p
hv+0x24b1ac:
fffff800`78e4b1ac ba38000000      mov     edx,38h
kd> p
hv+0x24b1b1:
fffff800`78e4b1b1 488bcb          mov     rcx,rbx
kd> p
hv+0x24b1b4:
fffff800`78e4b1b4 e847c3ffff      call    hv+0x247500 (fffff800`78e47500)
kd> g
```

[Figure 08]: Hypervisor Debugging -- part 2

In the instance of the WinDbg **debugging the kernel**, we have the following sequence:

```
Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target 192.168.0.96 on port 50020 on local IP 192.168.0.96.
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 26100 x64 target at (Wed Nov 20 15:02:54.920 2024 (UTC - 3:00)), ptr64
TRUE
Kernel Debugger connection established.

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*C:
\symbols*https://msdl.microsoft.com/download/symbols
Symbol search path is: srv*C:\symbols*https://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 26100 MP (1 procs) Free x64
Edition build lab: 26100.1.amd64fre.ge_release.240331-1435
Kernel base = 0xfffff805`c3000000 PsLoadedModuleList = 0xfffff805`c3ef4730
System Uptime: 0 days 0:00:00.640
nt!DebugService2+0x5:
fffff805`c34b94c5 cc              int     3
kd> g
KDTARGET: Refreshing KD connection
DriverEntry failed 0xc00000bb for driver \REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\uiomap
DriverEntry failed 0xc035001e for driver \REGISTRY\MACHINE\SYSTEM\ControlSet001\Services
\l1vhlwf
DriverEntry failed 0xc00000bb for driver \REGISTRY\MACHINE\SYSTEM\ControlSet001\Services
\NetworkPrivacyPolicy
WER/ReportFault/1208:743: ERROR CreateProcess/mode0 failed with Win32 error 267.
WER/ReportFault/1208:1011: ERROR StartCrashVertical failed: HRESULT 8007010B.
Shutdown occurred at (Wed Nov 20 15:04:12.615 2024 (UTC - 3:00))...unloading all symbol tables.
Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target 192.168.0.96 on port 50020 on local IP 192.168.0.96.
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 26100 x64 target at (Wed Nov 20 15:05:29.801 2024 (UTC - 3:00)), ptr64
TRUE
Kernel Debugger connection established.


************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*C:
\symbols*https://msdl.microsoft.com/download/symbols
Symbol search path is: srv*C:\symbols*https://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 26100 MP (1 procs) Free x64
Edition build lab: 26100.1.amd64fre.ge_release.240331-1435
Kernel base = 0xfffff805`84e00000 PsLoadedModuleList = 0xfffff805`85cf4730
System Uptime: 0 days 0:00:00.641
nt!DebugService2+0x5:
fffff805`852b94c5 cc              int     3
kd> k
 # Child-SP          RetAddr               Call Site
00 fffff805`16a29de8 fffff805`8527540d     nt!DebugService2+0x5
01 fffff805`16a29df0 fffff805`8593a4b0     nt!DbgLoadImageSymbols+0x8d
02 fffff805`16a29e40 fffff805`8590f466     nt!KdInitSystem+0x680
03 fffff805`16a29fc0 00000000`00000000     nt!KiSystemStartup+0x1d6
kd> lm
start             end                 module name
fffff805`84e00000 fffff805`8624f000   nt         (pdb symbols)          c:\symbols\ntkrnlmp.pdb
\B71612A847D4335437C475C195D345901\ntkrnlmp.pdb

Unable to enumerate kernel-mode unloaded modules, HRESULT 0x80004005
```

```
kd> lmDvmnt
Browse full module list
start             end                 module name
fffff805`84e00000 fffff805`8624f000   nt         (pdb symbols)          c:\symbols\ntkrnlmp.pdb
\B71612A847D4335437C475C195D345901\ntkrnlmp.pdb
    Loaded symbol image file: ntkrnlmp.exe
    Image path: ntkrnlmp.exe
    Image name: ntkrnlmp.exe
    Browse all global symbols  functions  data
    Image was built with /Brepro flag.
    Timestamp:        090D1C3D (This is a reproducible build file hash, not a timestamp)
    CheckSum:         00C2DAB0
    ImageSize:        0144F000
    File version:     10.0.26100.2222
    Product version:  10.0.26100.2222
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        1.0 App
    File date:        00000000.00000000
    Translations:     0409.04b0
    Information from resource tables:
        CompanyName:      Microsoft Corporation
        ProductName:      Microsoft® Windows® Operating System
        InternalName:     ntkrnlmp.exe
        OriginalFilename: ntkrnlmp.exe
        ProductVersion:   10.0.26100.2222
        FileVersion:      10.0.26100.2222 (WinBuild.160101.0800)
        FileDescription:  NT Kernel & System
        LegalCopyright:   © Microsoft Corporation. All rights reserved.

Unable to enumerate kernel-mode unloaded modules, HRESULT 0x80004005
kd> g
KDTARGET: Refreshing KD connection
DriverEntry failed 0xc00000bb for driver \REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\uiomap
DriverEntry failed 0xc035001e for driver \REGISTRY\MACHINE\SYSTEM\ControlSet001\Services
\l1vhlwf
DriverEntry failed 0xc00000bb for driver \REGISTRY\MACHINE\SYSTEM\ControlSet001\Services
\NetworkPrivacyPolicy
```

[Figure 09]: Kernel Debugging

Clearly readers can confirm the version of the kernel and that there are available symbols, as usual. After this last "**g**" command, the target Windows should boot normally, and you should see the logon screen.

I've used the "old" version of WinDbg in the figures above, and I didn't have any issue. The same should be true while using the new WinDbg (as known as WinDbg Preview).

Right now, the most important thing is having a working environment which makes us able to debug and research on hypervisor and further details will be explained later. Of course, as I had mentioned previously, readers can repeat this procedure using a previous version of VMware or even real machines.

Eventually, we could use IDA Pro to repeat the same approach shown so far because IDA offers a good integration with WinDbg.

The next method is an optional approach to debug VMware virtual machines and does not depend on the operating systems. As readers are going to learn, there are good and bad points, and it is up to you to decide which path you want to follow.

4.3 Hypervisor debugging via VMware

Another extremely attractive and useful alternative is debugging the virtual machine since its boot is using the built-in GDB debugger from VMware Workstation by editing the respective **.vmx file** (virtual machine configuration file) and addition the following parameters:

- **debugStub.listen.guest64** = "TRUE"
- **debugStub.listen.guest64.remote** = "TRUE"
- **monitor.debugOnStartGuest64** = "TRUE"
- **debugStub.hideBreakpoints** = "TRUE"
- **debugStub.port.guest64** = "55000"

All mentioned parameters' names are pretty intuitive, but a quick explanation follows:

- **debugStub.listen.guest64:** enable local virtual machine debuggingA\
- **debugStub.listen.guest64.remote:** enable remote virtual machine debugging
- **monitor.debugOnStartGuest64:** enable debugging since the virtual machine start-up
- **debugStub.hideBreakpoints:** try to hide hardware breakpoints from the virtual machine
- **debugStub.port.guest64:** establish a listening port for the GDB stub (the default is 8864).

All these parameters assume a 64-bit virtual machine.

Now you should start the virtual machine, but I recommend you follow the virtual machine log (**vmware.log**), which is located in the same directory as all other virtual machines' files. Two programs (there are many others) that you could use are:

- **Tail for Windows:** https://sourceforge.net/projects/wintail/
- **BareTail:** https://sourceforge.net/p/tailw/wiki/Home/

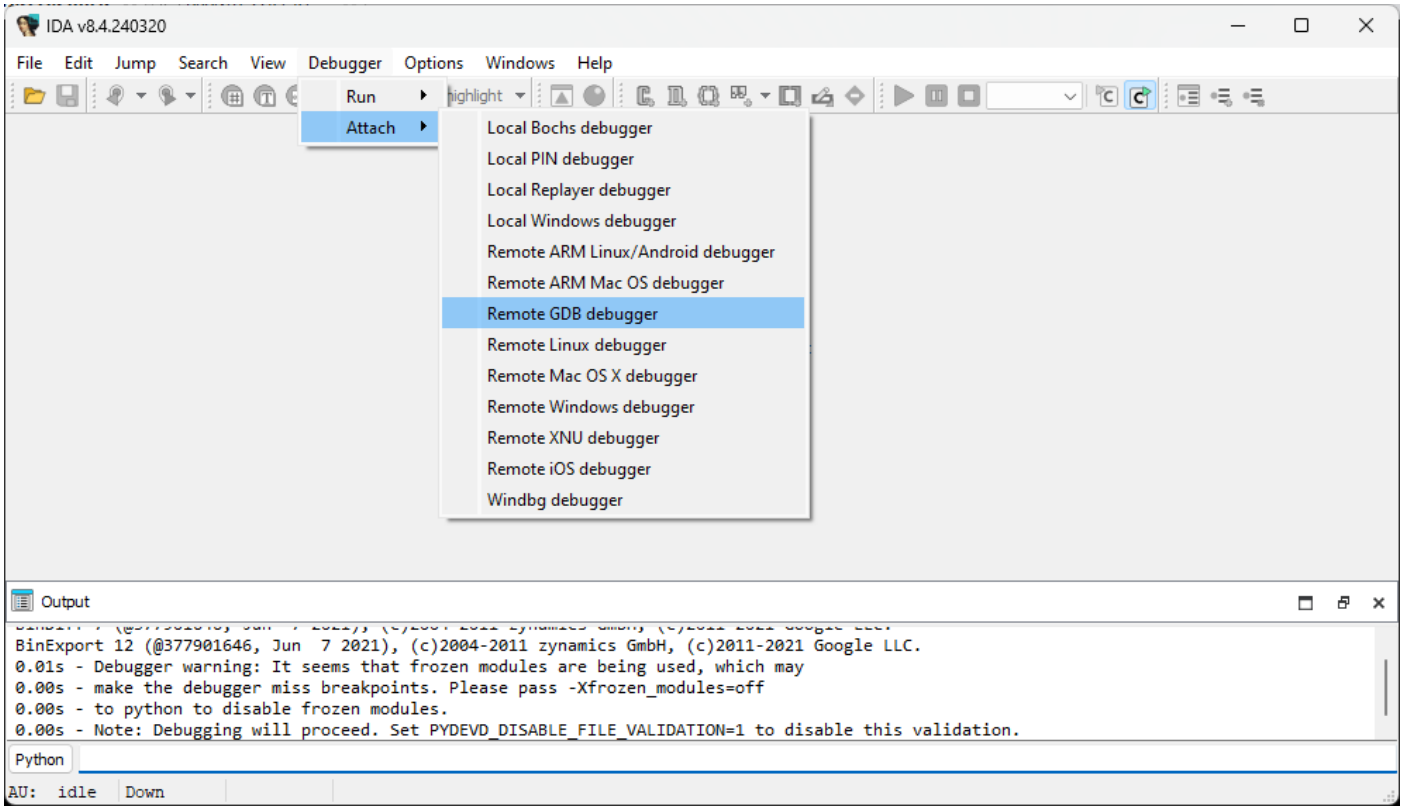
Readers will see something like shown below:

```
2024-08-23T03:34:43.359Z In(05) vcpu-0 Vix: [mainDispatch.c:4212]: VMAutomation_ReportPowerOpFinished: statevar=0, newAppState=1872,
2024-08-23T03:34:43.359Z In(05) vcpu-0 Vix: [mainDispatch.c:4129]: VMAutomationReportPowerStateChange: Reporting power state change
2024-08-23T03:34:43.359Z In(05) vcpu-0 Vix: [mainDispatch.c:4129]: VMAutomationReportPowerStateChange: Reporting power state change
2024-08-23T03:34:43.359Z In(05) vcpu-0 Transitioned vmx/execState/val to poweredOn
2024-08-23T03:34:43.359Z In(05) vcpu-0 Tools: Adding Tools inactivity timer.
2024-08-23T03:34:43.359Z In(05) vcpu-0 Intel VT: FlexPriority disabled, VPID enabled.
2024-08-23T03:34:43.362Z In(05) vmx USB: New set of 9 USB devices.
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:DisplayLink\ Dell\ Universal\ Dock\ D6000 vid:17e9 pid:6006 path:1/3/2/0
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Chicony\ HD\ Webcam vid:04f2 pid:b68b path:1/1/12 speed:high family:vide
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Intel(R)\ Wireless\ Bluetooth(R) vid:8087 pid:0026 path:1/1/13 speed:ful
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Realtek\ USB3.0-CRW vid:0bda pid:0316 path:1/1/17 speed:super family:sto
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Seagate\ RSS\ Expansion\ Desk vid:0bc2 pid:331a path:1/1/20 speed:super
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Seagate\ RSS\ Expansion\ HDD vid:0bc2 pid:2038 path:1/1/21 speed:super f
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Bizlink\ D6000\ Controller vid:06c4 pid:c413 path:1/1/0/3 speed:full fam
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Logitech\ Logi\ USB\ Camera\ (C270\ HD\ WebCam) vid:046d pid:0825 path:1
2024-08-23T03:34:43.362Z In(05) vmx USB: Found device [name:Virtual\ Bluetooth\ Adapter vid:0e0f pid:0008 speed:full family:wireless
2024-08-23T03:34:43.363Z No(00) vmx ConfigDB: Setting sata0:1.fileName = <not printed>
2024-08-23T03:34:43.366Z In(05) vmx VNET: MACVNetLinkStateEventHandler: event, up:1, adapter:0
2024-08-23T03:34:43.366Z In(05) vmx VNET: MACVNetLinkStateEventHandler: 'ethernet0' state from 4 to 6.
2024-08-23T03:34:43.366Z In(05) vmx VNET: MACVNetLinkStateEventHandler: event, up:1, adapter:0
2024-08-23T03:34:43.366Z In(05) vmx VNET: MACVNetLinkStateEventHandler: 'ethernet1' state from 4 to 6.
2024-08-23T03:34:43.383Z In(05) vcpu-0 Vix: [vmxCommands.c:4556]: VMAutomation_Pause: pause = TRUE
2024-08-23T03:34:43.384Z In(05) vcpu-0 Vix: [mainDispatch.c:6914]: VMAutomation: Ignoring VMAutomation_SendMsgToOneClient because co
2024-08-23T03:34:43.384Z Wa(03) vcpu-0 Waiting for Guest 64-bit debugger to connect!
```

[Figure 10]: vmware.log

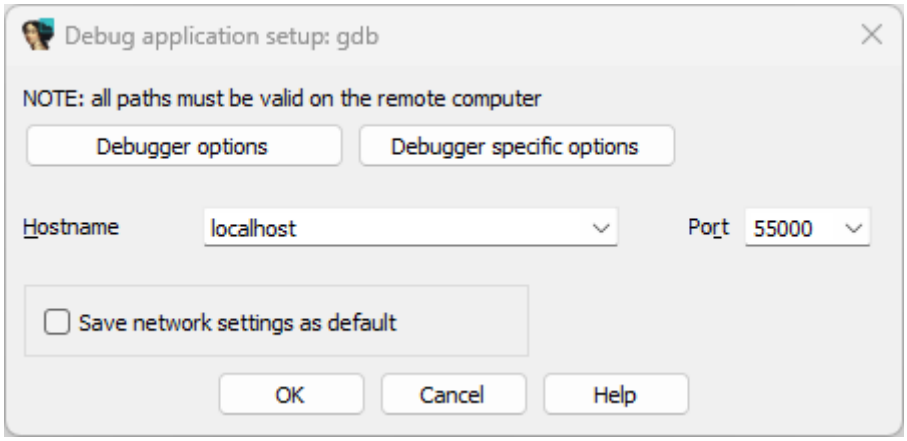
As highlighted, the guest is waiting for the connection from a debugger, and it is what we want.

Open IDA 64-bit and go to menu **Debugger | Attach | Remote GDB Debugger**, as shown below:



[Figure 11]: IDA Pro: Remote GDB Debugger

Soon you choose Remote GDB debugger, the following window will pop up:

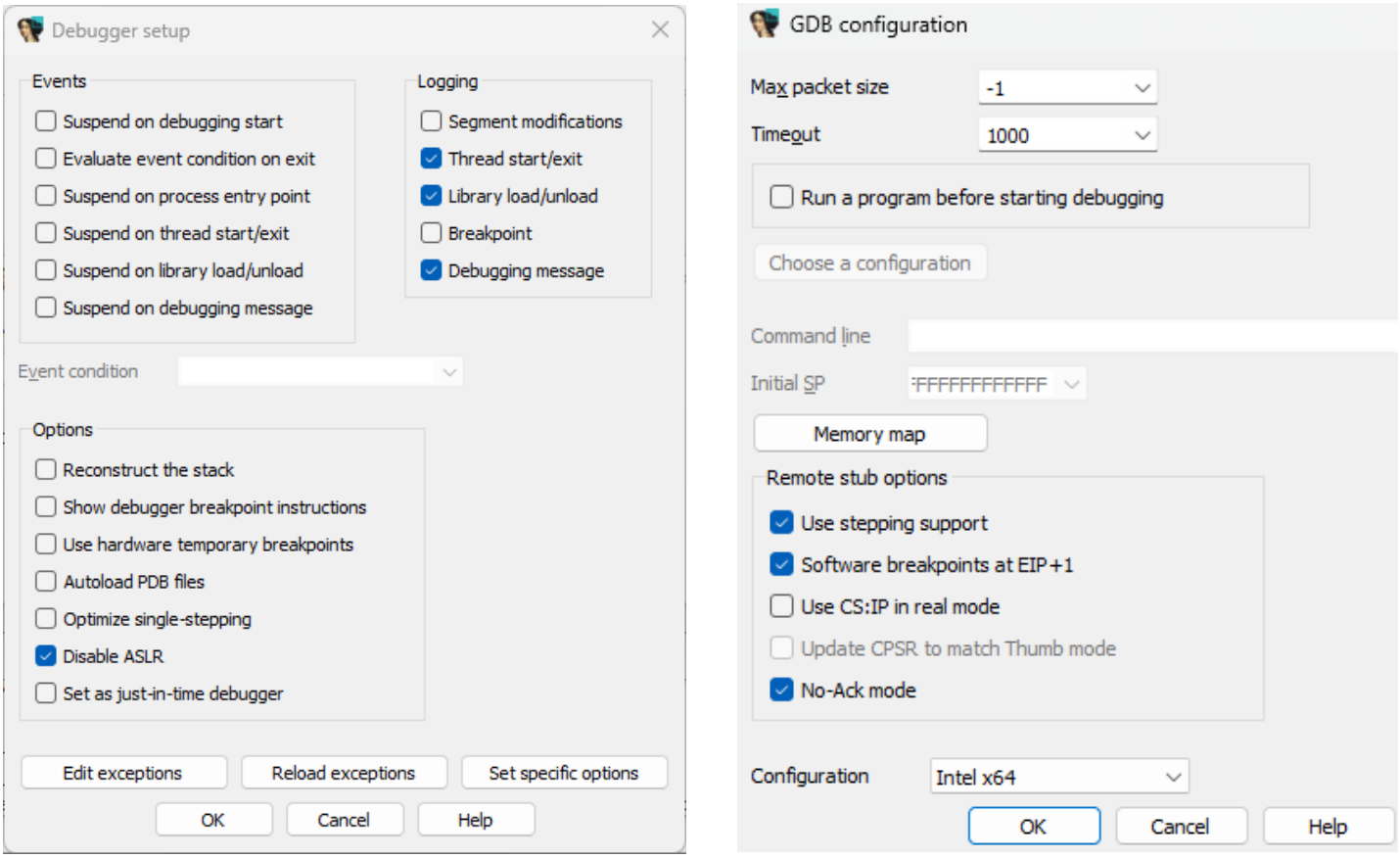


[Figure 12]: IDA Pro: Remote GDB Debugger

There are not too many options to be configured in this window:

- **hostname:** localhost
- **port:** 55000 (as we configured in the .vmx file

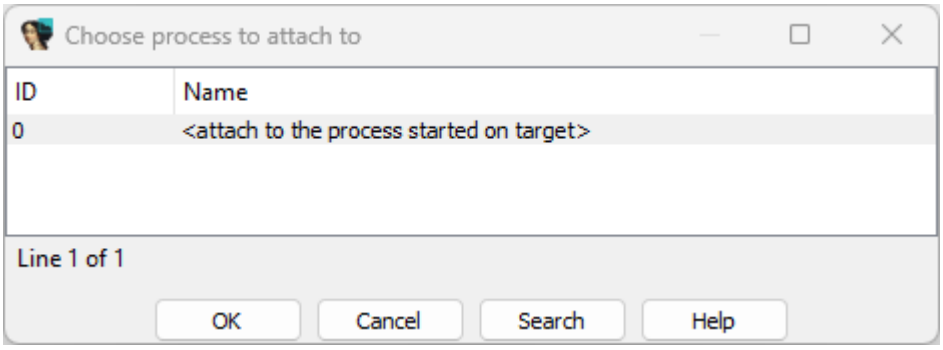
However, that is not enough. If you click on **Debugger options**, a list of classic and well-known options will be presented, which are the same as other situations when we use IDA Pro to debug any other application, but when we go Set specific options, we find an important setting for our purpose as **GDB Configuration**:



[Figure 13]: IDA Pro: Debugger configuration

Readers must ensure that the correct processor is set in the respective drop box.

Once you have finished these few steps, and returned to the Debug application setup window, click on OK and you will see the following window:



[Figure 14]: The IDA Pro offers to attach to the target virtual machine

If we proceed, certainly we will stop at the same point of the virtual machine initialization, but it will not make sense because we are not looking for this specific initialization, but the hypervisor initialization is what our target is. Therefore, it could be quite complicated to understand what-code-starts-where and, before using WinDbg or IDA Pro to retrieve further information, we need to learn foundations.

Therefore, it is time to move forward, learn such multiple and necessary concepts related to Hyper-V and later we return to practical debugging and choose an approach to initiate a dynamic analysis based on learned mechanisms.

5. General hypervisor concepts

As readers probably already know, different virtualization software relies its working on a set of processor instructions known as **Virtual Machine Extension (VMX)**, and there are commercial and open-source programs which represent hypervisors type 1 (bare metal, which run directly on the hardware) and type-2 (where the hypervisor works inside an operating system, which run on the hardware). Depending on the used hypervisor, there can be support or not for nested virtualization. In Intel processors, VMX operations are enabled by doing **CR4.VMXE[13] = 1 (14th bit)**. (*check it on page 3151, Control Registers, from Intel®64 and IA-32 Architecture Software Developer's Manual – Combined Volumes – October 2024*).

A brief list of key terms related to hypervisors follows:

- **Guest Software:** virtual machine running an operating system under the control of virtualization software such as Hyper-V, VMware, VirtualBox and so on. Guest software runs in VMX non-root operations.
- **Virtual Machine Monitor | VMM:** it is hypervisor itself, which plays the leading role as core control of virtualization operations by managing interactions between the guest software and physical hardware such as memory, processors, I/O and other ones. VMM runs in VMX root operation.
- **VM entry:** transition into VMX non-root operation.
- **VM exit:** transition from VMX non-root operation and VMX root operation.
- **Virtual Machine Control Structure (VMCS):** it is a structure that manages VMX non-root and VMS root transitions, and a VMM can use different VMCS structures for each supported virtual machine.
- **VMCS data is organized into six distinguished groups: guest-state area** stores the processor state on VM exits and such data is loaded during VM entries), **host-state area** (it holds the processor state that is loaded on VM exits), **VM-execution control fields** (hold information used for controlling the process behavior in VMX non-root operation), **VM-exit control fields** (holds data used for controlling VM exits), **VM-entry control fields** (holds data used for controlling VM entries) and **VM-exit information fields** (receive information on VM exits and contain the cause of the VM exits). Usually, VMCS is placed in a location named VMCS Region.

There are multiple instructions (virtual machine extensions) involved with virtualization operations , but the main ones are:

- **VMXOFF instruction:** it causes the software to leave VMX operation.
- **VMXON instruction:** it causes the software to enter into VMX operation, which is associated with VMXON region that is a region of memory that the logical processor uses to support such operation. The own VMXON is controlled by IA32_FEATURE_CONTROL MSR.
- **VMREAD:** it reads fields from VMCS.
- **VMWRITE:** it writes to the VMCS fields.
- **VMCLEAR:** it clears all VMCS fields (note that the VMCS region resides at the physical address).
- **VMLAUNCH:** it is executed soon after VMCLEAR and invokes a virtual machine (remember: VMM keeps a separate VMCS structure for each virtual machine).
- **VMRESUME:** this instruction resumes a virtual machine, which forces the processor to flip to VMX non-root mode.
- **VMCALL:** it allows a guest in VMX non-root operation to call the VMM for service (exiting from VM).

- **VMFUNC**: it allows a guess in VMX non-root operation to call a VM function.
- **VMPTRLD**: it makes the indicated VMCS active and current.
- **VMPTRST**: it stores the current VMCS pointer into a destination address/operand.

Another concept that deserves to be quickly explained is **SLAT (Second Level Address Translation)**, which is implemented by as **Shadow Page Tables** (no longer used) and **EPT** (Extended Page Tables). Although **EPT** (SLAT's implementation) seems like a complicated concept, the main idea is simple to be realized. Basically, guest operating systems cannot (and must not) access and control the physical memory address, which is obvious because we would have an absurd situation where the virtual machine controls the host's memory. Therefore, when a guest OS accesses physical memory, it is actually accessing an associated and intermediary page table (in other words, the guest believes that it is accessing a physical address, but it is accessing a virtual one), where memory operations are performed. There is a second intermediary page table maintained by the **Virtual Machine Monitor (VMM)**, which maps the guest's intermediary table to the real physical memory. In a few terms, it would be something like:

- **Guest OS → First intermediary PT → VMM → second intermediary PT → physical memory**

In other words: **GVA → GPA → SPA**, as it will be shown on the next page. These intermediary page tables are known as second layer EPT. Most modern systems support EPT:

```
C:\Users\Administrator>coreinfo64.exe -v

Coreinfo v3.6 - Dump information on system CPU and memory topology
Copyright (C) 2008-2022 Mark Russinovich
Sysinternals - www.sysinternals.com


Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz
Intel64 Family 6 Model 165 Stepping 2, GenuineIntel
Microcode signature: 000000E0
HYPERVISOR      -       Hypervisor is present
VMX             *       Supports Intel hardware-assisted virtualization
EPT             *       Supports Intel extended page tables (SLAT)
URG             *       Supports Intel unrestricted guest
```

[Figure 15]: Coreinfo64.exe (from sysinternals)

We can interact with the **Virtual Machine Monitor (VMM)** from an user-mode code through a driver by using **IOCTL Dispatcher mechanism** (in special, **DeviceIoControl** routine from client, and a callback routine for **IRP_MJ_DEVICE_CONTROL** IRP from the driver side) on Windows and even creating and operate on an own and minimal hypervisor, which we can send and receive data using well-known methods such as **METHOD_BUFFERED** (small amount of data), **METHOD_IN_DIRECT** and **METHOD_OUT_DIRECT** (large amount of data) or even **METHOD_NEITHER** (without any help from the I/O Manager). If you do not remember this subject, read the first two articles from this series.

6. Hyper-V foundations

Starting an article about Hyper-V (a typical type-1 hypervisor: https://en.wikipedia.org/wiki/Hypervisor) demands, before anything, to make clear main involved concepts and new nomenclatures as occur every

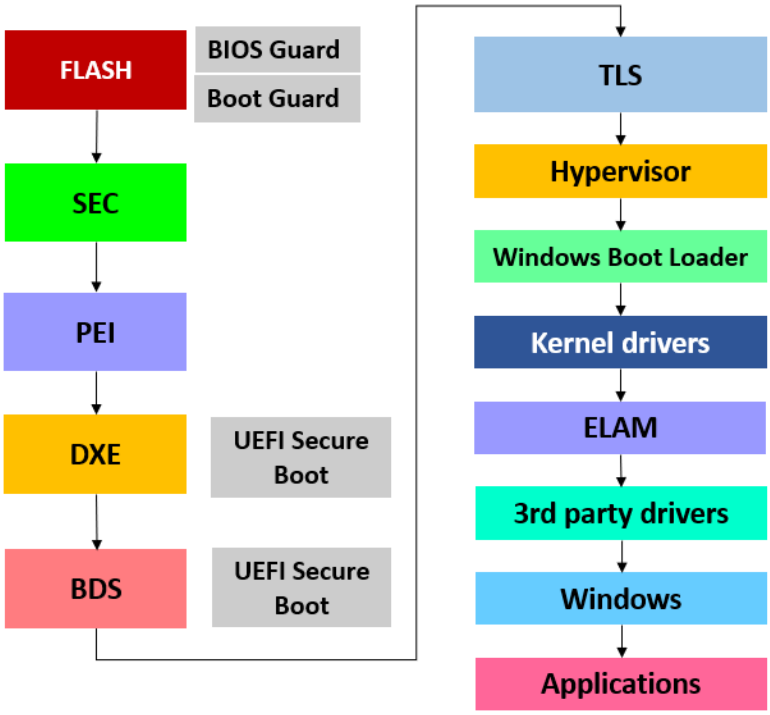
time we try to present or explain something, and even though terms might be obvious, is recommended to do it:

- **Hypercall:** basically, it is a system call to the hypervisor and as expected, it is only of methods of communication with the hypervisor from the guest operating system.
- **Partition:** it is an isolated container that is composed of a set of hardware resources and running operating system.
- **Root Partition:** in our configuration here is our host, a privileged partition where the virtualization stack runs with full access to hardware devices, and that contains child partitions.
- **Child Partition:** it is where the guest operating system runs.
- **System physical address (SPA):** it is the physical address seen by the CPU (host).
- **Guest physical address (GPA):** it is the physical memory seen by the guest operating system.
- **Guest virtual address (GVA):** it is virtual address (after translation) seen by a guest and, consequently, its applications.
- **VMBus:** it is a mechanism that works as a communication path between child partitions and the root partition. In other words, it makes an effective part of the virtualization stack.
- **Enlightened I/O:** it is a special virtualization feature, where different protocols are implemented and are aware of the virtualization as well as guests use specific drivers. The entire scheme runs on the top of VMBus (imagine a kind of stack, virtualization stack), and whose scheme allows the communication to skip the virtualization layer and directly access the I/O devices (mainly network and storage devices) to obtain a higher performance. This feature reminds **VMware DirectPath I/O**. To get such an advantage, Hyper-V Integration Services must be installed.

Readers also will notice that learning concepts of this area demands a different list of skills such as Windows internals, platform security, reverse engineering, processor's concepts and, if possible, programming. Finally, the big picture and goals are, in the long term:

- Understanding how all components work and interact with each other.
- Discovering multiple execution flows to get a comprehensive map about what routines are called and the order that they are called.
- Finding interfaces that make interaction possible.
- Studying involved rights and privileges with each component.
- Realizing that security principles are applied.
- Finding what rules we might try to break.
- Engineering a method of fuzzing available interfaces.

Of course, we cannot get everything at a time and not even using only one approach, so that is the reason why we compose the static and dynamic approach, as usual in every research, and keep in mind that hypervisors, like any other applications, are not alone and there are a series of interactions with other parts of the system such a platform components, operating system, kernel drivers and hardware. To provide a simple example, the following image is a macro and higher-overview about components involved in the Windows booting, passing through the hypervisor (where there are multiple tasks being executed), until reaching the operating system and applications working:



[Figure 16]: Simplified diagram containing main blocks since flash up to applications

Based on the definitions from the previous page, we can see the there are three different address space that are **SPA** (HV_SPA), **GPA** (HV_GPA) and **GVA** (HV_GVA), and guests can interact and request services from hypervisor by using hypercalls (specifying the GPA, as expected), which as classified by simple and repeat (known as rep), and actually latter is a sequence of simple hypercalls. Regardless of the hypercall type, they can be only invoked from **CPL (current privilege level)** equal to zero, which is the most privileged guest processor mode, and the specified **guess physical address (GPA)** must be 8-byte aligned, mapped, and marked as readable. Additionally, the partitions must have enough privilege to hypercall be succeeded and any returned value (failure or not) is saved into a status field whose type is **HV_STATUS**.

Hypercalls (regular or extended ones, whose code is above 0x8000) are invoked by using a specific opcode, and **HV_X64_MSR_HYPERCALL MSR** is used to initiate and manage such hypercalls, where actually the sequence is composed **HV_X64_MSR_GUEST_OS_ID** (guest OS identification) followed by **HV_X64_MSR_HYPERCALL MSR** with hypercall parameters.

There is a lengthy list of hypercalls, and they will be refreshed according to context that we are handling, so do not waste time right now trying to memory them. A brief list containing a few of them follows:

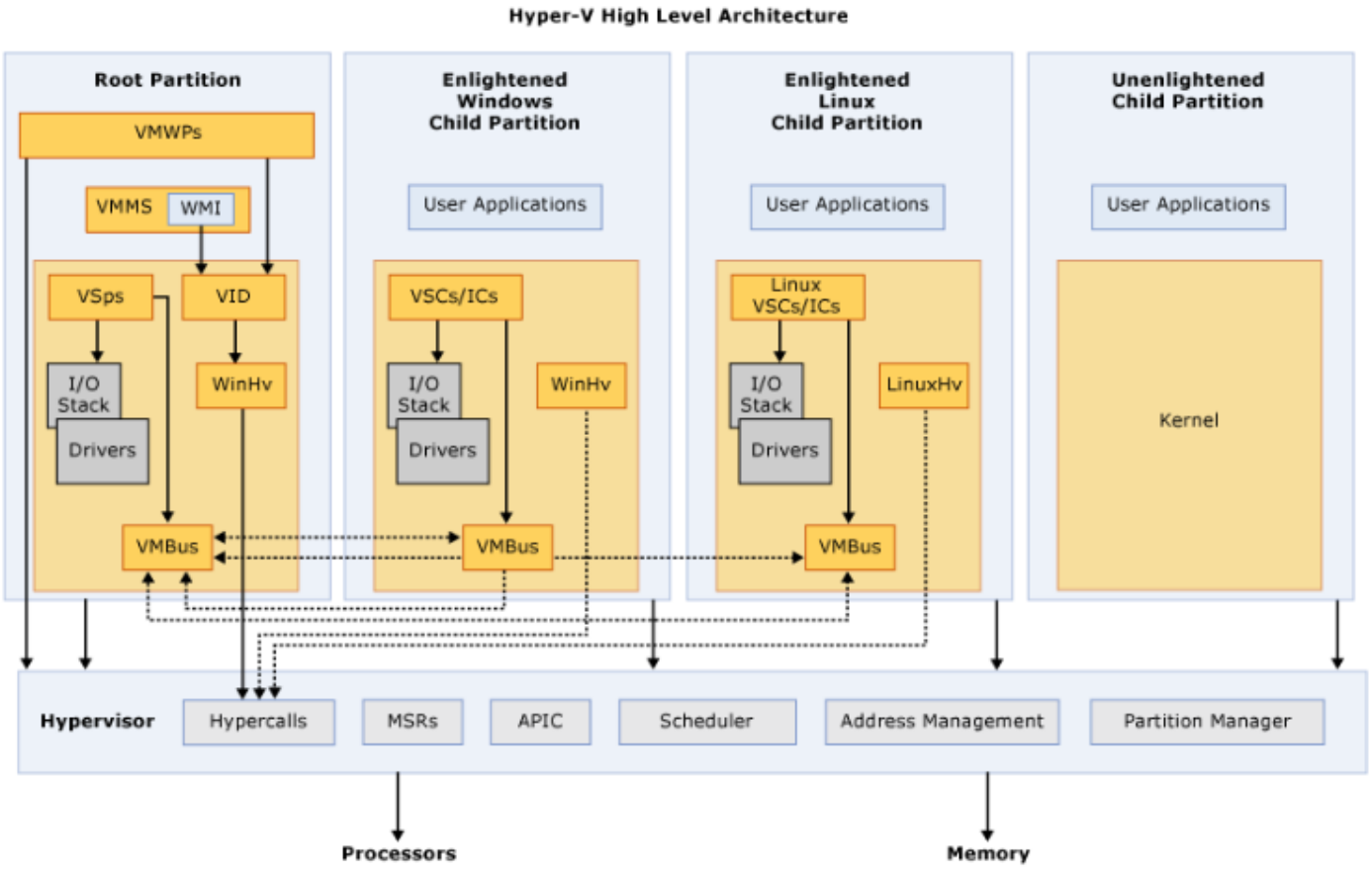
- **HvCallEnablePartitionVtl (0x000d)**
- **HvCallEnableVpVtl (0x000f)**
- **HvCallVtlCall (0x0011)**
- **HvCallVtlReturn (0x0012)**
- **HvCallPostMessage (0x005C)**
- **HvStartVirtualProcessor (0x0099)**
- **HvCalGetVpRegisters (0x0050)**
- **HvCallSetVpRegisters (0x0051)**

7. Hyper-V Architecture

A Hyper-V environment is roughly composed of a host (root or parent partition) and one or more guest operating system (one of each in a child partition), which does not have direct access to the processor and other hardware resources, and that is the exact scenario where the hypervisor acts as a judge and decides what is or is not authorized to flow from one side to another one.

The hypervisor, which is positioned in the way for communication among child partitions and the root partition, translates virtual addresses by using an **IOMMU** (Input Output Memory Management Unit), what means that is responsible for translating (remapping) physical memory addresses to virtual addresses used by child partitions. As readers have already noticed, any attempt from non-root partitions to access hardware is controlled by the hypervisor. In detail, the real communication happens from **Virtualization Service Consumers** (**VSCs**, from child partitions) and **Virtualization Service Providers** (**VSPs**, from the root partition), through the VMBus (defined previously in this text).

As mentioned, child partitions do not have direct access to hard, and actually **VDevs (virtual devices)**, which are similar to other operating systems like Orace Solaris, are presented to child partitions them as a kind of view of resources. As happens in any discussion on Hyper-V, we need a picture to make easier to visualize components and its respective locations, and the **Hyper-V High Level Architecture** offered by the official Microsoft documentation continue being the most recommended:



[Figure 17]: Hyper-V High Level Architecture | credits: Microsoft Corporation

As **VMbus**, represented by a GUID and a type, is the main communication channel between partitions and, as a consequence, it plays a central role in the Hyper-V, it is a common target when using fuzzing approach because attackers can send unexpected (or even random) data through VMBus aims to cause issues in the guest. Hyper-V is really complex, and there are multiple components that can be fuzzed.

At the same way we have seen in other virtualization technologies over years, Hyper-V also offers a method to access hardware resources directly, circumventing the whole device emulation layer presented by the hypervisor, named **Enlightened I/O** (installed via Integration Service components) and that makes the communication between VSCs (from child partition) and VSPs (from the root partition) faster and efficient. If you examine the previous figure, there are other terms that have not explained yet:

- **VMMS (Virtual Machine Management Service)**: it manages the state of all virtual machines in child partitions, where the status is influenced by shutdown, power off, start, resume, and pause commands and as expected, it also reads the configuration of the virtual machine. Additionally, VMMS also manages addition and removal of devices as well as creation of respective Virtual Machine Worker Process (check next bullets).

- **VMWP (Virtual Machine Worker):** it provides virtual machine management services from parent partition to guest OS running in the child partition.

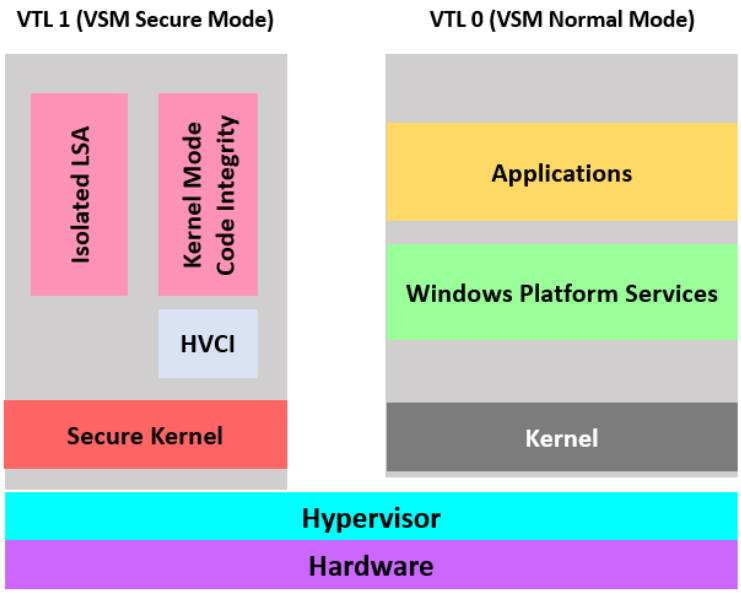
- **VID (Virtualization Infrastructure Driver):** the VID.sys is responsible and includes a manager for the virtualization stack memory (associated with the guest physical memory), and so that provides multiple services (processor management, memory, and partition services) for virtual machines running guest operating systems. Furthermore, the own VMWP and VMMS services use the same VID to communicate with the hypervisor, which makes them a useful source for looking for eventual vulnerabilities.

The next concept that we will review, and this one closer to the operating system, is **Virtual Machine Mode (VSM)**, which is a composition of features that allows the creation of security boundaries within Windows and allows well-known features such as Device Guard and Credential Guard to exist since Windows 10. Additionally, the VSM introduces the possibility of keeping in isolation between partitions through **Virtual Trust Level (VTL)**, where we have three current working VTLs:

- **VTL 0:** least privileged, where application and normal kernel run. Components running at this level cannot access any component frow upper levels.
- **VTL 1:** more privileged than VTL 0, where Secure Kernel and trustlets run. Obviously, such components are available for direct access from VTL 0, but there is controlled communication between both levels.
- **VTL 2:** it is where the OpenHCL (an open-source paravisor) runs. This new level allows separated VM workloads as well older Linux versions and Windows to run. Fundamentally, it is used in Azure and provides the possibility of executing a confidential VM (according to Microsoft, code and data are protected from the hypervisor and host OS).

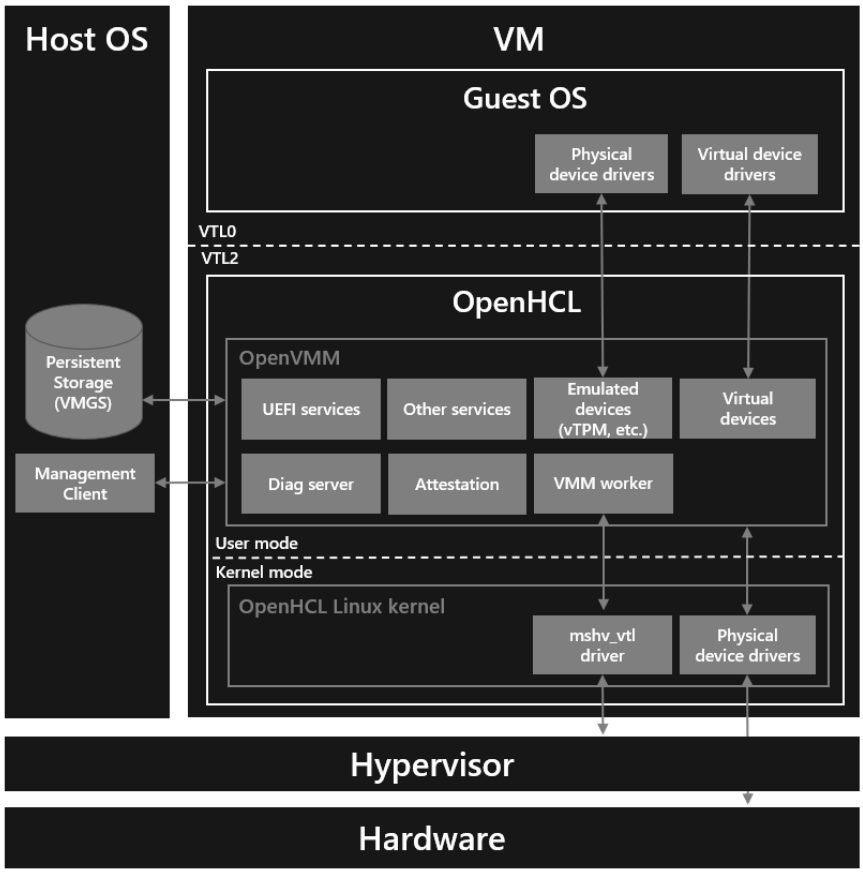
Certainly, the topic of **OpenHCL (HCL means Host Compatibility Layer)** is really cool, but I am not covering it in this article to avoid losing focus. I recommend you read the mentioned articles in the Reference section to learn more about them. From *Microsoft documentation* and *Windows Internals 7th edition book*

(part 1), we have an idea of how VTL 0 and VT1 are organized, where VTL 0 and VTL 1 are known as VSM Normal Mode and VSM Secure Mode, respectively:



[Figure 18]: Windows VTL 0 and VTL 1

Taking the same approach, and re-using a photo from *Microsoft documentation*, we also know as is the disposition of **Host OS**, **Hypervisor**, **VTL 2 (OpenHCL)** and **VTL 0 (Guest OS)** :



[Figure 19]: Windows VTL 0 and VTL 2 (credits: Microsoft)

VSM (Virtual Secure Mode) is built and works on the hypervisor and offers isolation between VTL 0 (running normal kernel) and VTL 1 (running secure kernel), secure devices (mapped into Secure Kernel in VTL 1), module verification through HVCI protection for each loaded module in VTL 0, security enclave memory and **CFG (Control Flow Guard)** at the kernel level (known as **KCFG**).

Before proceeding to other concepts, there are further details to be commented on. The **Virtualization Service Provider (VSP)** is hosted in the root partition and offers a point of contact for requests coming from **Virtualization Service Consumers (VSCs)** of child partitions (where guests are running) through the **VMBus**, which means intensive traffic to both directions, and that works through shared memory communications acting between root partition (host) and child partition (guest). In another words, the interface is composed of a series of VMBus pipes that make communication possible.

To start such a communication, the VSC calls **VmbChannelEnable( )** (from **KMCL driver,** which provides callbacks as reaction for messages received that are copied to memory) to open **VMBus channel**, which is offered by the root. Without offering a detailed level of details at this time, communication between child and root partition happens by writing and reading data to/from a ring buffer, where the sending is performed by invoking **VmbChannelSendSynchronousRequest( )** from **KMCL driver** too. Of course, those are not unique interfaces to communicate and use VMBus, and Hyper-V Sockets also offers communication using sockets through **HvSockets.sys** (low-level services) and **HvSocketControl.sys** (loads the **HvSocket** provider if necessary) **drivers**, and **HvSocket.dll** library. An important aspect of the available code is that everything aims to keep the least privilege approach, and the hypervisor and the host kernel provide the minimal interference as possible.

As common machines hold different devices, Hyper-V (actually, the virtualization stack) also offers such devices to child partitions that run operating systems as **emulated devices** (slow), **paravirtualized devices** (as well as synthetic devices, which have reliable performance) and **direct-access devices** (known as hardware-accelerated devices). Regardless of the device type, operations happen through **port I/O** using either an **exclusive I/O address space** or **mapped I/O** (memory accessible from processors). In a very simplified way, we have:

- **virtual processor → I/O Port | device MMIO space → VMEXIT → VID driver → VM Worker Thread → VDEV (virtual device) callback.**

To clear a term mentioned above, **VDEV (virtual device)** is an emulated or paravirtualized device that is hosted in user-mode and as expected, controlled by the hypervisor, and examples of VDEV are storage, networking, video, PCI bus and other ones, while we can have the same storage, networking and video offered as paravirtualized component. Actually, another component exists that is named **Integration Component (IC)**, which guests communicate with and, in last instance, are also faced as VDEV while being targets during verification and attacks, and examples are Heartbeat, Time Sync and other ones.

Finally, the next and really important topic to be discussed in this quick review is about the Secure Kernel, which runs in **VTL 1**, and it is mostly implemented by **securekernel.exe** that is extended by **vmsvcext.dll**, **cng.sys** and **skci.dll** modules that have been loaded by the Windows Loader. One of the key services of the Secure Kernel is to provide **Isolated User Mode (IUM)**, which runs in **VTL 1 (CPL==3)**, with services, and such services are attended by making use of hypercalls, which is the main interface to fulfill requests that demands transition from VTL 0 to VTL 1. One of the most important of these hypercalls is the **HvCallVtlCall**, which has an assigned secure service system call number equal to 0x11, and it is exactly responsible for the transition between VTL 0 and VTL 1. The communication itself is based on **Mailslot** and **ALPC**. As readers

can remember, while **Windows Mailslots** are IPC mechanisms that provide one-way communication between processes, **ALPC (Advanced Local Procedure Call)** is asynchronous high-speed Inter-Process Communication mechanism also dedicated to exchange messages between system components.

Once inside **Secure Kernel**, it implements another class of system calls named secure system calls, which are exclusive for trustlets (applications) running in VTL 1, and not directly available for normal kernel from VTL 1. If the normal kernel needs any service offered by the **Secure Kernel**, it invokes **VslpEnterIumSecureMode** by providing the target secure call number. This function starts a sequence of calls given by **HvlSwitchToVsmVtl1** and **HvlpSwitchToVsmVtl1RetpolineHelper**, ending with the **HvVtlCall** hypercall. Being very honest, even though the sequence of calls is exciting to follow, it is not really quite relevant right now and serves only to provide some context on secure kernel operation. Actually, learning too many details does not help to get comprehension of the general architecture at this moment.

So far readers have already noticed that we have different types of calls such as **hypercall**, **secure call** (happens from VTL0 to VTL1, which is represented by a sequence **nt!VslpEnterIumSecureCode + vmcall (0x11)** and **normal/regular calls** (happens from VTL1 to VTL0, which is represented by a sequence **sk!SkCallNormalMode + vmcall (0x12)**). Furthermore, calling hypercalls is not a free task, and a series of restrictions are applied to allow such calls being invoked. While we are only focusing on hypercalls, the picture is much bigger, and a simple application being called from guest OS passes through a series of components, and it is something like **application → I/O Stack → storVSC → VMBus → (host) StorVSP → VSMB (optionally) → I/O Stack → Storage**.

Based on this overview we have seen so far, there are good targets that deserve further investigation and research to find potential vulnerabilities. Certainly, any interface is a good starting point, and as an example we have hypercalls and intercepting handlers such EPT page faults, in/out instructions, and other ones, but there is no doubt that the major Hyper-V attack surface is in the root partition because everything starts from there. Therefore, it is a suitable time to get a list of vulnerabilities related to Hyper-V that we have seen over years:

- Denial of Service
- Vulnerabilities in VMBus and any associated drivers.
- Uninitialized stack objects.
- Uninitialized function pointers.
- EoP (elevation of privilege).
- Read/Write primitives in any exposed data structure.
- Hypercalls vulnerabilities.
- Out-of-Bounds RW in critical mechanisms, components, and structures.
- Any kind of I/O Interception.
- Information leak from MMIO.
- Information leaks from shared virtual hard disk and other structures.
- Classical UAF, dangling pointers, double free and race conditions.

Another interesting aspect that we can forget is that, as we learned, Hyper-V have diverse levels that can be attacked such as root partition, guest partition and hypervisor. Probably, one of the most well-known targets is hypercalls because they can be fuzzed from guest or root partition, and an eventual side effect will be propagated to the hypervisor. Of course, the fuzzing can be completely random and needs to respect the requested input format. Based on existing rules, mutations are done and then the fuzzing task is performed.

A suggestion is to retrieve a list of known Hyper-V vulnerabilities of the last years, to understand . Using the excellent **CVEProject/cvelistV5 project**, which is updated many times during the day (you need to update it using git pull command), and a funny sequence of commands that is almost entirely fixed and saves reader of programming any script in Python, you can get the following result:

- **git clone https://github.com/CVEProject/cvelistV5**
- **cd /root/github/cvelistV5/cves**
- **grep -r '"title": ".*Windows Hyper-V.*"' * | sort -r | sed -e 's/ \+/\t/g' -e 's/\",//g' | awk -F'"title": "' '{print $1 $2}' | grep -P '(?<=/)[^/.]+(?=\.)'**

```
root@ubuntu01:~/github/cvelistV5/cves# grep -r '"title": ".*Windows Hyper-V.*"' * | sort -r | sed -e 's/    \+/\
t/g' -e 's/\",//g' | awk -F'"title": "' '{print $1 $2}' | grep -P '(?<=/)[^/.]+(?=\.)'
2024/43xxx/CVE-2024-43633.json: Windows Hyper-V Denial of Service Vulnerability
2024/43xxx/CVE-2024-43624.json: Windows Hyper-V Shared Virtual Disk Elevation of Privilege Vulnerability
2024/43xxx/CVE-2024-43575.json: Windows Hyper-V Denial of Service Vulnerability
2024/43xxx/CVE-2024-43567.json: Windows Hyper-V Denial of Service Vulnerability
2024/43xxx/CVE-2024-43521.json: Windows Hyper-V Denial of Service Vulnerability
2024/38xxx/CVE-2024-38235.json: Windows Hyper-V Denial of Service Vulnerability
2024/38xxx/CVE-2024-38127.json: Windows Hyper-V Elevation of Privilege Vulnerability
2024/38xxx/CVE-2024-38080.json: Windows Hyper-V Elevation of Privilege Vulnerability
2024/30xxx/CVE-2024-30092.json: Windows Hyper-V Remote Code Execution Vulnerability
2024/30xxx/CVE-2024-30017.json: Windows Hyper-V Remote Code Execution Vulnerability
2024/30xxx/CVE-2024-30011.json: Windows Hyper-V Denial of Service Vulnerability
2024/30xxx/CVE-2024-30010.json: Windows Hyper-V Remote Code Execution Vulnerability
2024/29xxx/CVE-2024-29064.json: Windows Hyper-V Denial of Service Vulnerability
2024/21xxx/CVE-2024-21408.json: Windows Hyper-V Denial of Service Vulnerability
2024/21xxx/CVE-2024-21407.json: Windows Hyper-V Remote Code Execution Vulnerability
2024/20xxx/CVE-2024-20700.json: Windows Hyper-V Remote Code Execution Vulnerability
2024/20xxx/CVE-2024-20699.json: Windows Hyper-V Denial of Service Vulnerability
2024/20xxx/CVE-2024-20684.json: Windows Hyper-V Denial of Service Vulnerability
2024/20xxx/CVE-2024-20659.json: Windows Hyper-V Security Feature Bypass Vulnerability
2023/36xxx/CVE-2023-36908.json: Windows Hyper-V Information Disclosure Vulnerability
2023/36xxx/CVE-2023-36427.json: Windows Hyper-V Elevation of Privilege Vulnerability
2023/36xxx/CVE-2023-36408.json: Windows Hyper-V Elevation of Privilege Vulnerability
2023/36xxx/CVE-2023-36407.json: Windows Hyper-V Elevation of Privilege Vulnerability
2023/36xxx/CVE-2023-36406.json: Windows Hyper-V Information Disclosure Vulnerability
2023/32xxx/CVE-2023-32013.json: Windows Hyper-V Denial of Service Vulnerability
2023/23xxx/CVE-2023-23411.json: Windows Hyper-V Denial of Service Vulnerability
2022/44xxx/CVE-2022-44682.json: Windows Hyper-V Denial of Service Vulnerability
2022/41xxx/CVE-2022-41094.json: Windows Hyper-V Elevation of Privilege Vulnerability
2022/38xxx/CVE-2022-38015.json: Windows Hyper-V Denial of Service Vulnerability
2022/37xxx/CVE-2022-37979.json: Windows Hyper-V Elevation of Privilege Vulnerability
2022/35xxx/CVE-2022-35751.json: Windows Hyper-V Elevation of Privilege Vulnerability
2022/34xxx/CVE-2022-34696.json: Windows Hyper-V Remote Code Execution Vulnerability
2022/30xxx/CVE-2022-30223.json: Windows Hyper-V Information Disclosure Vulnerability
2022/30xxx/CVE-2022-30163.json: Windows Hyper-V Remote Code Execution Vulnerability
2022/29xxx/CVE-2022-29106.json: Windows Hyper-V Shared Virtual Disk Elevation of Privilege Vulnerability
2022/26xxx/CVE-2022-26785.json: Windows Hyper-V Shared Virtual Hard Disks Information Disclosure Vulnerability
2022/26xxx/CVE-2022-26783.json: Windows Hyper-V Shared Virtual Hard Disks Information Disclosure Vulnerability
2022/24xxx/CVE-2022-24539.json: Windows Hyper-V Shared Virtual Hard Disks Information Disclosure Vulnerability
2022/24xxx/CVE-2022-24537.json: Windows Hyper-V Remote Code Execution Vulnerability
2022/24xxx/CVE-2022-24490.json: Windows Hyper-V Shared Virtual Hard Disks Information Disclosure Vulnerability
2022/24xxx/CVE-2022-24466.json: Windows Hyper-V Security Feature Bypass Vulnerability
2022/23xxx/CVE-2022-23268.json: Windows Hyper-V Denial of Service Vulnerability
2022/23xxx/CVE-2022-23257.json: Windows Hyper-V Remote Code Execution Vulnerability
2022/22xxx/CVE-2022-22713.json: Windows Hyper-V Denial of Service Vulnerability
2022/22xxx/CVE-2022-22712.json: Windows Hyper-V Denial of Service Vulnerability
2022/22xxx/CVE-2022-22042.json: Windows Hyper-V Information Disclosure Vulnerability
2022/22xxx/CVE-2022-22009.json: Windows Hyper-V Remote Code Execution Vulnerability
```

[Figure 20]: Last Hyper-V vulnerabilities

One of advantages of this extremely simple approach is that:

- It saves time and always presents updated results.
- It allows you to investigate further details on the vulnerability using a simple Python script containing the target fields.

Writing a simple script you have:

```python
import json
import argparse
import sys

# Set up argument parser
parser = argparse.ArgumentParser(description='Parse specific fields from a JSON file.')
parser.add_argument('file', type=str, nargs='?', help='Path to the JSON file')

# Parse arguments
args = parser.parse_args()

# Check if the file argument is provided
if not args.file:
    print("Usage: python3 parse_json.py <path_to_json_file>")
    sys.exit(1)

# Read JSON data from file
with open(args.file, 'r') as file:
    json_data = file.read()

# Parse JSON
data = json.loads(json_data)

# Extract required fields with checks
url = data.get('containers', {}).get('cna', {}).get('references', [{}])[0].get('url', 'N/A')
date_published = data.get('cveMetadata', {}).get('datePublished', 'N/A')

# Extract all lessThan versions, descriptions, and products with checks
less_than_list = [version['lessThan'] for affected in data.get('containers', {}).get('cna', {}).get('affected', []) for version in affected.get('versions', []) if 'lessThan' in version]
cwe_list = [desc.get('cweId', 'N/A') for problem in data.get('containers', {}).get('cna', {}).get('problemTypes', []) for desc in problem.get('descriptions', [])]
description_list = [desc.get('description', 'N/A') for problem in data.get('containers', {}).get('cna', {}).get('problemTypes', []) for desc in problem.get('descriptions', [])]
product_list = [affected.get('product', 'N/A') for affected in data.get('containers', {}).get('cna', {}).get('affected', [])]
base_score = data.get('containers', {}).get('cna', {}).get('metrics', [{}])[0].get('cvssV3_1', {}).get('baseScore', 'N/A')
exploitation = data.get('containers', {}).get('adp', [{}])[0].get('metrics', [{}])[0].get('other', {}).get('content', {}).get('options', [{}])[0].get('Exploitation', 'N/A')
impact = data.get('containers', {}).get('adp', [{}])[0].get('metrics', [{}])[0].get('other', {}).get('content', {}).get('options', [{}])[2].get('Technical Impact', 'N/A')

# Print extracted fields with '[+]' only in attribute names
print(f"[+] url: {url}")
print(f"[+] datePublished: {date_published}")
print("[+] lessThan versions:")
for less_than in less_than_list:
    print(f"  - {less_than}")

print("[+] cwe:")
for cwe in cwe_list:
    print(f"  - {cwe}")

print("[+] descriptions:")
for description in description_list:
    print(f"  - {description}")

print("[+] products:")
for product in product_list:
    print(f"  - {product}")

print(f"[+] baseScore: {base_score}")
print(f"[+] Exploitation: {exploitation}")
print(f"[+] Impact: {impact}")
```

[Figure 21]: Script for parsing important fields from the JSON file

Running the script against one of returned CVE files we have the following result:

```
root@ubuntu01:~/github/cvelistV5/cves# python parse_cve.py 2024/43xxx/CVE-2024-43624.json
[+] url: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2024-43624
[+] datePublished: 2024-11-12T17:54:01.498Z
[+] lessThan versions:
  - 10.0.17763.6532
  - 10.0.17763.6532
  - 10.0.17763.6532
  - 10.0.20348.2849
  - 10.0.20348.2819
  - 10.0.19044.5131
  - 10.0.22621.4460
  - 10.0.19045.5131
  - 10.0.22631.4460
  - 10.0.22631.4460
  - 10.0.25398.1251
  - 10.0.26100.2314
  - 10.0.26100.2240
  - 10.0.26100.2314
  - 10.0.26100.2240
  - 10.0.26100.2314
  - 10.0.26100.2240
[+] cwe:
  - CWE-822
[+] descriptions:
  - CWE-822: Untrusted Pointer Dereference
[+] products:
  - Windows 10 Version 1809
  - Windows Server 2019
  - Windows Server 2019 (Server Core installation)
  - Windows Server 2022
  - Windows 10 Version 21H2
  - Windows 11 version 22H2
  - Windows 10 Version 22H2
  - Windows 11 version 22H3
  - Windows 11 Version 23H2
  - Windows Server 2022, 23H2 Edition (Server Core installation)
  - Windows 11 Version 24H2
  - Windows Server 2025
  - Windows Server 2025 (Server Core installation)
[+] baseScore: 8.8
[+] Exploitation: none
[+] Impact: total
```

[Figure 22]: JSON output

8. Code perspective

It is time to leave all important architecture details aside for a moment and start to review initial aspects on the hypervisor binaries from the static analysis point of view before returning to the dynamic experiments. During the initialization and loading of the hypervisor, three components are critical, but other components are also relevant, and a few superficial definitions follow below:

- **hvix64.exe:** it is responsible for loading and initializing the hypervisor itself as well establishing the correct memory management and CPU configuration to the virtualization. Unfortunately, Microsoft does **not offer any symbol of hvix64.exe**.

- **hvloader.dll:** as the name suggests, this DLL is loaded and used for loading the hypervisor to ensure that it is loaded into the memory and also offers an appropriate environment for its working. Similar to hvix64.exe, Microsoft **does not offer any symbol of it too**.

- **winload.efi:** it is the actual EFI Windows boot loader, which loads boot drivers and also offers the correct environment for managing virtual machines' memory. Unlike the other two components, **we have symbols offered by Microsoft**, which help to analyze and debug it.

- **winhv.sys and winhvr.sys**: The winhv.sys is loaded only when the Windows is running in a child partition and exposes hypercalls available in this child partition while winhvr.sys (kernel hypervisor interface) loads when the OS is running in the root partition but exposes hypercalls available in both child and root partitions. At the end, both are responsible for facilitating the communication between the host system (root partition) and the virtual machines (child partitions).

- **sercurekernel.exe:** as expected, it is the main component of the Secure Kernel implementation, which runs on VTL 1 | CPL 0. Secure Kernel offers a wide range of concepts and ideas, and we will return to it later.

There are many other important files to be considered, and all of them are located in *C:\Windows\System32\drivers* (kernel-mode side):

- **vmbusr.sys:** it contains VMbus associated code.
- **Vid.sys:** it is the virtualization infrastructure driver.
- **storvsp.sys:** it holds code for managing paravirtualized storage.
- **vpci.sys**: It contains code related to paravirtualized PCI.

Other ones take care of different components already mentioned, and work in the user-mode side:

- **vmwp.exe:** it manages resources and respective status for guest virtual machines such as virtual and non-virtual device access.
- **vmms.exe**: it provides state management for virtual machines, but without offering an additional interface for attacks by adversaries, and at the same time, it works in conjunction with the vmwp.exe.

To get a better comprehension of the possible code involved, it is appropriate time to quickly mention further details about hypercalls, which as normal system calls, accept a set of input parameters and return results through output parameters, which hold data that are aligned 8 bytes. Of course, not all hypercalls (invoked by using **HV_X64_MSR_HYPERCALL** – 0x4000001) follow this calling convention, and some of them receive one or two parameters, and not even return an output. Independent of the referred hypercall, there are two classes (or types) of them that can be **simple** or **rep** (from repeat), and the latter is actually a small sequence of hypercalls. In the other hand, we have three current calling conventions: **standard hypercalls** (input and output parameters passed as RDX and R8 registers), **fast hypercalls** (only input parameters) and **extended fast hypercalls** (similar to fast one but accept floating-pointer registers too). In general, hypercalls are specified through a 64-bit value provided by callers, and the 16 first bits specify which hypercall is requested, and it is necessary to know that, for 64-bit, we have the following convention:

- **RCX:** Hypercall Input Value
- **RDX:** Input Parameters
- **R8:** Output Parameters
- **RAX**: result value and returned parameters, if any.

The returned has also 64-but, and the first 16 bits indicates the **HV_STATUS**. Based on the VTL concepts that we reviewed previously, the **HvCallVtlCal**l is one of the most interesting hypercalls because it initiates exactly the transition between the current VTL and the next higher one. As we do not have any further information about hypercalls because there are no symbols for **hvix64.exe**, the most important sources of information are **winhvr.sys, winhv.sys, securekernel.exe** and, for sure, **ntoskrnl.exe**. Other files, such as **winload.efi** and **hvloader.dll** also share common code between them.

Before proceeding our discussing on hypercalls, I would like to remember you that, In general, I will be using Windows 11 (mainstream and also Windows Insider), but you can also use any recent version that will not effectively change results so much. As I had mentioned previously, we have symbols for almost every important file, but we don't have any for **hvax64.exe, hvix64.exe** and **hvloader.dll**. If you want to check available symbols, one suggestion is to create a list of files and use **symchk.exe** as shown below:

```
C:\Users\Administrator>symchk /obdvt /om hyperv_manifest.txt /r /it Documents\hyperv_files.txt /s srv*c:\symbols*https://msdl.microsoft.com/download/symbols
SYMCHK: C:\Windows\System32\drivers\passthruparser.sys [10.0.26100.1150 ] PASSED  - PDB: passthruparser.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\storvsp.sys [10.0.26100.1930 ] PASSED  - PDB: storvsp.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\vhdmp.sys [10.0.26100.2222 ] PASSED  - PDB: vhdmp.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\vhdparser.sys [10.0.26100.1930 ] PASSED  - PDB: vhdparser.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\Vid.sys [10.0.26100.2222 ] PASSED  - PDB: Vid.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\vmbus.sys [10.0.26100.1930 ] PASSED  - PDB: vmbus.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\vmbusr.sys [10.0.26100.1930 ] PASSED  - PDB: vmbusr.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\vpci.sys [10.0.26100.1930 ] PASSED  - PDB: vpci.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\vpcivsp.sys [10.0.26100.2130 ] PASSED  - PDB: vpcivsp.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\winhv.sys [10.0.26100.1930 ] PASSED  - PDB: winhv.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\drivers\winhvr.sys [10.0.26100.2130 ] PASSED  - PDB: winhvr.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\hvax64.exe [10.0.26100.2222 ] FAILED  - hvax64.pdb mismatched or not found
SYMCHK: C:\Windows\System32\hvix64.exe [10.0.26100.2222 ] FAILED  - hvix64.pdb mismatched or not found
SYMCHK: C:\Windows\System32\hvloader.dll [10.0.26100.2200 ] FAILED  - hvloader.pdb mismatched or not found
SYMCHK: C:\Windows\System32\hvhostsvc.dll [10.0.26100.2222 ] PASSED  - PDB: hvhostsvc.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmbuspipe.dll [10.0.26100.1930 ] PASSED  - PDB: vmbuspipe.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmbuspiper.dll [10.0.26100.1930 ] PASSED  - PDB: vmbuspiper.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmbusvdev.dll [10.0.26100.1150 ] PASSED  - PDB: vmbusvdev.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmchipset.dll [10.0.26100.2222 ] PASSED  - PDB: vmchipset.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmcompute.dll [10.0.26100.2222 ] PASSED  - PDB: vmcompute.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmcompute.exe [10.0.26100.2222 ] PASSED  - PDB: vmcompute.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmconnect.exe [10.0.26100.2222 ] PASSED  - PDB: vmconnect.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmdebug.dll [10.0.26100.1150 ] PASSED  - PDB: vmdebug.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmdynmem.dll [10.0.26100.1150 ] PASSED  - PDB: vmdynmem.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmemulateddevices.dll [10.0.26100.2200 ] PASSED  - PDB: VmEmulatedDevices.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\VmEmulatedNic.dll [10.0.26100.1150 ] PASSED  - PDB: VmEmulatedNic.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\VmEmulatedStorage.dll [10.0.26100.1150 ] PASSED  - PDB: VmEmulatedStorage.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmicrdv.dll [10.0.26100.1150 ] PASSED  - PDB: vmicrdv.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmictimeprovider.dll [10.0.26100.1150 ] PASSED  - PDB: vmictimeprovider.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmicvdev.dll [10.0.26100.1150 ] PASSED  - PDB: vmicvdev.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmms.exe [10.0.26100.2222 ] PASSED  - PDB: vmms.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmrdvcore.dll [10.0.26100.1150 ] PASSED  - PDB: vmrdvcore.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmserial.dll [10.0.26100.1930 ] PASSED  - PDB: vmserial.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmsif.dll [10.0.26100.2222 ] PASSED  - PDB: vmsif.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmsifproxystub.dll [10.0.26100.1150 ] PASSED  - PDB: vmsifproxystub.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmsmb.dll [10.0.26100.1930 ] PASSED  - PDB: vmsmb.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmsp.exe [10.0.26100.1    ] PASSED  - PDB: vmsp.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmsynthfcvdev.dll [10.0.26100.1150 ] PASSED  - PDB: VmSynthFcVdev.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\VmSynthNic.dll [10.0.26100.1150 ] PASSED  - PDB: VmSynthNic.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmsynthstor.dll [10.0.26100.1150 ] PASSED  - PDB: VmSynthStor.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmtpm.dll [10.0.26100.1150 ] PASSED  - PDB: vmtpm.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmuidevices.dll [10.0.26100.2213 ] PASSED  - PDB: VmUiDevices.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmusrv.dll [10.0.26100.1150 ] PASSED  - PDB: vmusrv.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmwp.exe [10.0.26100.2222 ] PASSED  - PDB: vmwp.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmwpctrl.dll [10.0.26100.1150 ] PASSED  - PDB: vmwpctrl.pdb DBG: <N/A>
SYMCHK: C:\Windows\System32\vmprox.dll [10.0.26100.1150 ] PASSED  - PDB: vmprox.pdb DBG: <N/A>

SYMCHK: FAILED files = 3
SYMCHK: PASSED + IGNORED files = 43
```

[Figure 23]: Check symbols availability

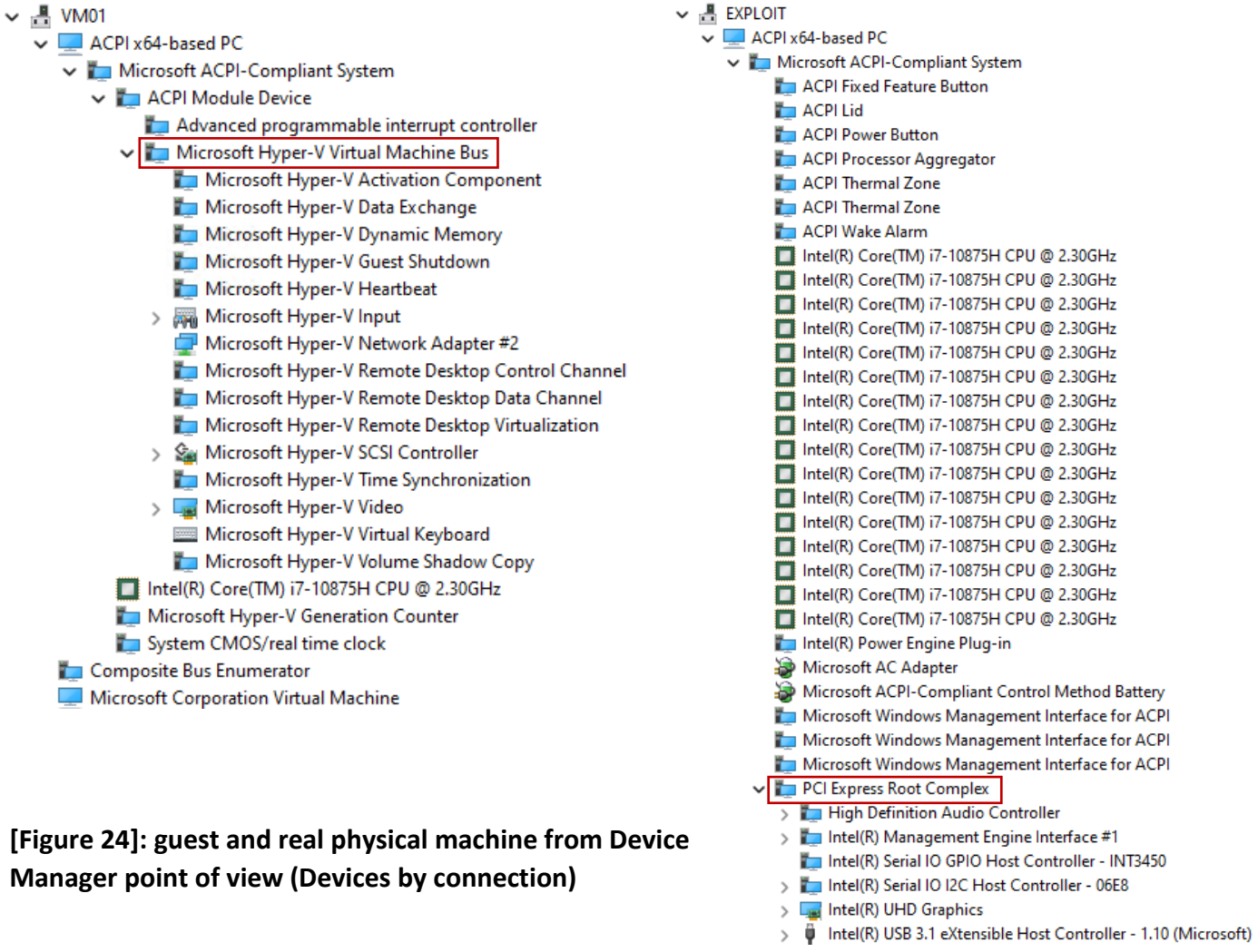
To make clear, we used the following command against a text file that contains a list of important files names, including their respective path, which make part of the Hyper-V on Windows:

- **symchk /obdvt /om hyperv_manifest.txt /r /it Documents\hyperv_files.txt /s srv*c:\symbols*https://msdl.microsoft.com/download/symbols**

To download such symbols using the created hyperv_manifest.txt file and save them into C:\Symbols directory, execute the following command:

- **symchk /im hyperv_manifest.txt /s C:\Symbols**

Returning to hypercalls topic, they are most of the time involved with any Hyper-V operations and, as many of them end to virtual devices and inevitably the data and command flow will pass through the **VMBus** (usually from the child partition to the root partition). However, as VMBus is a macro component, readers probably already know that there are many subcomponents involved and interconnected, and all of them exposed by ACPI, as shown below:



[Figure 24]: guest and real physical machine from Device Manager point of view (Devices by connection)

While there is not absolutely nothing new here, we can trace a kind of parallel with physical machines (on the right), where the Root Complex is the real center and holder of PCIe components on the system, and the real point of communication between hardware and software.

At the same way, VMBus has a similar role within a Hyper-V virtual machine. In both cases, the **ACPI (Advanced Configuration and Power Interface)** describes and exposes their components (from **Root Complex** and **VMBus**) to the operating system. If readers want to collect further information about VMBus, it can use **!devnode 0 1** on WinDbg to check them:

```
DevNode 0xffffb904aaf8eab0 for PDO 0xffffb904aaf7fb00
  InstancePath is "ACPI\VMBus\0"
  ServiceName is "vmbus"
  State = DeviceNodeStarted (0x30a)
  Previous State = DeviceNodeEnumerateCompletion (0x30f)
  DevNode 0xffffb904aafcbab0 for PDO 0xffffb904aafc3d40
    InstancePath is "VMBUS\PowerDevice\PowerDevice0"
    State = DeviceNodeStarted (0x30a)
    Previous State = DeviceNodeEnumerateCompletion (0x30f)
  DevNode 0xffffb904ab6ef830 for PDO 0xffffb904ab6d0d10
    InstancePath is "VMBUS\{57164f39-9115-4e78-ab55-382f3bd5422d}\{fd149e91-82e0-4a7d-afa6-2a4166cbd7c0}"
    State = DeviceNodeStarted (0x30a)
    Previous State = DeviceNodeEnumerateCompletion (0x30f)
  DevNode 0xffffb904ab6efbe0 for PDO 0xffffb904ab63acf0
    InstancePath is "VMBUS\{a9a0f4e7-5a45-4d96-b827-8a841e8c03e6}\{242ff919-07db-4180-9c2e-b86cb68c8c55}"
    State = DeviceNodeStarted (0x30a)
    Previous State = DeviceNodeEnumerateCompletion (0x30f)
  DevNode 0xffffb904ab6f0010 for PDO 0xffffb904ab6d1d40
    InstancePath is "VMBUS\{0e0b6031-5213-4934-818b-38d90ced39db}\{b6650ff7-33bc-4840-8048-e0676786f393}"
    State = DeviceNodeStarted (0x30a)
    Previous State = DeviceNodeEnumerateCompletion (0x30f)
  DevNode 0xffffb904ab6f03c0 for PDO 0xffffb904ab6d4800
    InstancePath is "VMBUS\{9527e630-d0ae-497b-adce-e80ab0175caf}\{2dd1ce17-079e-403c-b352-a1921ee207ee}"
    State = DeviceNodeStarted (0x30a)
    Previous State = DeviceNodeEnumerateCompletion (0x30f)
  DevNode 0xffffb904ab6f0770 for PDO 0xffffb904ab6d6bf0
    InstancePath is "VMBUS\{35fa2e29-ea23-4236-96ae-3a6ebacba440}\{2450ee40-33bf-4fbd-892e-9fb06e9214cf}"
```

[Figure 25]: WinDbg session

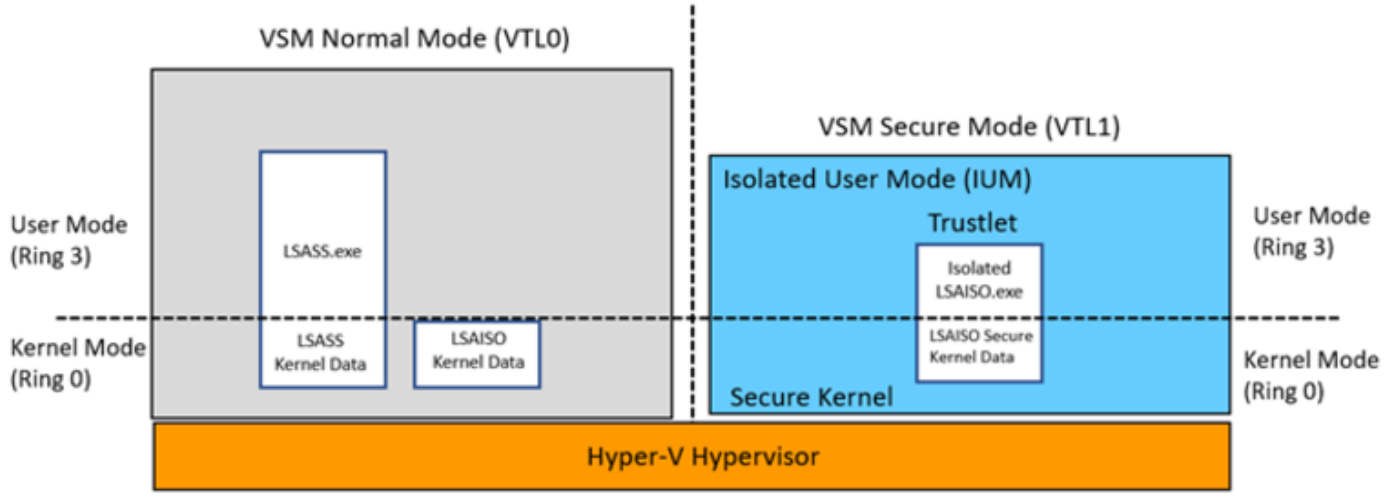
The VMBus and respective channels above, which have been exposed by the ACPI, are used and transformed into nodes by the own guest operating system. This transformation is more relevant whether we remember, once more, the basic flow:

- I/O Stack (child partition) → **VSC (child partition)** → VMbus (child partition) → hypervisor (synthetic interrupt) →VMbus (root partition) → **VSP (root partition)** → I/O Stack (root partition) → hypervisor → Devices

Clearly the **VSC** and **VSP** are virtualization peers responsible for transferring data back-and-forth, as expected, they are part of the attack surface because there are drivers involved (**vpci.sys** on the guest side and **vpcivsp. sys** root partition side) for establishing and making communication and data transfer possible. However, the most meaningful point is understand that real (physical) hardware devices are directly exposed, skipping the hypervisor, thought the **VMBus** to the guest via **vPCI (virtual PCI)** by using mechanisms such as **Single Root I/O Virtualization | SR-IOV** (https://learn.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-) or even **Discrete Device Assignment | DDA** (https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/plan/plan-for-deploying-devices-using-discrete-device-assignment). A valid point such exposition of devices to the guest only happens when requested.

The **VSC** and **VSC** are associated with communication, regardless of any physical device that is removed or added because the **vPCI** abstracts such access and makes any event transparent to upper layers.

A good point to highlight is that people might think, most Hyper-V vulnerabilities (and attack surface) are not in the hypervisor, but in the root partition. Certainly, resources and, specifically, hardware aspects are not the only that deserve to be investigated, and targets from **VTL1 (Virtual Secure Mode)** should be investigated, and also mainly because it supports a series of features such **as Credential Guard, TPMs, Device Guard** and other ones. One of these examples would be **IUM (Isolated User Mode)** processes (as knows as **Trustlets**, which are programs that run VTL1 as, for example, **LSAISO.exe (LSA isolated)**, and an appropriate visualization is given by Microsoft:



[Figure 26]: Isolated User Mode (IUM) | Credits: Microsoft

Handling VTL1 brings up natural challenges for us because it is necessary to learn how to debug such as trustlets/IUMs (they are not viable to debug by default) and, as expected, how to debug the whole **VTL1**, which we will touch again in this and articles continuing the current one.

When we talk on IUM topic, it is unavoidable not returning in the discussion about transition between the **non-secure mode (VTL0)** and **secure mode (VTL1)** world because, as stated in previous sections and subsections, hypercalls are the center of everything to do the transition between VTL0 and VTL1, but also vice-versa. The transition from VTL0 to VTL1 is initiated by invoking **VslpEnterIumSecureMode**, **HvlSwitchToVsmVtl1**, **HvlpSwitchToVsmVtl1RetpolineHelper** and **HvlpVsmVtlCallVa**, which emits the final call including **0x11 hypercall**. At the end of this process, communication between the secure mode and the hypervisor is kept and the Secure Kernel is able to map and access VTL0 memory. Finally, the transition back starts when the Secure Kernel invokes a **HvVtlReturn hypercall**.

By the way, as hypercalls have their associated handler and, in terms of assembly code, readers can locate them by using respective read and write operations. but there are multiple conditions that must be fulfilled for such handlers are available to be invoked, and of the critical one is the virtual processor state. Furthermore, and honestly, I am repeating myself to highlight such foundations once more, hypercalls are invoked from a **privileged level (CPL == 0)** through the processor hypercall interface, where from VTL0 to VTL1 the sequence is **VslpEnterIumSecureMode + vmcall(0x11)**, and from VTL1 to VTL0 the sequence is **SkCallNormalMode + vmcall(0x12).** The value used by vmcall is known as **SSCN (Secure Service Call Number)**.

Besides every technical fact mentioned so far, readers should remember that there are protections around every corner in the normal kernel such as **KASLR, Hyperguard, HVCI (Hypervisor-enforced Code Integrity), CFG (Control Flow Guard), XFG, ACG (Arbitrary Code Guard), CIG (Code Integrity Guard), EPT**

enforcement, R^W, MTE (Memory Tagging Extension) on ARM architecture, and other ones. Therefore, security researchers need to pay attention to different classes of vulnerabilities such as race conditions, integer overflow, out-of-boundary vulnerabilities (it could be mitigated using std::vectors and safe functions), and memory bugs such stack overflow, type confusions, heap overflow, UAF (mitigated using smart pointer and, in ARM processors, by MTE), double-free, uninitialized variables and pointers (they can cause leaks), and other ones. In special type confusions can be a hard problem to manage because this kind of vulnerability can have assorted reasons and origins:

- **logical issues:** they are always extremely complex problems because there is not a standard procedure to locate such errors.

- **incorrect static castings:** static_cast is a classic compile-type casting used for converting types (fundamental and related ones), but that can produce serious side-effects when converting two unrelated or incompatible types.

- **reinterpret_cast:** it is another type of casting, and it is usually used for casting a pointer type to another one, including unrelated (or incompatible) pointer, or even converting a pointer to an integer type. As this kind of cast does not perform any verification, wrong operations can generate complications and even vulnerabilities.

In terms of exploitation itself, you should remember that most of the time KASLR can be bypassed by using either a leak or even partial overwriting, but both cases might be unlikely depending on the context. In Hyper-V we could launch fuzzing sessions aiming to find bugs in the hypervisor from the root partition, child partitions or even a nested root partition (as known as L2) against the mentioned hypervisor. Obviously, the hypervisor is not the only valid target, and we might test hypercalls, involved drivers, which include drivers using **MMIO (Memory Mapped I/O)** an **MmMapIoSpace(Ex)** function (device memory is actually device registers -- is usually mapped in the memory, but not always RAM memory -- through this function), **MCFG table** (including PCI configuration and also MMIO information), and other components.

I quickly mentioned **MmMapIoSpace** and **MmMapIoSpaceEx**, which maps a physical address range to non-paged system space, because readers likely know that they map write-and-executable memory in systems without VBS enabled and allow to specify page protections, respectively, and a misunderstanding of how it works can cause grave vulnerabilities. Anyway, you could easily set breakpoints on these functions on WinDbg and check their operation.

The subject related to PCI bus and associated components can be strange at first sight, and eventually to comment a few words can help to get an initial understanding. Basically, communication at the bus level is composed of reading and writing operations in memory, which is represented by RAM type -- most of the time, but not always -- and device memory, being this one involved with communication to a device. Thus, **MMIO (Memory-Mapped I/O)** works as a mechanism used for communication through Transmission Layer Packets (TLPs) between computer components and devices, where registers of such devices are mapped into the memory.

One of differences between **MMIO** and **DMA (Direct Memory Access)**, which readers have already heard about, is who initiates the communication and while MMIO communication is started by CPU, in DMA context it is **asynchronously** started by the device. In this memory context we have **Extended Configuration Access Management** (as known as **ECAM**), which maps the PCI configuration to the physical memory address and whose location and management is guided by the kernel, **acpi.sys** and **pci.sys** drivers.

Therefore, ECAM works like a centralized repository containing configuration settings of devices and even applications. As expected, the ECAM (a hardware mechanism) is strongly related to the **MCFG (Multiple Configuration)**, which is a table that comes from the ACPI and that provides information on the address ranges used for the PCI configuration space that will be used by the operating system. In simple and imprecise words, it would be something like **ECAM (firmware) → MCFG (APCI) → OS**.

To bring up a bit more of concepts, memory regions that a PCI device can access are defined for each device by a set of **BARs (Base Address Registers)**, which are used to map device memory (device registers) into the reserved memory. Thus, for each device its PCI configuration space is parsed and allocated memory regions that are indicated by BARs that point to the starting address of such regions.

Just as a basic demonstration before we proceed with our hypervisor's content, it is possible to locate and dump and MCFG as shown below:

```
1: kd> !acpicache
Dumping cached ACPI tables...
  XSDT @(fffff7d3c0003018) Rev: 0x1 Len: 0x00006c TableID: 440BX
  MCFG @(fffff7d3c0004018) Rev: 0x1 Len: 0x00003c TableID: EFIMCFG
  FACP @(fffff7d3c000f018) Rev: 0x4 Len: 0x0000f4 TableID: 440BX
  SRAT @(fffff7d3c0010018) Rev: 0x3 Len: 0x000110 TableID: EFISRAT
  WAET @(fffff7d3c0011018) Rev: 0x1 Len: 0x000028 TableID: VMW WAET
  APIC @(fffff7d3c0013018) Rev: 0x3 Len: 0x00007a TableID: EFIAPIC
  HPET @(fffff7d3c0017018) Rev: 0x1 Len: 0x000038 TableID: VMW HPET
  TCPA @(ffffe609657164a8) Rev: 0x2 Len: 0x000032 TableID: EDK2
  WSMT @(ffffe60965716938) Rev: 0x1 Len: 0x000028 TableID: VMW WSMT
  TPM2 @(ffffe60965716988) Rev: 0x3 Len: 0x00004c TableID: VMW_TPM2
  DSDT @(ffffe609658db018) Rev: 0x1 Len: 0x018af1 TableID: Custom
1: kd> !acpitable MCFG
HEADER - fffff7d3c0004018
  Signature:                MCFG
  Length:                   0x0000003c
  Revision:                 0x01
  Checksum:                 0x6a
  OEMID:                    VMWARE
  OEMTableID:               EFIMCFG
  OEMRevision:              0x06040001
  CreatorID:                VMW
  CreatorRev:               0x000007ce
BODY - fffff7d3c000403c
fffff7d3`c000403c  00 00 00 00 00 00 00 00-00 00 00 e0 00 00 00 00
fffff7d3`c000404c  00 00 00 7f 00 00 00 00
1: kd> !dw [uc] e0000000
#e0000000 8086 7190 0006 0200 0001 0600 0000 0000
#e0000010 0000 0000 0000 0000 0000 0000 0000 0000
#e0000020 0000 0000 0000 0000 0000 0000 15ad 1976
#e0000030 0000 0000 0000 0000 0000 0000 0000 0000
#e0000040 0401 0000 0008 ffc0 0000 0000 0000 0000
#e0000050 0000 0000 0000 0000 1000 3001 3001 0001
#e0000060 0080 0080 0080 0080 0000 0000 0000 0000
#e0000070 0000 0010 0000 0000 0000 0000 0000 0000
1: kd> !dw [uc] e0000000 L1
#e0000000 8086
```

[Figure 27]: MCFG from ACPI via WinDbg

My demonstration above happened in a VMware virtual machine, but it works in a physical machine, of course. Be aware that I was debugging the kernel.

As we can see it was not hard to retrieve basic information on the MCFG from the **ACPI (Advanced Configuration and Power Interface)**. The pending task is how to interpret the output correctly, and we need to search for the ACPI specification and, in particular, the MCFG structure header to decode and understand its respective fields.

Once we are able to interpret the output then we can locate the ECAM and get further information:

Offset	Length	Description
0	4	Table Signature ("MCFG")
4	4	Length of table (in bytes)
8	1	Revision (1)
9	1	Checksum (sum of all bytes in table & 0xFF = 0)
10	6	OEM ID (same meaning as other ACPI tables)
16	8	OEM table ID (manufacturer model ID)
24	4	OEM Revision (same meaning as other ACPI tables)
28	4	Creator ID (same meaning as other ACPI tables)
32	4	Creator Revision (same meaning as other ACPI tables)
36	8	Reserved
44 + (16 * n)	16	Configuration space base address allocation structures. Each structure uses the following format:

Offset	Length	Description
0	8	Base address of enhanced configuration mechanism
8	2	PCI Segment Group Number
10	1	Start PCI bus number decoded by this host bridge
11	1	End PCI bus number decoded by this host bridge
12	4	Reserved

[Figure 28]: MCFG table | credit: https://wiki.osdev.org/PCI_Express

The almost whole MCFG is decoded by WinDbg exactly according to the tables shown above, excepting the Reserved field (offset 36). The body itself is also not decoded, but we can see that the base address of the ECAM is given by first 8 bytes after Reserved field (8 bytes) and, as it is in little endian order, the value is e0000000.

Thus, as shown in **Figure 27**, I checked the respective physical addresses and found that the vendor ID of 00:00.0 (given by the first two bytes of the **Configuration Space (ECAM)** and remember that the format is **Bus:Device.Function,Offset**) is 8086 (Intel). As you can see, it is not a very smart way to collect such

information and serves only for education purposes. The best way to do it is using the **pci command** to dump the entire information for the same hardware (00:00.0)

```
1: kd> !pci 100 0 0 0

PCI Configuration Space (Segment:0000 Bus:00 Device:00 Function:00)
Common Header:
    00: VendorID       8086 Intel Corporation
    02: DeviceID       7190
    04: Command        0006 MemSpaceEn BusInitiate
    06: Status         0200 DEVSELTiming:1
    08: RevisionID     01
    09: ProgIF         00
    0a: SubClass       00 Host Bridge
    0b: BaseClass      06 Bridge Device
    0c: CacheLineSize  0000
    0d: LatencyTimer   00
    0e: HeaderType     00
    0f: BIST           00
    10: BAR0           00000000
    14: BAR1           00000000
    18: BAR2           00000000
    1c: BAR3           00000000
    20: BAR4           00000000
    24: BAR5           00000000
    28: CBCISPtr       00000000
    2c: SubSysVenID    15ad
    2e: SubSysID       1976
    30: ROMBAR         00000000
    34: CapPtr         00
    3c: IntLine        00
    3d: IntPin         00
    3e: MinGnt         00
    3f: MaxLat         00
Device Private:
    40: 00000401 ffc00008 00000000 00000000
    50: 00000000 00000000 30011000 00013001
    60: 00800080 00800080 00000000 00000000
    70: 00100000 00000000 00000000 00000000
    80: 00000000 00000000 00000000 00000000
    90: 00000000 00000000 00000000 00000000
    a0: 00000000 00000000 00000000 00000000
    b0: 00000000 00000000 00000000 00000000
    c0: 00000013 e0000003 00000000 00000000
    d0: 00000000 00000000 00000000 00000000
    e0: 00000000 00000000 00000000 00000000
    f0: 00000000 00000000 00000000 00400000
```

[Figure 29]: Reading ECAM information for a given hardware

I have just demonstrated these manual steps to provide you with some context related to hardware and buses, in general, because when research about hypervisors such knowledge can be useful.

There are tools (like **RWEverything**) and also open-source tools such as **ACPICA project** (https://github.com/acpica/acpica/) that do a much better job, by far. Additionally, they already provide us with all necessary binaries that are available on https://github.com/acpica/acpica/releases/tag/R2024_12_12. An example of the **ACPI table** of a physical machine follows below:

```
C:\Users\Administrator\Downloads\iasl-win-20241212>acpidump.exe > phys_acpi_table.dat

C:\Users\Administrator\Downloads\iasl-win-20241212>acpixtract.exe -l phys_acpi_table.dat

Intel ACPI Component Architecture
ACPI Binary Table Extraction Utility version 20241212
Copyright (c) 2000 - 2023 Intel Corporation

  Signature  Length    Version Oem       Oem         Oem         Compiler Compiler
                               Id        TableId     RevisionId  Name     Revision
  ---------  ----------  ----  --------  ----------  ----------  -------  ----------

  01)  DBGP  0x00000034  0x01  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x01000013
  02)  MCFG  0x0000003C  0x01  "ALASKA"  "A M I   "  0x01072009  "MSFT"   0x00000097
  03)  FACP  0x00000114  0x06  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x00010013
  04)  APIC  0x0000012C  0x04  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x00010013
  05)  DMAR  0x000000A8  0x01  "INTEL "  "EDK2    "  0x00000002  "    "   0x01000013
  06)  HPET  0x00000038  0x01  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x01000013
  07)  FPDT  0x00000044  0x01  "ALASKA"  "CML     "  0x01072009  "AMI "   0x01000013
  08)  SSDT  0x00002087  0x02  "CpuRef"  "CpuSsdt "  0x00003000  "INTL"   0x20160527
  09)  FIDT  0x0000009C  0x01  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x00010013
  10)  MSDM  0x00000055  0x03  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x01000013
  11)  SSDT  0x000031F4  0x02  "SaSsdt"  "SaSsdt  "  0x00003000  "INTL"   0x20160527
  12)  SSDT  0x000028DA  0x02  "Intel "  "PegSsdt "  0x00001000  "INTL"   0x20160527
  13)  SSDT  0x000017D4  0x02  "ALASKA"  "CmlH_Tbt"  0x00001000  "INTL"   0x20160527
  14)  SSDT  0x00000E3D  0x02  "ALASKA"  "Ther_Rvp"  0x00001000  "INTL"   0x20160527
  15)  SSDT  0x000031E6  0x02  "INTEL "  "xh_cfht4"  0x00000000  "INTL"   0x20160527
  16)  NHLT  0x0000002D  0x00  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x01000013
  17)  SSDT  0x000006BB  0x02  "Intel "  "PerfTune"  0x00001000  "INTL"   0x20160527
  18)  LPIT  0x00000094  0x01  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x01000013
  19)  SSDT  0x00001DD8  0x02  "ALASKA"  "TbtTypeC"  0x00000000  "INTL"   0x20160527
  20)  DBG2  0x00000054  0x00  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x01000013
  21)  SSDT  0x0000135D  0x02  "ALASKA"  "UsbCTabl"  0x00001000  "INTL"   0x20160527
  22)  SSDT  0x000000AE  0x02  "SgRef "  "SgPeg   "  0x00001000  "INTL"   0x20160527
  23)  BGRT  0x00000038  0x01  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x00010013
  24)  TPM2  0x0000004C  0x04  "ALASKA"  "A M I   "  0x00000001  "AMI "   0x00000000
  25)  SSDT  0x00002753  0x01  "OptRf1"  "Opt1Tabl"  0x00001000  "INTL"   0x20160527
  26)  WSMT  0x00000028  0x01  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x00010013
  27)  XSDT  0x000000F4  0x01  "ALASKA"  "A M I   "  0x01072009  "AMI "   0x01000013
  28)  DSDT  0x00041E4A  0x02  "ALASKA"  "A M I   "  0x01072009  "INTL"   0x20160527

Found 28 ACPI tables in phys_acpi_table.dat
```

[Figure 30]: Using an appropriate tool to list the ACPI table

As readers can notice, the output for a physical machine is longer than for a physical one, as expected, and the list of signatures is also even more attractive.

The good news is that the same tool is able to extract the **MCFG** from the **ACPI table** and there is a specific option to accomplish this task. While I recommended readers to try **acpixtract.exe --help** to list all available options, notice the **-s option** allows us to extract any table and, in this case, we are interested in retrieving the **MCFG**. Therefore, we can extract and decode the referred table, and then visualize it using a Windows built in command as shown below:

```
C:\Users\Administrator\Downloads\iasl-win-20241212>acpixtract.exe -s MCFG phys_acpi_table.dat

Intel ACPI Component Architecture
ACPI Binary Table Extraction Utility version 20241212
Copyright (c) 2000 - 2023 Intel Corporation

  MCFG -      60 bytes written (0x0000003C) - mcfg.dat

C:\Users\Administrator\Downloads\iasl-win-20241212>iasl mcfg.dat

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20241212
Copyright (c) 2000 - 2023 Intel Corporation

File appears to be binary: found 37 non-ASCII characters, disassembling
Binary file appears to be a valid ACPI table, disassembling
Input file mcfg.dat, Length 0x3C (60) bytes
ACPI: MCFG 0x0000000000000000 00003C (v01 ALASKA A M I    01072009 MSFT 00000097)
Acpi Data Table [MCFG] decoded
Formatted output:  mcfg.dsl - 1543 bytes

C:\Users\Administrator\Downloads\iasl-win-20241212>type mcfg.dsl
/*
 * Intel ACPI Component Architecture
 * AML/ASL+ Disassembler version 20241212 (32-bit version)
 * Copyright (c) 2000 - 2023 Intel Corporation
 *
 * Disassembly of mcfg.dat
 *
 * ACPI Data Table [MCFG]
 *
 * Format: [HexOffset DecimalOffset ByteLength]  FieldName : FieldValue (in hex)
 */

[000h 0000 004h]                   Signature : "MCFG"    [Memory Mapped Configuration Table]
[004h 0004 004h]                Table Length : 0000003C
[008h 0008 001h]                    Revision : 01
[009h 0009 001h]                    Checksum : E1
[00Ah 0010 006h]                      Oem ID : "ALASKA"
[010h 0016 008h]                Oem Table ID : "A M I "
[018h 0024 004h]                Oem Revision : 01072009
[01Ch 0028 004h]             Asl Compiler ID : "MSFT"
[020h 0032 004h]       Asl Compiler Revision : 00000097

[024h 0036 008h]                    Reserved : 0000000000000000

[02Ch 0044 008h]                Base Address : 00000000E0000000
[034h 0052 002h]        Segment Group Number : 0000
[036h 0054 001h]            Start Bus Number : 00
[037h 0055 001h]              End Bus Number : FF
[038h 0056 004h]                    Reserved : 00000000

Raw Table Data: Length 60 (0x3C)

    0000: 4D 43 46 47 3C 00 00 00 01 E1 41 4C 41 53 4B 41  // MCFG<.....ALASKA
    0010: 41 20 4D 20 49 20 00 00 09 20 07 01 4D 53 46 54  // A M I ... ..MSFT
    0020: 97 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E0  // ................
    0030: 00 00 00 00 00 00 00 FF 00 00 00 00              // ............
```

[Figure 31]: Decoding MCFG

As readers realized, the interaction between the software and hardware is really important while researching hypervisors, and there are multiple opportunities to spot vulnerabilities of different classes such as race conditions, read-write out-of-boundary, double-fetching (a subclass of race-conditions), integer overflow and other ones, mainly when handling drivers that are responsible for reading, writing and control tasks. Finally, if you expand it to both VSM world (VTL0 and VTL1), it is possible to fuzz the secure kernel from VTL1 too (VTL0 → VTL1) and the opposite direction regarding such hardware and drivers' nuances, and we remember that there are two hypercalls that are responsible for switching from one side to another that is **HvVtlCall( )** and **HvVTLReturn( )**.

Regardless of potential bugs found, this is only part of the work, and we need to evaluate whether the bug is actually a security vulnerability and, if it is, learn how we can exploit it and circumvent all related protections, which demands a great effort since getting a stable way to corrupt structures of the target, finding any read-write-what-where primitives, bypassing all involved protections until gaining access to the system, with a possible elevation of privilege in some point. Eventually protections cause distinct side effects depending on the adopted direction (VTL0 to VTL1 or VTL1 to VTL0).

It is relatively straightforward to realize that many security vulnerabilities arise from interfaces and boundaries, and everything related to interception context. From Hyper-V, VID driver is really an intercept driver, which receives message from child partitions besides providing distinct kinds of management services for the virtual machines.. If we pay attention, the **VM Worker process,** which is created by the **VMM Service**, also intercepts components from virtualization stack, and even there is a hypercall named **HvInstallIntercept** that is responsible for deploying such interception.

Of course, they are natural candidates for the first evaluation. At the same line, the **SynIC (Synthetic Interrupt Controller)** provides inter-communication to virtual machines and the **VMBus root driver (Vmbusr.sys)** also creates a port in the child partition (there are several types of ports such as message ports, event ports and monitor port). As I said, at the end, everything is related to communication, interfaces, and boundaries.

As you might have noticed, I am leaving out of discussion anything related to platform security, which could directly or indirectly to be used to compromise everything else, including operating system and hypervisor. Following the same mindset, I am not considering issues like **Option ROM (OROM)**, for example, which is an extension firmware associated with drivers that provides storage controller and network cards with initialization routines for such hardware pieces and cause serious vulnerabilities. At the end of the day, how many people and companies apply firmware updates for machines, network adapters, storage controllers or even enable the Secure Boot in UEFI even in 2025? You already know the answer.

Furthermore, we have not regarded any real protection against hardware configurations vulnerabilities, which can and will interfere with Hyper-V because, according to past experience, most of the time they really exist, but are not applied. For example, a few years ago, it was very accessible to use S3 scripts, which are activated with S3 resume process, to modify Hyper-V and get full access to Secure VSM world. Another example around the same theme is using **SMM (System Management Mode)** to alter anything inside VTL1 or even the hypervisor and correlated components.

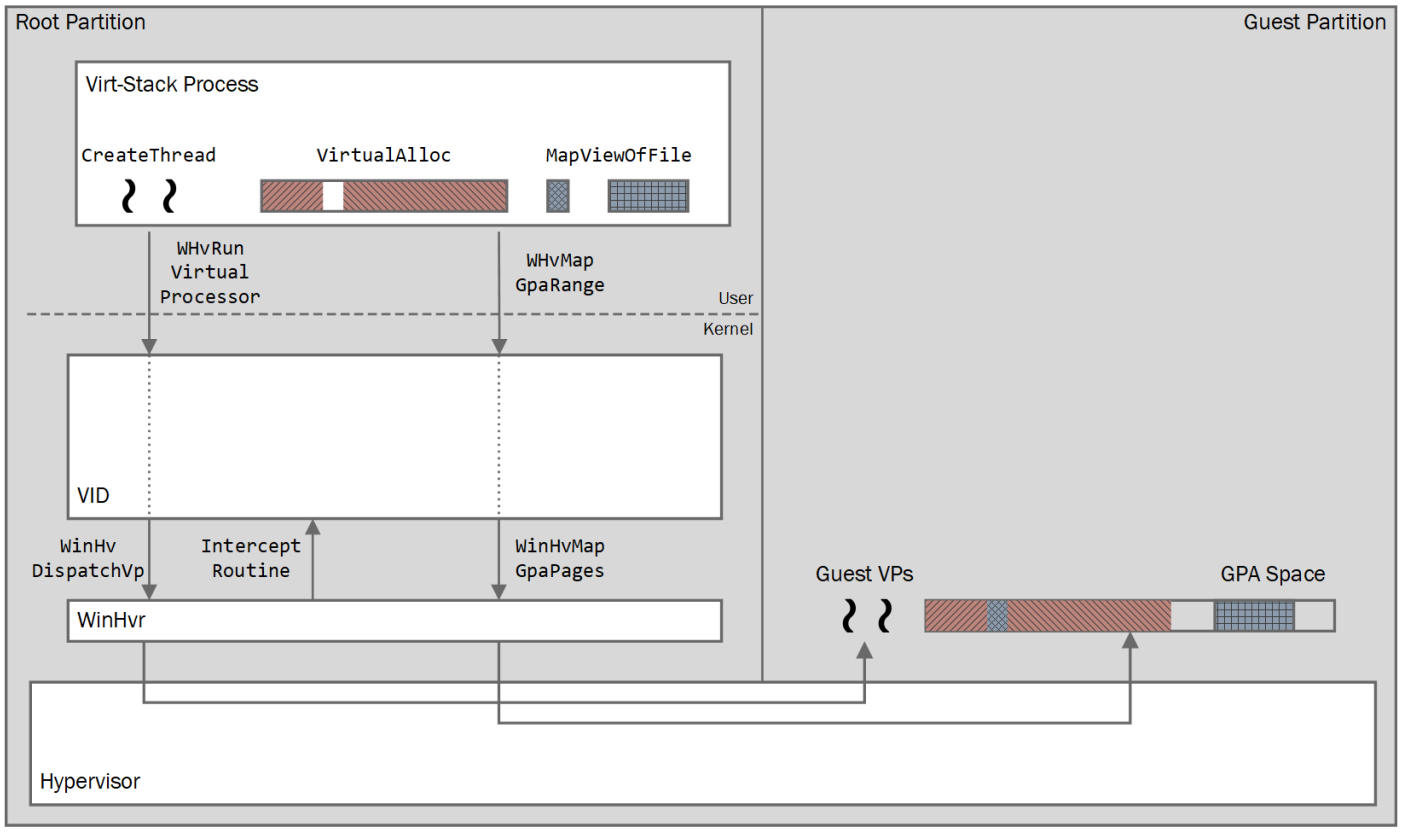
Eventually I could offer a superficial review about key concepts, components and security ideas on Hyper-V and associated mechanism. Honestly, each paragraph here could become an entire section and study because Hyper-V is a really complex software.

9. Reversing notes

This section will present a brief overview of a few details related to reverse engineering. There are multiple interesting files that deserves to be analyzed in depth and have an entire separated articles to analysis details, security issues and consequences, but this article is not about it yet. As symbols are not offered for important files, we will use all available resources to get the maximum information about them:

- Binary diffing
- Public projects
- IDA Pro Lumina
- Windows and Hyper-V development headers
- Tools from Windows, SDK and WDK
- Other Windows binaries and debugging tools, including extensions
- Public windows driver samples

Before moving to our next tasks, as readers will be analyzing code and interaction, eventually the following image of the hypervisor platform architecture might help you:



[Figure 32]: Hypervisor Platform Architecture | credit: Microsoft

Of course, I do not have any plans to do it extensively here, but I would like to show the first steps, which start by creating a folder, copy important files to there, open each one on a disassembler tool such IDA, Binary Ninja or Ghidra, and decompile such files whether this option is available for you (on IDA Pro go to **File | Produce File | Create C File**). As a limited example, I have picked the following files:

- **hvhostsvc.dll:** it provides services (integration) between host and child virtual machines.
- **hvix64.exe:** part of hypervisor (symbols not available).
- **hvloader.dll:** the hypervisor loader library (symbols not available).
- **hvservice.sys:** hypervisor service driver.
- **hvsocket.dll**: makes public socket interfaces, which can be called from user-mode.
- **hvsocket.sys:** manages communication between root and child partitions.
- **hvsocketcontrol.sys:** it loads the HvSocket whether VMBus interface is not available.
- **kdnet.exe:** it is used for network kernel debugging.
- **ntkrla57.exe:** nt kernel file that offers support for 5-level paging.
- **ntoskrnl.exe:** it is the Windows kernel image.
- **securekernel.exe:** it implements the Secure Kernel.
- **securekernella57.exe:** it implements the Secure Kernel, but with 5-level paging support.
- **storvsc.sys:** it provides virtualized storage features for child partitions on virtual machine side.
- **storvsp.sys:** it provides virtualized storage features for child partitions on root-partition side.
- **vid.dll:** it implements communication, cpu state memory management for virtual machines.
- **vid.sys:** it is the virtualization infrastructure driver and provides management services.
- **vmbkmcl.sys:** it implements the VMBus kernel mode client library on child partition side.
- **vmbkmclr.sys:** it implements the VMBus kernel mode client library on root partition side.
- **vmbuspipe.dll:** it offers API and structure's definitions (VMBus) and facilitates communication.
- **vmbuspiper.dll:** equal to the above and offers logging and interception of API calls.
- **vmbusr.sys**: it is the VMBus root driver that coordinates communication to the virtual machine.
- **vmbusvdev.dll:** it implements the VMBus virtual device.
- **vmdebug.dll:** it provides support for debugging Hyper-V components.
- **vmemulatedstorage.dll:** it provides management and emulation for storage operations.
- **vmwp.exe:** it manages virtual machines lifecycle.
- **vmwpctrl.dll:** it implements functions for virtual machines control and management.
- **vpcivsp.sys:** it creates and manages the virtual bus object (associated with virtual devices).
- **winhv.sys:** it exposes the hypercalls available from child partition.
- **winhvemulation.dll:** it implements APIs of Windows hypervisor platform and instruction emulator.
- **winhvplatform.dll:** it implements APIs of Windows hypervisor platform in the user-mode.
- **winhvr.sys**: it exposes the hypercalls available from root and child partition.
- **winload.efi:** it is used by UEFI to load Windows.

Along with this section I will be using the current version of Windows, but readers could prefer using Windows Insider versions (Canary, Dev, Beta or Release Preview Channel) to use the last updates. To automate the copy operation to a given destination folder (**C:\HVFiles** in my example), and being able to expand the list later, a simple script follows:

C: > HVFiles > CopyHVFiles.ps1

```
# List of files to search for
$filesToFind = @(
    "hvhostsvc.dll",
    "hvix64.exe",
    "hvloader.dll",
    "hvservice.sys",
    "hvsocket.dll",
    "hvsocket.sys",
    "hvsocketcontrol.sys"
```

```
    "hvsocketcontrol.sys"
    "kdnet.exe",
    "ntkrla57.exe",
    "ntoskrnl.exe",
    "securekernel.exe",
    "securekernella57.exe",
    "storvsc.sys",
    "storvsp.sys",
    "vid.dll",
    "vid.sys",
    "vmbkmcl.sys",
    "vmbkmclr.sys",
    "vmbuspipe.dll",
    "vmbuspiper.dll",
    "vmbusr.sys",
    "vmbusvdev.dll",
    "vmdebug.dll",
    "vmemulatedstorage.dll",
    "vmms.exe",
    "vmwp.exe",
    "vmwpctrl.dll",
    "vpcivsp.sys",
    "winhv.sys",
    "winhvemulation.dll",
    "winhvplatform.dll",
    "winhvr.sys",
    "winload.efi"
)

# Source and destination directories
$sourceDirs = @("C:\Windows\System32", "C:\Program Files (x86)\Windows
Kits\10\Debuggers\x64")
$destinationDir = "C:\HVFiles"
# Create the destination directory if it doesn't exist
if (-not (Test-Path -Path $destinationDir)) {
    New-Item -ItemType Directory -Path $destinationDir | Out-Null
}

# Search for and copy the files
foreach ($file in $filesToFind) {
    $fileFound = $false
    foreach ($sourceDir in $sourceDirs) {
        $filePath = Get-ChildItem -Path $sourceDir -Filter $file
        -Recurse -ErrorAction SilentlyContinue | Select-Object -First 1
        if ($filePath) {
            Copy-Item -Path $filePath.FullName -Destination
            $destinationDir -Force
            Write-Output "Copied $($filePath.FullName) to
            $destinationDir"
            $fileFound = $true
            break
        }
    }
```

```
58      if (-not $fileFound) {
59          Write-Output "File $file not found in the specified source
            directories"
60      }
61  }
```

[Figure 33]: Script for copying chosen files to a central folder

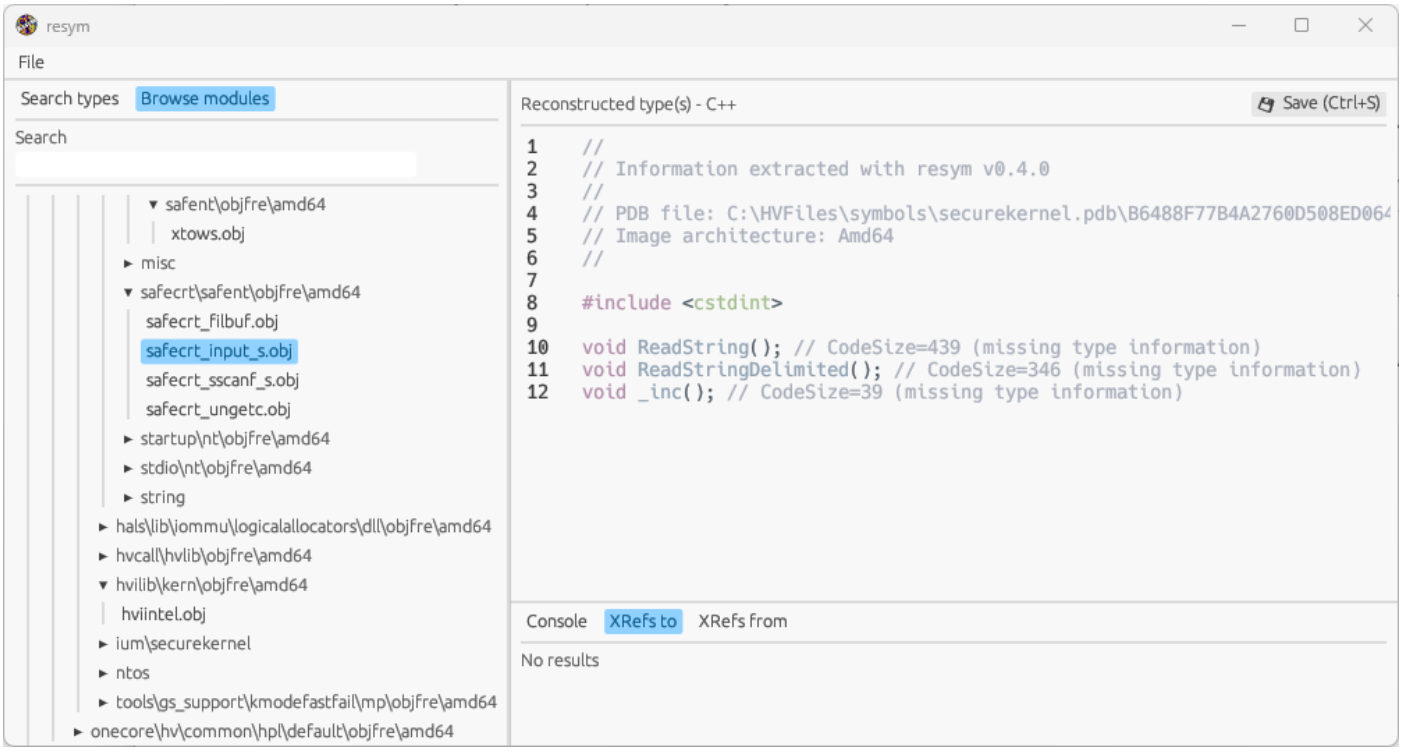
Once again, I have chosen only a few Hyper-V related files, and there are other ones that are important to the daily research. Additionally, we already have almost all symbols for the presented files from Hyper-V, but for the main two files such symbols are not offered, unfortunately:

- **hvix64.exe:** part of hypervisor (symbols not available)
- **hvloader.dll:** the hypervisor loader library (symbols not available)

As mentioned, there are a multitude of ways for trying to populate part of these missing symbols. On the other hand, as already demonstrated too, there are different approaches to getting the existing symbols:

- **symchk /obdvt /om hyperv_manifest.txt /r /it C:\HVFiles\hyperv_files.txt /s srv*C:\HVFiles\symbols*https://msdl.microsoft.com/download/symbols**
- **symchk /im hyperv_manifest.txt /s C:\HVFiles\symbols**

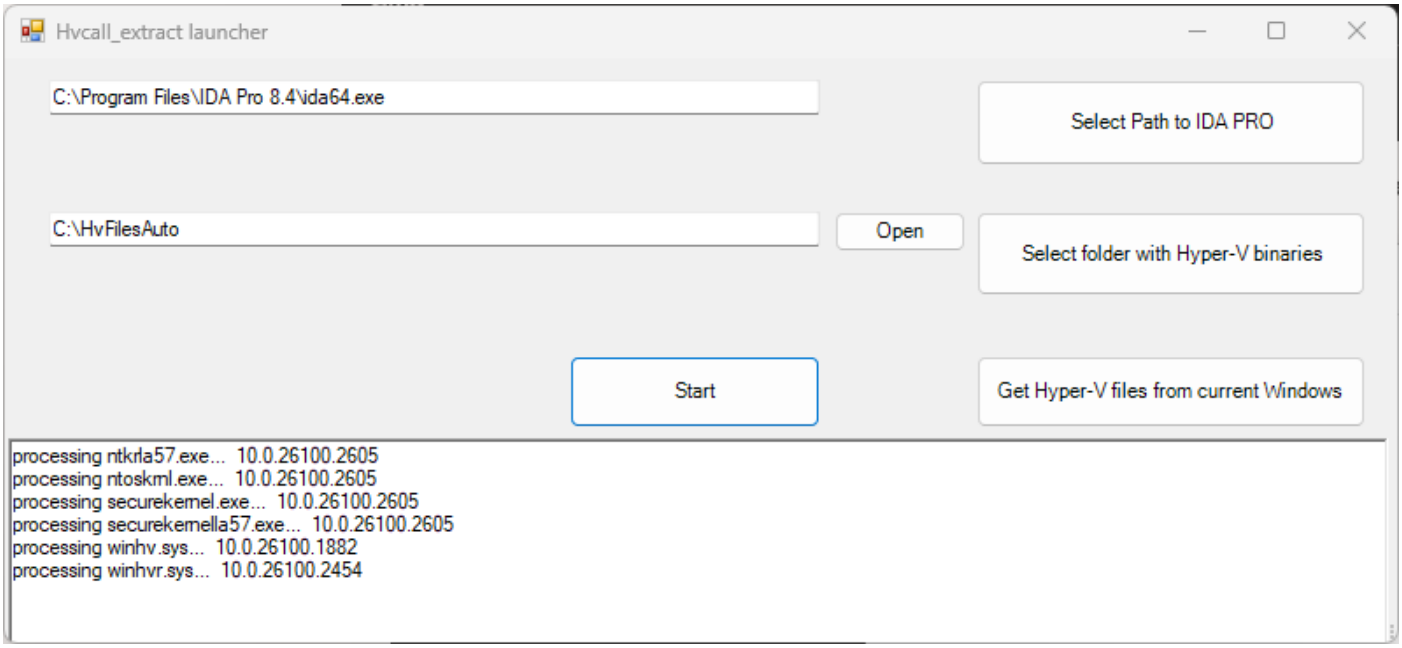
Just in case readers want to analyze a PDB file, you can use **resym** (https://github.com/ergrelet/resym), which is an excellent tool for browsing and extracting types from PDB files, and runs on Windows, macOS and Linux, for visualizing any of these downloaded PDF files. For example, examining the securekernel.exe's PDB, we have:



[Figure 34]: Browsing PDB files

The challenge is to retrieving symbols for **hvix64.exe** and **hvloader.dll** files. One of the first resources is the tool from **Arthur "Gerhart" Khudyaev**, which provides us with an automated way to extract hypercalls given a selected set of files. The steps that I usually follow are:

- git clone https://github.com/gerhart01/Hyper-V-scripts
- create a folder such as **C:\HvFilesAuto** (the folder where I saved all my original copied files and respective IDA databases is **C:\HvFiles**).
- copy **ntoskrnl.exe, securekernel.exe, winhvr.sys, winhv.sys, securekernella57.exe** and **ntkrla57.exe** to **C:\HvFilesAuto**.
- copy the entire **extract_hvcalls_gui folder** from project to **C:\HvFilesAuto**.
- get into the **extract_hvcalls_gui** folder, right-click on **hvcall_extract.Export.ps1** PowerShell script and choose "run with PowerShell".
- Select the appropriate IDA Pro path.
- Select the folder with Hyper-V files (**C:\HvFilesAuto**).
- Press Start



[Figure 35]: Extracting hypercalls

If you provided all necessary files, the tool should run smoothly, and initially you will find a series of files (extensions id0, id1, id2, nam and til), which represent an extracted IDA idb file, for each processed Hyper-V related file. Please, wait until all IDA databases have been closed and, at the end of the process, you will see all respective .idb files. Within **extract_hvcalls_gui folder**, you will see a series of Json files, but likely there will be some duplicated values, which is expected. Open a command prompt in the same folder, run the following command:

- **python hvcalls_merge.py**

Once the script is executed, the result folder will be created and, inside this folder, a file named **hvcalls_results.json** is created. The script being executed and the generated file is shown below:

```
C:\HvFilesAuto\extract_hvcalls_gui>python hvcalls_merge.py
processing ntkrla57.exe_10.0.26100.2605.json... hvcall count:60
count of duplicated hypercalls (with parameters):  0
processing ntoskrnl.exe_10.0.26100.2605.json... hvcall count:58
count of duplicated hypercalls (with parameters):  0
processing securekernel.exe_10.0.26100.2605.json... hvcall count:23
count of duplicated hypercalls (with parameters):  0
processing securekernella57.exe_10.0.26100.2605.json... hvcall count:23
count of duplicated hypercalls (with parameters):  0
processing winhv.sys_10.0.26100.1882.json... hvcall count:3
count of duplicated hypercalls (with parameters):  0
empty dictionary in C:\HvFilesAuto\extract_hvcalls_gui\hvcalls_json_files\winhvr.sys_10.0.26100.2454.jso
n
Saving file to  C:\HvFilesAuto\extract_hvcalls_gui\result\hvcalls_results_with_duplicates.json
processing ntkrla57.exe_10.0.26100.2605.json... hvcall count:60
   duplicate in key found. key: 0x2 dict1: HvlpSlowFlushListTb_hardcoded_value dict2 HvlpSlowFlushListTb
_hardcoded_value
   duplicate in key found. key: 0x7 dict1: HvlpDynamicUpdateMicrocode_hardcoded_value dict2 HvlpDynamicU
pdateMicrocode_hardcoded_value
   duplicate in key found. key: 0x13 dict1: HvlpSlowFlushAddressSpaceTbEx_hardcoded_value dict2 HvlpSlow
FlushAddressSpaceTbEx_hardcoded_value
   duplicate in key found. key: 0x14 dict1: HvlpSlowFlushListTbEx_hardcoded_value dict2 HvlpSlowFlushLis
tTbEx_hardcoded_value
   duplicate in key found. key: 0x15 dict1: HvlpSlowSendSyntheticClusterIpiEx_hardcoded_value dict2 Hvlp
SlowSendSyntheticClusterIpiEx_hardcoded_value
   duplicate in key found. key: 0x48 dict1: HvlMapGpaPages_hardcoded_value dict2 HvlMapGpaPages_hardcode
d_value
```

[Figure 36]: Merging hypercalls (truncated output)

C: > HvFilesAuto > extract_hvcalls_gui > result > {} hvcalls_results.json > ...

```
{
    "0x2": "HvCallFlushEntireTb",
    "0x3": "HvCallNotifyPageHeat_hardcoded_value",
    "0x7": "HvCallDynamicUpdateMicrocode_hardcoded_value",
    "0xb": "HvCallSendSyntheticClusterIpiEx",
    "0xc": "HvCallpProtectPages",
    "0xd": "HvCallpEnablePartitionVtl",
    "0x13": "HvCallSlowFlushAddressSpaceTbEx_hardcoded_value",
    "0x14": "HvCallSlowFlushListTbEx_hardcoded_value",
    "0x15": "HvCallSlowSendSyntheticClusterIpiEx_hardcoded_value",
    "0x18": "HvCallLoadHypervisorPatch",
    "0x1b": "HvCallWritePerfRegister",
    "0x1c": "HvCallReadPerfRegister",
    "0x44": "HvCallGetPartitionProperty",
    "0x45": "HvCallSetPartitionProperty",
    "0x48": "HvCallDepositPages_hardcoded_value",
    "0x4d": "HvCallInstallExceptionIntercept",
    "0x4e": "HvCallCreateRootVirtualProcessor_hardcoded_value",
    "0x50": "HvCallGetVpRegisters",
    "0x51": "HvCallSetVpRegisters",
    "0x52": "HvCallTranslateVirtualAddress",
    "0x6e": "HvCallMapSparseGpaPages_hardcoded_value",
    "0x6f": "HvCallUnblockDefaultDma",
    "0x7c": "HvCallMapDeviceInterrupt_hardcoded_value",
    "0x7f": "HvCallRetargetDeviceInterrupt_hardcoded_value",
    "0x82": "HvCallRegisterDeviceId_hardcoded_value",
    "0x83": "HvCallUnregisterDeviceId",
    "0x88": "HvCallLpReadMultipleMsr_hardcoded_value",
    "0x89": "HvCallLpWriteMultipleMsr_hardcoded_value",
    "0x8e": "SkLiveDumpCollect",
```

```
    "0x94": "HvCallInjectVtl0Nmi",
    "0x9d": "HvCallCreatePasidSpace",
    "0x9f": "HvCallSetPasidAddressSpace",
    "0xa0": "HvCallFlushPasidAddressSpace",
    "0xa1": "HvCallSlowFlushPasidAddressList_hardcoded_value",
    "0xa2": "HvCallAttachPasidSpace",
    "0xa3": "HvCallDetachPasidSpace",
    "0xa4": "HvCallEnablePasid",
    "0xa5": "HvCallDisablePasid",
    "0xa6": "HvCallSlowAcknowledgePageRequest_hardcoded_value",
    "0xa7": "HvCallCreatePrQueue",
    "0xa8": "HvCallDeletePrQueue",
    "0xa9": "HvCallClearPrqStalled",
    "0xab": "HvCallSetDeviceDmaEnabled",
    "0xae": "HvCallSetGpaPageAttributes",
    "0xb1": "IumGetDmaEnabler",
    "0xb2": "HvCallAttachDeviceDomain",
    "0xb3": "HvCallMapDeviceGpaPages",
    "0xb4": "HvCallUnmapDeviceGpaPages",
    "0xbc": "HvCallAddRemovePhysicalMemory_hardcoded_value",
    "0xc3": "HvCallProcessIommuPrq",
    "0xc4": "HvCallDmaDetachDevice",
    "0xc5": "HvCallpDmaDeleteEnabler",
    "0xc7": "HvCallDmaMapDeviceSparsePages_hardcoded_value",
    "0xc8": "HvCallDmaUnmapDeviceSparsePages_hardcoded_value",
    "0xca": "HvCallGetSparseGpaPagesAccessState_hardcoded_value",
    "0xce": "HvCallDmaConfigureDeviceDomain",
    "0xd0": "HvCallDmaFlushDeviceDomain",
    "0xd1": "HvCallDmaFastFlushDeviceDomainVaList",
    "0xd8": "HvCallModifySparseSpaPageHostAccess",
    "0xdb": "HvCallModifySparseGpaPageHostVisibility",
    "0x100": "HvCallConfigureSecureAtsDevice",
    "0x103": "HvCallRestoreTime",
    "0x10b": "HvCallAttachPrQueue",
    "0x10c": "HvCallDetachPrQueue",
    "0x10d": "HvCallDmaReserveDeviceDomainAttachment",
    "0x10e": "HvCallDmaUnreserveDeviceDomainAttachment"
}
```

[Figure 37]: List of extracted hypercalls

Values presented above are not the only result and whether you check the **hvcalls_json_files folder** you will find extracted hypercalls for each input file, as shown below:

- ntkrla57.exe_10.0.26100.2605.json
- ntoskrnl.exe_10.0.26100.2510.json
- securekernel.exe_10.0.26100.2494.json
- securekernella57.exe_10.0.26100.2605.json
- unknown (folder)
- winhv.sys_10.0.26100.1930.json
- winhvr.sys_10.0.26100.2416.json

Of course, such files present different contexts and functions, and there may be duplicated values or not. For a quick overview of each file, execute:

```
C:\HvFilesAuto2\extract_hvcalls_gui\hvcalls_json_files>C:\cygwin64\bin\head.exe -n5 *.json
==> ntkrla57.exe_10.0.26100.2605.json <==
{
    "0x2": "HvlpSlowFlushListTb_hardcoded_value",
    "0x7": "HvlpDynamicUpdateMicrocode_hardcoded_value",
    "0x13": "HvlpSlowFlushAddressSpaceTbEx_hardcoded_value",
    "0x14": "HvlpSlowFlushListTbEx_hardcoded_value",

==> ntoskrnl.exe_10.0.26100.2510.json <==
{
    "0x7": "HvlpDynamicUpdateMicrocode_hardcoded_value",
    "0x13": "HvlpSlowFlushAddressSpaceTbEx_hardcoded_value",
    "0x14": "HvlpSlowFlushListTbEx_hardcoded_value",
    "0x15": "HvlpSlowSendSyntheticClusterIpiEx_hardcoded_value",

==> securekernel.exe_10.0.26100.2494.json <==
{
    "0x2": "ShvlFlushEntireTb",
    "0xb": "ShvlSendSyntheticClusterIpiEx",
    "0xc": "SkmiProtectPlaceholderPages",
    "0xd": "ShvlpEnablePartitionVtl",

==> securekernella57.exe_10.0.26100.2605.json <==
{
    "0x2": "ShvlFlushEntireTb",
    "0xb": "ShvlSendSyntheticClusterIpiEx",
    "0xc": "ShvlpProtectPages",
    "0xd": "ShvlpEnablePartitionVtl",

==> winhv.sys_10.0.26100.1930.json <==
{
    "0x50": "WinHvGetVpRegisters",
    "0x51": "WinHvSetVpRegisters",
    "0xdb": "WinHvModifySparseGpaPageHostVisibility"
}
==> winhvr.sys_10.0.26100.2416.json <==
{}
```

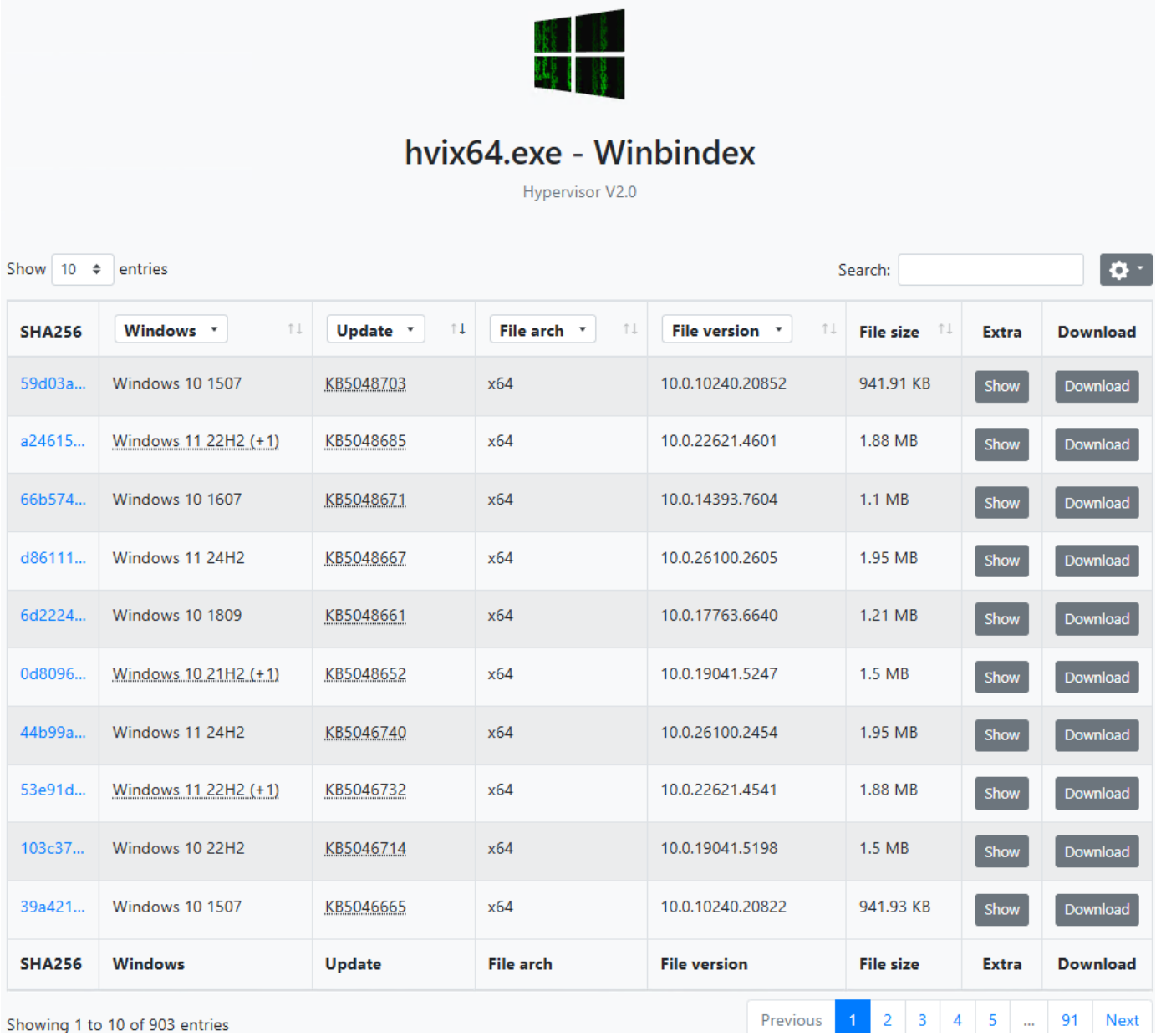
[Figure 38]: Inspecting individual results

Although this is a really introductory article, it is interesting to follow these hypercalls and notice that they might have changed over time. Although we have picked all files from a live system, an alternative (and sometimes better) would have been retrieving them from **Winbindex** (https://winbindex.m417z.com/), which can be very useful while getting current and past versions of such binaries. Personally, and that is only an opinion, I classify **Winbindex** as one of the most useful projects for vulnerability research on Windows because it allows us to make multiple experiments using multiple versions of a Windows binary.

Of course, not having symbols causes a huge issue while trying to analyze the Hyper-V behavior, and mainly whether we remember that the three most important files for an initial analysis are the following ones, loaded in the presented order:

- **winload.dll:** it is effectively the Windows loader | there are symbols!
- **hvloader.dll:** it is the Hyper-V loader | there are no symbols
- **hvix64.exe:** if the hypervisor for Intel systems: there are no symbols (hvax64.exe is the equivalent for AMD).

The next step is trying to discover whether there are functions in these binaries, which we do not have symbols, which can be found in other files. No doubt, the best approach to accomplishing this task is using binary diffing. This task can be performed using **BinDiff** (https://zynamics.com/software.html), which provides us with an excellent tool and plugin for IDA Pro. Therefore, we can use BinDiff to compare a file such as hvix64.exe against any other file that we collected and use any eventual matching to mark up our target code. If readers need a specific version of hvix64.exe (or any file used as comparison), **Winbindex** is a desirable alternative to get it:



hvix64.exe - Winbindex

Hypervisor V2.0

Show 10 entries Search:

SHA256	Windows	Update	File arch	File version	File size	Extra	Download
59d03a...	Windows 10 1507	KB5048703	x64	10.0.10240.20852	941.91 KB	Show	Download
a24615...	Windows 11 22H2 (+1)	KB5048685	x64	10.0.22621.4601	1.88 MB	Show	Download
66b574...	Windows 10 1607	KB5048671	x64	10.0.14393.7604	1.1 MB	Show	Download
d86111...	Windows 11 24H2	KB5048667	x64	10.0.26100.2605	1.95 MB	Show	Download
6d2224...	Windows 10 1809	KB5048661	x64	10.0.17763.6640	1.21 MB	Show	Download
0d8096...	Windows 10 21H2 (+1)	KB5048652	x64	10.0.19041.5247	1.5 MB	Show	Download
44b99a...	Windows 11 24H2	KB5046740	x64	10.0.26100.2454	1.95 MB	Show	Download
53e91d...	Windows 11 22H2 (+1)	KB5046732	x64	10.0.22621.4541	1.88 MB	Show	Download
103c37...	Windows 10 22H2	KB5046714	x64	10.0.19041.5198	1.5 MB	Show	Download
39a421...	Windows 10 1507	KB5046665	x64	10.0.10240.20822	941.93 KB	Show	Download
SHA256	Windows	Update	File arch	File version	File size	Extra	Download

Showing 1 to 10 of 903 entries Previous 1 2 3 4 5 ... 91 Next

[Figure 39]: Winbindex | hvix64.exe

As a first example, open the IDA database (**.idb** extension) of the **hvix64.exe** (previously analyzed), go to **Edit → Plugins → BinDiff**, click on **Diff Database** and choose the IDA Database from any other analyzed file as, for example, **winload.efi**. You will notice that there is a lengthy list of matched routines, as shown below:

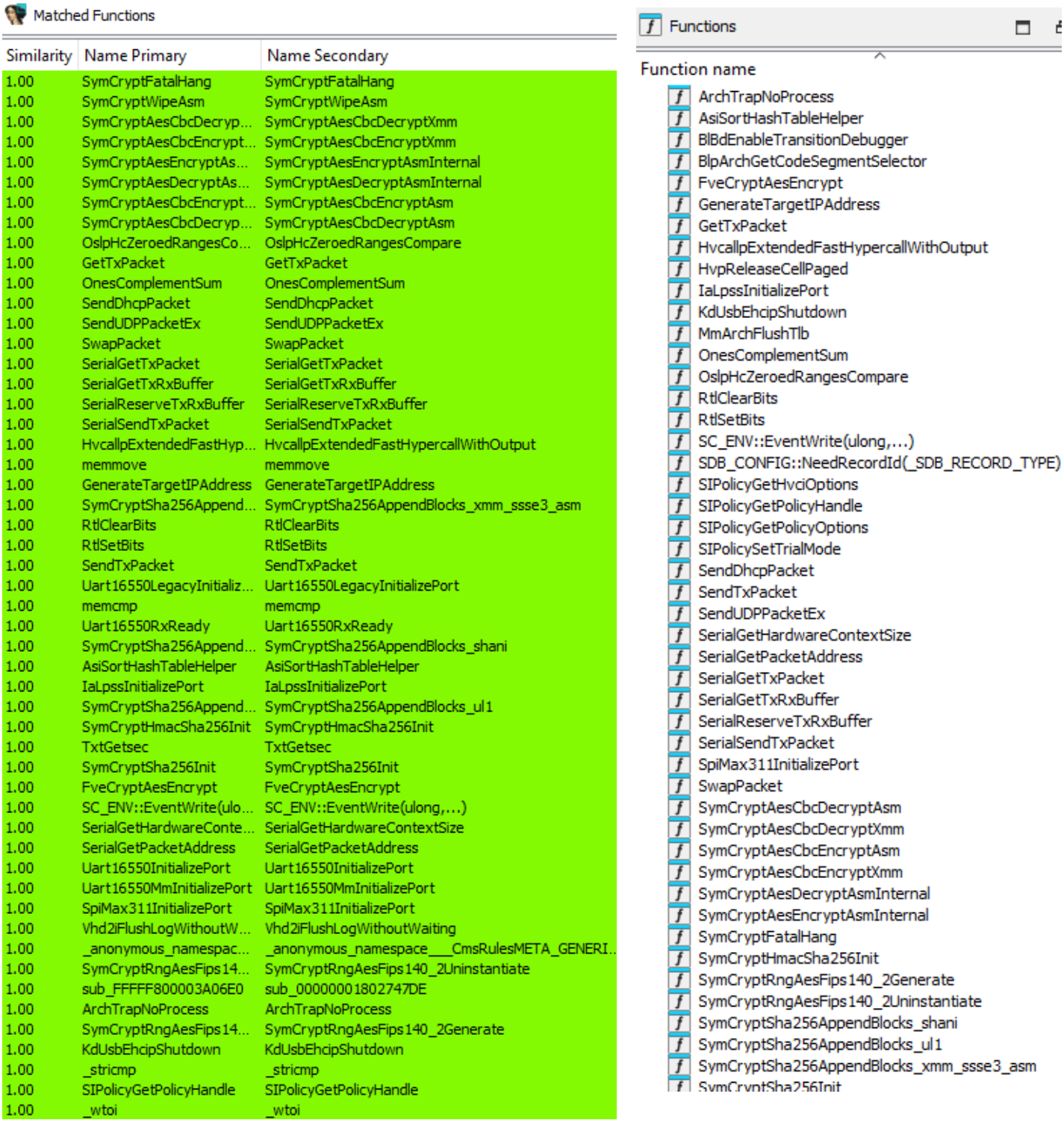
Matched Functions

Similarity	Confic	Change	EA Primary	Name Primary	EA Secondary	Name Secondary
1.00	0.99	-------	FFFFF8000022D838	sub_FFFFF8000022D838	000000018020D6...	SymCryptFatalHang
1.00	0.99	-------	FFFFF8000022D8D0	sub_FFFFF8000022D8D0	00000001802730...	SymCryptWipeAsm
1.00	0.99	-------	FFFFF8000022FB70	sub_FFFFF8000022FB70	000000018020FA...	SymCryptAesCbcDecryptXmm
1.00	0.99	-------	FFFFF800002300B8	sub_FFFFF800002300B8	000000018020F9...	SymCryptAesCbcEncryptXmm
1.00	0.99	-------	FFFFF80000230250	sub_FFFFF80000230250	00000001802731...	SymCryptAesEncryptAsmInternal
1.00	0.99	-------	FFFFF80000230430	sub_FFFFF80000230430	00000001802733...	SymCryptAesDecryptAsmInternal
1.00	0.99	-------	FFFFF80000230600	sub_FFFFF80000230600	00000001802736...	SymCryptAesCbcEncryptAsm
1.00	0.99	-------	FFFFF800002306D0	sub_FFFFF800002306D0	00000001802736...	SymCryptAesCbcDecryptAsm
1.00	0.99	-------	FFFFF80000267FF0	sub_FFFFF80000267FF0	000000018002B7...	OslpHcZeroedRangesCompare
1.00	0.99	------C	FFFFF8000031DA38	sub_FFFFF8000031DA38	00000001801DB4...	GetTxPacket
1.00	0.99	-------	FFFFF8000031EE00	sub_FFFFF8000031EE00	00000001801DB0...	OnesComplementSum
1.00	0.99	-------	FFFFF8000031F4BC	sub_FFFFF8000031F4BC	00000001801DE7...	SendDhcpPacket
1.00	0.99	-------	FFFFF80000320250	sub_FFFFF80000320250	00000001801DC...	SendUDPPacketEx
1.00	0.99	-------	FFFFF80000320310	sub_FFFFF80000320310	00000001801DB0...	SwapPacket
1.00	0.99	-------	FFFFF800003218A0	sub_FFFFF800003218A0	00000001801E04...	SerialGetTxPacket
1.00	0.99	-------	FFFFF80000321928	sub_FFFFF80000321928	00000001801DFF...	SerialGetTxRxBuffer
1.00	0.99	-------	FFFFF80000321B00	sub_FFFFF80000321B00	00000001801DFF...	SerialReserveTxRxBuffer
1.00	0.99	-------	FFFFF80000321BA0	sub_FFFFF80000321BA0	00000001801E04...	SerialSendTxPacket
1.00	0.99	-------	FFFFF800003A4560	sub_FFFFF800003A4560	0000000180275B...	HvcallpExtendedFastHypercallWithOutput
1.00	0.99	-------	FFFFF800003B1380	sub_FFFFF800003B1380	00000001802766...	memmove
1.00	0.99	-------	FFFFF8000031CB20	sub_FFFFF8000031CB20	00000001801DE5...	GenerateTargetIPAddress
1.00	0.99	-------	FFFFF800002308B0	sub_FFFFF800002308B0	0000000180270D...	SymCryptSha256AppendBlocks_xmm_ssse3_asm
1.00	0.99	-------	FFFFF80000242890	sub_FFFFF80000242890	00000001800501...	RtlClearBits
1.00	0.99	-------	FFFFF80000242C1C	sub_FFFFF80000242C1C	00000001800502...	RtlSetBits
1.00	0.99	------C	FFFFF800003201D0	sub_FFFFF800003201D0	00000001801DB4...	SendTxPacket
1.00	0.99	-------	FFFFF80000321E00	sub_FFFFF80000321E00	00000001801E09...	Uart16550LegacyInitializePort
1.00	0.99	-------	FFFFF800003B1750	sub_FFFFF800003B1750	00000001802765...	memcmp
1.00	0.99	------C	FFFFF80000321FD0	sub_FFFFF80000321FD0	00000001801E0C...	Uart16550RxReady
1.00	0.99	-------	FFFFF8000022DE60	sub_FFFFF8000022DE60	0000000180208A...	SymCryptSha256AppendBlocks_shani
1.00	0.99	-------	FFFFF8000023ADF0	sub_FFFFF8000023ADF0	000000018024A4...	AsiSortHashTableHelper
1.00	0.99	-------	FFFFF80000322480	sub_FFFFF80000322480	00000001801E05...	IaLpssInitializePort
1.00	0.99	-------	FFFFF8000022E224	sub_FFFFF8000022E224	00000001802077...	SymCryptSha256AppendBlocks_ul1
1.00	0.99	-------	FFFFF8000022F7B8	sub_FFFFF8000022F7B8	0000000180204D...	SymCryptHmacSha256Init
1.00	0.99	-------	FFFFF800003A41D0	sub_FFFFF800003A41D0	0000000180274D...	TxtGetsec
1.00	0.98	-------	FFFFF8000022F44C	sub_FFFFF8000022F44C	00000001802075...	SymCryptSha256Init
1.00	0.98	------C	FFFFF80000253F00	sub_FFFFF80000253F00	00000001801D4E...	FveCryptAesEncrypt
1.00	0.98	-------	FFFFF80000258410	sub_FFFFF80000258410	0000000180073B...	SC_ENV::EventWrite(ulong,...)
1.00	0.98	------C	FFFFF80000321650	sub_FFFFF80000321650	00000001801E05...	SerialGetHardwareContextSize
1.00	0.98	-------	FFFFF80000321680	sub_FFFFF80000321680	00000001801E04...	SerialGetPacketAddress
1.00	0.98	-------	FFFFF80000321D20	sub_FFFFF80000321D20	00000001801E0A...	Uart16550InitializePort
1.00	0.98	-------	FFFFF80000321E70	sub_FFFFF80000321E70	00000001801E0A...	Uart16550MmInitializePort
1.00	0.98	-------	FFFFF80000322B10	sub_FFFFF80000322B10	00000001801E0E...	SpiMax311InitializePort
1.00	0.98	------C	FFFFF8000032974C	sub_FFFFF8000032974C	00000001801E9E...	Vhd2iFlushLogWithoutWaiting
1.00	0.97	-------	FFFFF8000029DBD0	sub_FFFFF8000029DBD0	00000001801084...	_anonymous_namespace___CmsRulesMETA_GENERI...
1.00	0.97	-------	FFFFF80000347DF0	sub_FFFFF80000347DF0	00000001802060...	SymCryptRngAesFips140_2Uninstantiate
1.00	0.97	-------	FFFFF800003A06E0	sub_FFFFF800003A06E0	00000001802747...	sub_00000001802747DE
1.00	0.95	-------	FFFFF800003ADF80	nullsub_7	00000001802748...	ArchTrapNoProcess
1.00	0.82	------C	FFFFF80000225A60	sub_FFFFF80000225A60	00000001802060...	SymCryptRngAesFips140_2Generate
1.00	0.82	-------	FFFFF8000023B4B0	sub_FFFFF8000023B4B0	0000000180263C...	KdUsbEhcipShutdown
1.00	0.82	------C	FFFFF80000254280	sub_FFFFF80000254280	000000018026B5...	_stricmp
1.00	0.82	-------	FFFFF80000258430	sub_FFFFF80000258430	000000018021C0...	SIPolicyGetPolicyHandle
1.00	0.82	-------	FFFFF8000029FF7C	sub_FFFFF8000029FF7C	000000018026C6...	_wtoi

[Figure 40]: BinDiff | hvix64.exe and winload.efi

There have been 68 function full-matches (100% similarity) between both binaries. Is it too much? It is not and also is not a perfect method (there are issues, obviously), but certainly is much better than nothing. Actually, every time you can retrieve information such as function name, enumeration type, structure definition and others are always welcome because it will make our analysis easier. When we have a function, we can try to determine its prototype and, from this point, we can build a better marked up code.

Once we select all functions with similarity equal to 1.0, we can right-click them and pick up **Import Symbols/Comments**. You will notice that all imported symbols (names) will appear in **Function View** and also in the **Name Primary column** in **Matched Functions**:



[Figure 41]: BinDiff | hvix64.exe and winload.efi -- imported symbols

Do not forget to **save** both the **difference database** (choose **Save Results** in the BinDiff window) and save the hvix64's database (**CTRL+W**).

We can repeat this same procedure considering other binaries because every single identical function that we can find is always helpful for the analysis, and it is appropriate to remember that hvix64.exe has over 4500 routines (in my case, 4659 routines). Therefore, I will be showing only a summary of a binary diffing between hvix64.exe and the following binaries and including only a perfect match (similarity equal to 1.0), but readers must be aware that every month there are updates on Windows and, obviously, versions presented below are not the newest ones. Furthermore, there will be a series of overlapping functions (actually, most of them) during the binary diffing process, so reader need to check functions with missing identification, check whether they make sense, and import only these chosen ones. For example, readers might pay attention to matched functions that have been discovered using BinDiff algorithm such as Prime Signature, Address Sequence, MD Index, and other ones. In these cases, if the Name Primary is something like sub_<address> and the Name Secondary offers a name, take it. Certainly, readers should also evaluate whether it is valid to import symbols of functions whose similarity is not 1.0. According to my past experience, taking functions with similarity 0.99 or even 0.98 is not a complete bad idea because changes are minimal and, in many opportunities, they could help a lot because we do not have any available symbol. Therefore, the results below show the total of matched functions and, so make the process less convoluted, I have included matches detected by the Name Hash algorithm:

ntoskrnl.exe	90 functions
securekernel.exe	48 functions
winhvr.sys	15 functions
winhv.sys	10 functions
securekernella57.exe	42 functions
ntkrla57.exe	102 functions
kdnet.exe	07 functions
hvhostsvc.dll	12 functions
hvsocket.dll	6 functions
hvservice.sys	9 functions
hvsocketcontrol.sys	4 functions
storvsc.sys	8 functions
storvsp.sys	7 functions
vid.dll	2 functions
vid.sys	28 functions
vmbkmcl.sys	16 functions
vmbkmclr.sys	17 functions
vmbuspipe.dll	6 functions
vmbuspiper.dll	6 functions
vmbusr.sys	20 functions
vmbusvdev.dll	15 functions
vmdebug.dll	12 functions
vmemulatedstorage.dll	23 functions
vmss.exe	34 functions
vmwp.exe	41 functions
vmwpctrl.dll	12 functions
vpcivsp.sys	13 functions
winhvemulation.dll	14 functions
winhvplatform.dll	23 functions
winload.efi	68 functions

[Figure 42]: BinDiff | hvix64.exe and some files from Windows system

This simplified approach retrieved symbols from 161 functions. Using **IDA Lumina** (**F12 - Push All**), we have 191 functions, where some of functions identified by Lumina follow below:

Function name	Segment	Start	Length
_tlgCreate1Sz_char	.text	FFFFF8000020009C	0000002D
HalpTimerGetInternalData	.text	FFFFF8000021E3F0	00000023
sgx_oc_cpuidex	.text	FFFFF8000022D5F8	00000024
SymCryptWipeAsm	**.text**	**FFFFF8000022D8D0**	**000000B1**
SymCryptHmacSha256Init	.text	FFFFF8000022F7B8	00000022
SymCryptHmacSha256Result	.text	FFFFF8000022F7E4	0000009A
SymCryptAesEncryptAsmInternal	.text	FFFFF80000230250	000001D1
SymCryptAesDecryptAsmInternal	.text	FFFFF80000230430	000001C1
SymCryptAesCbcEncryptAsm	.text	FFFFF80000230600	000000BF
SymCryptAesCbcDecryptAsm	.text	FFFFF800002306D0	00000117
SymCryptSha256AppendBlocks_xmm_ssse3_asm	.text	FFFFF800002308B0	0000229D
KdpSuspendAllBreakpoints	.text	FFFFF80000250C0C	00000026
EtwpPsProvTracePriority	.text	FFFFF8000025C6E0	00000042
ShvlpAcquireHypercallPage	.text	FFFFF80000261BB0	00000020
SDL_UninitializedVideo	.text	FFFFF800002764C0	00000021
HvAcquireSparseGpaPageHostAccess	.text	FFFFF8000028CB10	00000023
HvAcquireSparseGpaPageHostAccess_0	.text	FFFFF8000028CB40	00000023
HvReleaseSparseGpaPageHostAccess	.text	FFFFF8000028EF50	00000023
HvReleaseSparseGpaPageHostAccess_0	.text	FFFFF8000028EF80	00000023
HvCallSetEventLogGroupSources	.text	FFFFF80000297610	000000B9
ObInsertObject	**.text**	**FFFFF800002D8E14**	**00000027**
HviGetHypervisorFeatures	.text	FFFFF8000031C5F4	00000039
HviIsAnyHypervisorPresent	.text	FFFFF8000031C674	00000055
OnesComplementSum	.text	FFFFF8000031EE00	0000004B
SerialGetPacketAddress	.text	FFFFF80000321680	00000022
SerialGetPacketLength	.text	FFFFF800003216B0	00000027
SerialGetTxPacket	.text	FFFFF800003218A0	0000007F
SerialReleaseRxPacket	.text	FFFFF80000321AD0	00000029
SerialSendTxPacket	.text	FFFFF80000321BA0	00000043
Uart16550GetByte	.text	FFFFF80000321C30	000000E2
Uart16550InitializePortCommon	.text	FFFFF80000321D44	000000B3
Uart16550RxReady	.text	FFFFF80000321FD0	0000002D
Uart16550SetBaudCommon	.text	FFFFF80000322030	0000009F
ReadPortWithIndex16	.text	FFFFF800003220E0	00000025
ReadPortWithIndex16_0	.text	FFFFF80000322110	00000025
ReadPortWithIndex16_1	.text	FFFFF80000322140	00000025
ReadPortWithIndex16_2	.text	FFFFF80000322170	00000025
ReadPortWithIndex16_3	.text	FFFFF800003221A0	00000025
ReadPortWithIndex16_4	.text	FFFFF800003221D0	00000025
ReadPortWithIndex16_5	.text	FFFFF80000322200	00000025
UartpSetAccess	.text	FFFFF8000032222C	000000F4
WritePortWithIndex16	.text	FFFFF80000322330	00000029
WritePortWithIndex16_0	.text	FFFFF80000322360	00000029
WritePortWithIndex8	.text	FFFFF80000322390	00000028
WritePortWithIndex16_1	.text	FFFFF800003223C0	00000029

[Figure 43]: IDA Pro | subroutines identified by Lumina

Of course, there are multiple observations that should be made here:

- To provide readers with an example of trying to identify the functions, I have picked up quite a few binaries from various locations, but there are other files that could be added to the list, including potential binaries installed by Windows SDK / WDK.

- There is an obvious issue with the function's name to be chosen because, depending on the binary, a function is named reflecting the functionality and the own binary. In this case, readers will need to check the names and take a decision. Anyway, the most important is not the function's name itself, but the symbols imported by it.

- This simplified method can be used as a first and useful technique because, as readers remember, we did not have anything about **hvix64.exe** file. Therefore, every help is welcome, and makes possible to identify functions such this one below:

```
.text:FFFFF8000031C634 ; char HviGetHypervisorVendorAndMaxFunction()
.text:FFFFF8000031C634 HviGetHypervisorVendorAndMaxFunction proc near
.text:FFFFF8000031C634                                         ; CODE XREF: sub_FFFFF8000031A198+1C5↑p
.text:FFFFF8000031C634                                         ; HviGetHardwareFeatures+25↑p
.text:FFFFF8000031C634                 push    rbx
.text:FFFFF8000031C636                 sub     rsp, 20h
.text:FFFFF8000031C63A                 mov     r10, rcx
.text:FFFFF8000031C63D                 call    HviIsAnyHypervisorPresent
.text:FFFFF8000031C642                 xor     ecx, ecx
.text:FFFFF8000031C644                 test    al, al
.text:FFFFF8000031C646                 jz      short loc_FFFFF8000031C660
.text:FFFFF8000031C648                 mov     eax, 40000000h
.text:FFFFF8000031C64D                 cpuid
.text:FFFFF8000031C64F                 mov     [r10], eax
.text:FFFFF8000031C652                 mov     [r10+4], ebx
.text:FFFFF8000031C656                 mov     [r10+8], ecx
.text:FFFFF8000031C65A                 mov     [r10+0Ch], edx
.text:FFFFF8000031C65E                 jmp     short loc_FFFFF8000031C667
.text:FFFFF8000031C660 ; ---------------------------------------------------------------------------
.text:FFFFF8000031C660
.text:FFFFF8000031C660 loc_FFFFF8000031C660:                   ; CODE XREF: HviGetHypervisorVendorAndMa
.text:FFFFF8000031C660                 mov     [r10], rcx
.text:FFFFF8000031C663                 mov     [r10+8], rcx
.text:FFFFF8000031C667
.text:FFFFF8000031C667 loc_FFFFF8000031C667:                   ; CODE XREF: HviGetHypervisorVendorAndMa
.text:FFFFF8000031C667                 add     rsp, 20h
.text:FFFFF8000031C66B                 pop     rbx
.text:FFFFF8000031C66C                 retn
.text:FFFFF8000031C66C HviGetHypervisorVendorAndMaxFunction endp
```

[Figure 44]: IDA Pro | an example of identified routine

- Although the number of matches can be reasonable at the first view, there is a substantial number of matches overlapping.

- **BinDiff** uses the following algorithms (https://www.zynamics.com/bindiff/manual/) to identify function matches:

  - **hash matching** match quality: very good
  - **name hash matching** match quality: very good
  - **edges flowgraph MD index** match quality: very good
  - **edges callgraph MD index** match quality: good
  - **MD index matching** match quality: good
  - **prime signature matching** match quality: good
  - **edges proximity MD index** match quality: medium
  - **relaxed MD index matching** match quality: medium
  - **address sequence** match quality: poor
  - **string references** match quality: medium
  - **loop count matching** match quality: poor
  - **call sequence matching** match quality: very poor

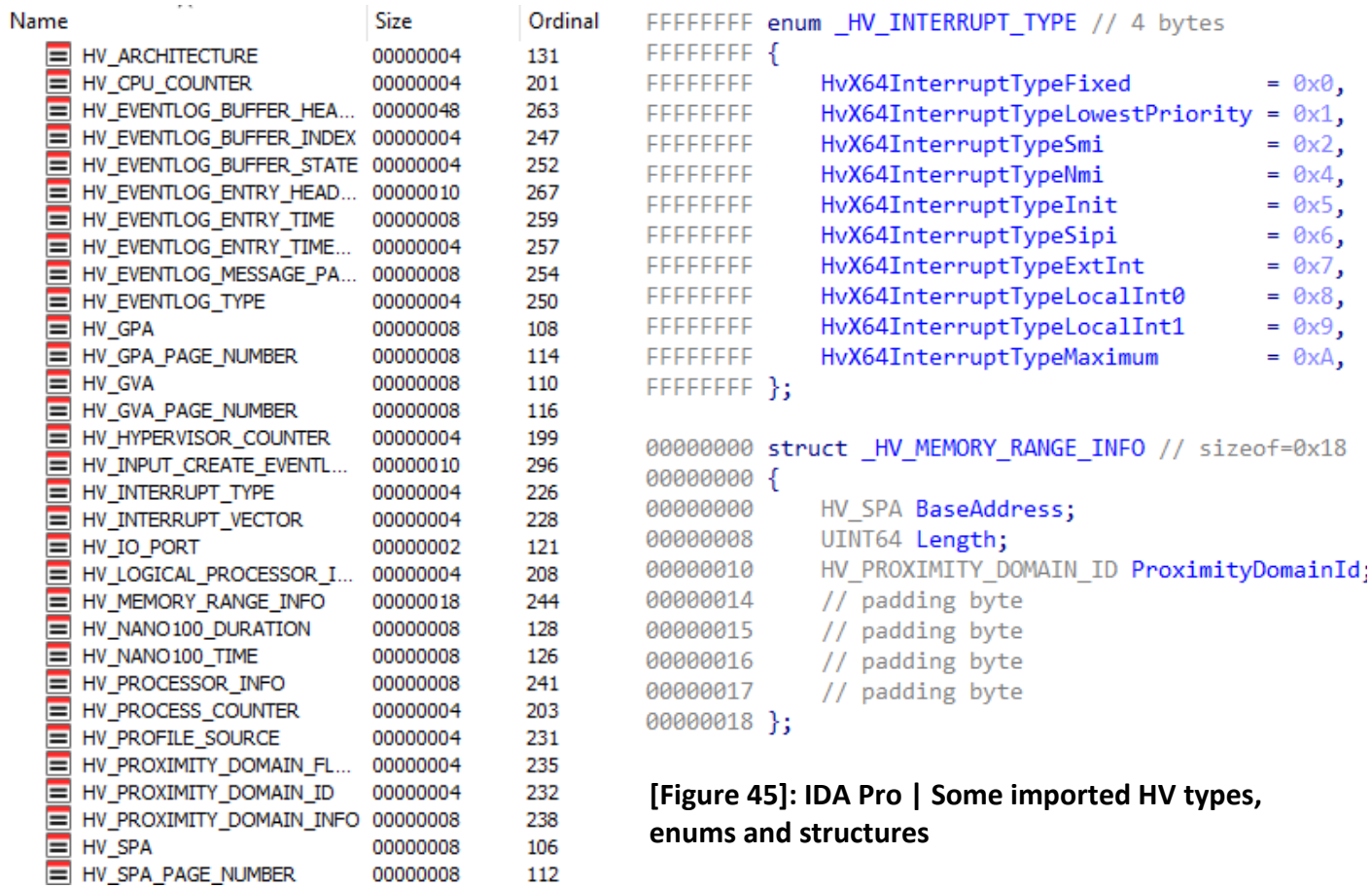
- Based on the quality of matches, readers can use this metric as a guideline to learn about reliability of potential function matches.

The next step to improve the code representation is looking for header files related to Hyper-V because they include a series of types and symbols that might be useful:

- **hvgdk.h:** https://github.com/ionescu007/hdk/blob/master/hvgdk.h
- **vmx.h:** https://github.com/ionescu007/SimpleVisor/blob/master/vmx.h

There are other Hyper-V related header files such as winhvplatform.h, winhvplatformdefs.h, winhvemulation.h and hvsocket.h (all of them from WDK location - C:\Program Files (x86)\Windows Kits\10\Include), but they are really useful and also require other header files.

To import these header files into IDA, go to **File → LoadFile → Parse C header file** (**CTRL+F9**). Eventually, you could need to do adjustments on the used compiler (**Options → Compiler**) or even to include definitions (maybe this Gist could be useful to guide you in eventual modifications: https://gist.githubusercontent.com/Dliv3/d011325312292182a9674797761d2b41/raw/4daa1f2dd3bc68b6dee39829e4f833dc0cd57440/defs.h. Once you have imported headers, you will have something similar to:



[Figure 45]: IDA Pro | Some imported HV types, enums and structures

Furthermore, there are other imported structures, constants, enums, but I have not shown all of them here because it is unnecessary. These header files will help you to get a better understanding of the code and mainly its logic. Of course, while analyzing Hyper-V even in a static form through IDA Pro, you must remember that order of loading: **winload.dll → hvloader.dll → hvix64.exe**. Do not forget this order.

From this point, and as also suggested by other researchers, the first steps is to prepare the hypercall structure and gather all possible information about them from public (https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/tlfs) and private sources, and without forgetting that most of these hypercalls are also directly and indirectly called from the ntoskrnl.exe.

10. Final observations

At this time, we are evaluating the next step, which is finding the potential attack interfaces, and some of the well-known are:

- **Hypercalls handlers / Hypervisor Interface (winhvr.sys)**
- **Control registers (MSRs)**
- **Intercept Handling**
- **Instruction Emulation**
- **VMBus**
- **I/O Port / MMU**
- **VMSwitch (vmswitch.sys)**
- **Para-virtualized storage (storvsp.sys)**
- **Virtualization Infrastructure Driver (vid.sys)**
- **Para-virtualized PCI (vpci.sys)**
- **VM Work Process (Pvmwp.exe)**

Readers can interact with distinguished structures such as content of MSR, content of VMCS, EPT entries and IO through the **hvext extension** maintained by **Satoshi Tanda** on https://github.com/tandasat/hvext. I avoid showing the output from the hvext_intel.js script because it is only a manual task, and it will be a valid topic to the next part when we bring more context to our study. I would like to proceed with the dynamic analysis on the loading of the hypervisor, but as I would have to repeat these steps in the next part and also my articles have been extensive, I decided to leave these steps to the appropriate article.

11. References

There is a brief list of references about Hyper-V and even other ones, which are always useful for getting further information and collecting different points of view.

- **Windows Internals book – 7th edition** (Andrea Allievi, Alex Ionescu, Mark E. Russinovich and David A. Solomon)
- **Intel® 64 and IA-32 Architectures Software Developer Manuals** (combined volume set): https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- **OpenHCL: the new, open source paravisor:** https://techcommunity.microsoft.com/blog/windowsosplatform/openhcl-the-new-open-source-paravisor/4273172
- **Confidential VMs on Azure:** https://techcommunity.microsoft.com/blog/windowsosplatform/confidential-vms-on-azure/3836282

- **Windows Hyper-V:** https://github.com/microsoft/openvmm/blob/main/Guide/src/user_guide/openhcl/run/hyperv.md
- **Windows OpenVMM:** https://github.com/microsoft/openvmm/blob/main/Guide/src/user_guide/openhcl/run/openvmm.md
- **OpenHCL Architecture:** https://github.com/microsoft/openvmm/blob/main/Guide/src/reference/architecture/openhcl.md
- **Hyper-V documentation:** https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/
- **Debugging the undebuggable – Part 1:** https://www.andrea-allievi.com/blog/debugging-the-undebuggable-part-1/
- **A Dive into Hyper-V Architecture and Vulnerabilities**: https://www.youtube.com/watch?v=2bK_rC81_Eo
- **Exploiting the Hyper-V IDE Emulator to Escape the Virtual Machine:** https://www.youtube.com/watch?v=50xxJEODO3M
- **Isolated User Mode (IUM) Processes:** https://learn.microsoft.com/en-us/windows/win32/procthread/isolated-user-mode--ium--processes
- **Fuzzing para-virtualized devices in Hyper-V:** https://msrc.microsoft.com/blog/2019/01/fuzzing-para-virtualized-devices-in-hyper-v/
- **First Steps in Hyper-V Research:** https://msrc.microsoft.com/blog/2018/12/first-steps-in-hyper-v-research/
- **Growing Hypervisor 0day with Hyperseed:** https://www.youtube.com/watch?v=Qms328deZ68
- **ACPICA project:** https://github.com/acpica/acpica
- **Hardening Hyper-V through offensive security research:** https://i.blackhat.com/us-18/Thu-August-9/us-18-Rabet-Hardening-Hyper-V-Through-Offensive-Security-Research.pdf
- **Microsoft Virtualization Documentation:** https://github.com/MicrosoftDocs/Virtualization-Documentation/tree/main
- **And now for something (not) completely different:** https://ourwindowsman.wordpress.com/page/2/
- **Hyper-V debugging for beginners. 2nd edition:** https://hvinternals.blogspot.com/2021/01/hyper-v-debugging-for-beginners-2nd.html

12. Conclusion

This article presented a basic and superficial introduction to Hyper-V research, which is a quite extensive topic, and there are much more to be presented in future articles. Stay tuned!

Just in case you want to stay connected:

- **Twitter: @ale_sp_brazil**
- **Blog:** https://exploitreversing.com

Keep learning, reversing, and exploiting everything, and I will see you next time!

Alexandre Borges