# Hyper-V Internals

**Tuesday, September 10, 2019**

## Hyper-V memory internals. Guest OS memory access

Software, used in article (operation systems have August 2019 patches):

Windows 10, build 1903 x64
Windows Server 2019
Windows Server 2016
WinDBG Preview
Visual Studio 2019
Process Hacker
PyKd plugin for WinDBG

Testing lab works on Intel-based PC. Therefore, Intel specific Hyper-V terms: hvix64.exe, vmcall instruction, etc will be used in article context.

**Terms and definitions:**

- **WDAG –** Windows Defender Application Guard;
- **Full VM (virtual machine)** – virtual server, which was created in Hyper-V manager. Differs from WDAG container, Windows Sandbox, docker in Hyper-V isolation mode;
- **Root OS** – operation system, where server part of Hyper-V is installed;
- **Guest OS** – operation system, which works in Hyper-V emulation context, uses virtual devices, which is presented by Hyper-V infrastructure. It can be Full VM and Hyper-V containers;

**Links**

- My twitter: @gerhart_x

**About Me**

Gerhart X

View my complete profile

**Blog Archive**

▼ 2019 (1)
▼ September (1)
Hyper-V memory internals. Guest OS memory access

► 2017 (7)
► 2015 (2)

- **TLFS** – Hypervisor Top-Level Functional Specification 5.0;
- **GPA (guest physical address)** – Guest OS physical memory address;
- **SPA (system physical address)** – Root OS physical memory address;
- **Hypercall** – hypervisor service function, which is called by vmcall execution with specifying hypercall number;
- **PFN** – page frame number.

Source of hvmm.sys driver on github.com:
https://github.com/gerhart01/LiveCloudKd/hvmm

Python-script for GPAR and MBlock objects parsing
https://github.com/gerhart01/Hyper-V-Internals/ParsePrtnStructure.py

**Intro**

Long time ago I didn't write anything in my blogpost. It doesn't mean, that I stopped Hyper-V research. Since Microsoft issued WDAG in Windows 10, build 1803, I started investigate it, but got much problems. First, it was impossible to attach to container, because it doesn't support it. WDAG is isolated environment, and bcdedit options for debugging can't be configured. More then, every configuration option is reset after rebooting. Sysinternals LiveKD supports Hyper-V attaching, but compatibility was broken in latest OS versions, more then, guest OS memory reading hypercall HvReadGpa, which is used by LiveKd, is not compatible with containers.

It was stalemate, but it turned out, that Matt Suiche (@msuiche) from ?omae Technologies shared LiveCloudKd source code for me (many thanks to him!). That program allows attach WinDBG to guest OS, using vid.dll API for reading guest OS memory. But next problem is vid.dll execution blocked by Microsoft: functions from vid.dll can be executed only from vmwp.exe process context, otherwise it will be blocked by vid.sys driver, which compared _EPROCESS object of function's usermode caller process with parent vmwp.exe _EPROCESS. Additionally, some of original LiveCloudKd techniques stopped working in Windows 10. I had to update it too.

Working on adaptation of LiveCloudKd can help me understand Hyper-V guest memory internals better. Soon Matt shared sources on github (https://github.com/comaeio/LiveCloudKd).

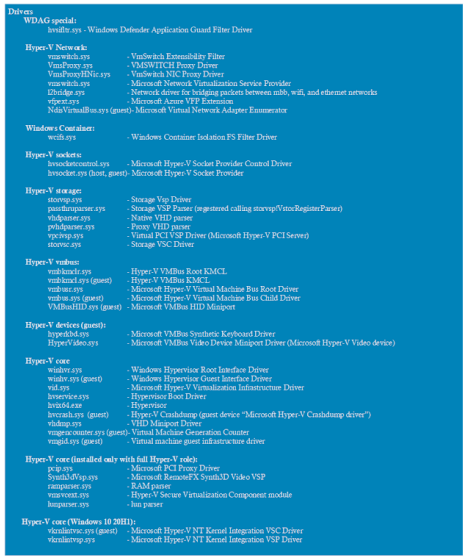
In 2017, Andrea Allievi made Hyper-V memory management architecture presentation (www.andrea-allievi.com/files/Recon_2017_Montreal_HyperV_public.pptx). Good work, but details were described quite abstractly, it was hard to match information from presentation to real vid.sys code. I believe it was because at the moment of presentation, Hyper-V symbols information has not yet been published.

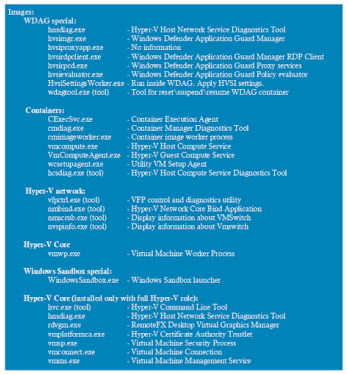
Btw, thanks to Andrea to pointing me to some names of vid.sys structures.

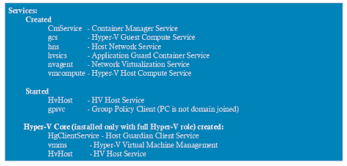
Additionally, need say thanks to Microsoft company, which decided to publish symbols for many Hyper-V modules (https://docs.microsoft.com/en-us/virtualization/community/team-blog/2018/20180425-hyper-v-symbols-for-debugging). Without them it was hard to analyze memory-managed vid functions.

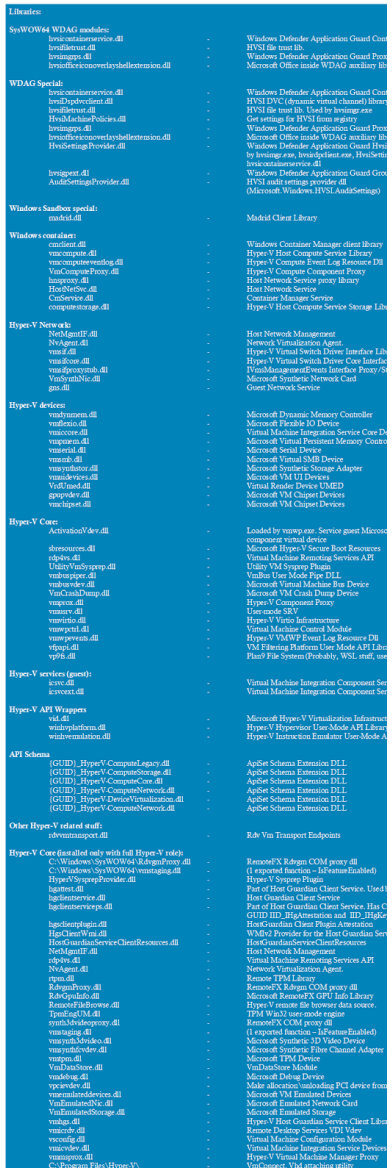
First, I planned wrote article about Hyper-V containers, but I made research log above 150 pages (6 from 9 font), but still don't understand whole working scheme. After that I decided to make a list of Hyper-V container components (then, it was extended to all Hyper-V

components cheat sheet – no much files were need to add. Containers and Hyper-V has very similar components base).




After that, I understood, that it has much components and too big for 1 article description. Therefore, I decided to highlighted more interesting things in separate article about guest OS memory structures.

Why guest OS only? Hyper-V kernel hvix64.exe already has memory description in TLFS docs, and de facto it involved in memory operation only in allocation\deallocation stage. Read\write memory guest OS made independently of hypervisor. Yes, of course hypervisor make memory access attribution\isolates guest OS memory from root OS, and other OSes, but it made by hardware feature like EPT and don't need evolve hypervisor on every memory reading\writing operation.

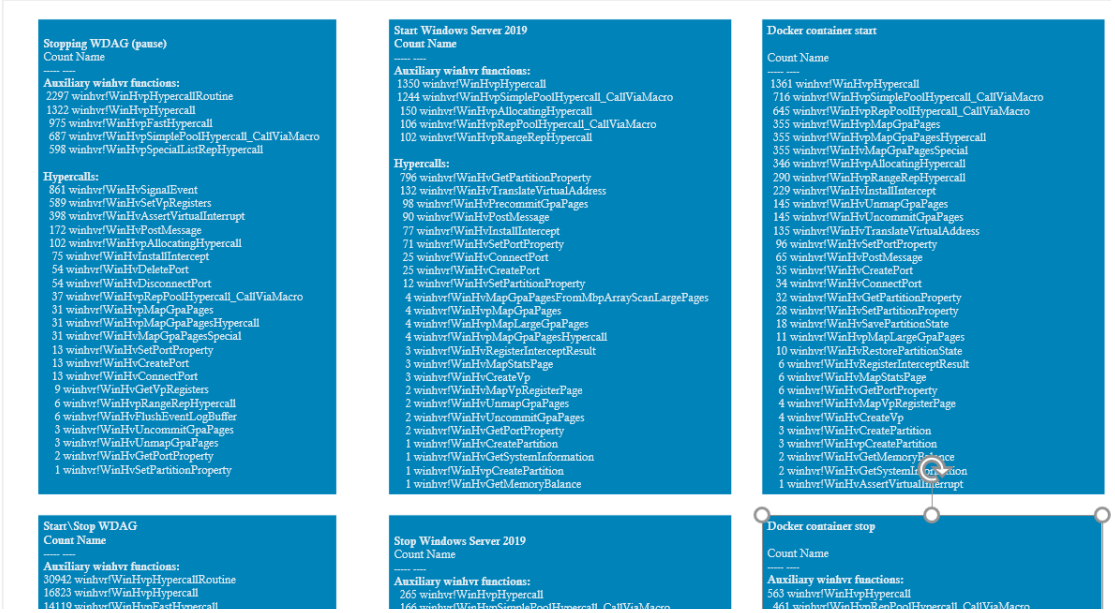
I describe memory access to Full VM, WDAG, Windows Sandbox and shortly Docker containers. During research hvmm driver was created. Main function of it – provide interface for reading guest OS memory from root OS without access to vid.sys, hvix64.exe API. That driver was integrated to LiveCloudKd project.

Detailed description of Hyper-V internals we will see in part 2 of Windows Internals book, 7th, writing by Andrea Allievi. But while book under develop, you can read shot description of Hyper-V guest OS memory structures in this article :)

Let's beginning.

**Direct memory access to Full VM and Hyper-V containers**

Vmwp.exe is the main process, that controls guest OS execution and provide device emulation. It is launched by vmms.exe in case of starting Full VM, or vmcompute.exe in case of starting container. When starting, the vmwp.exe process accesses to the hypervisor interfaces (hypercalls) through the vid.dll interface. I got hypercall usage statistic for Windows Server 2019 VM, Docker container in Hyper-V isolation mode (nanoserver image: 1809) and WDAG container. The WDAG container generates too many hypercalls, so due some delays, caused by the debugger writing results, the container immediately started to turn off after being turned on (WDAG-manage application hvsimgr.exe controls execution timeouts of some procedures), and therefore the WDAG results contains summary indicator (I want to try dtrace, relatively recently developed under Windows, to collect such statistics - in theory, it should reduce the cost of recording the collected data and remove hvsimgr.exe timeout limitations). Separately there is recorded shutdown statistics, so that the approximate order can be estimated. In comparing to Full VM, it is quite large:






What categories of hypercalls can be distinguished from this calling statistics? Partition creation, configuring its properties, creating virtual processors and virtual ports (use to send signals, messages), setting interceptions, and various hypercalls for memory management.

See to winhvr.sys!WinHvMapGpaPagesFromMbpArrayScanLargePages function. Rdx contains page number, rsi - size (in pages).

When we start Windows Server 2019 with 1500 Mb of RAM, we got:

1st call rdx=0000000000000000 rsi=000000000005dc00
2nd call rdx=00000000000f8000 rsi=0000000000000800
3rd call rdx=0000000000fff800 rsi=0000000000000800

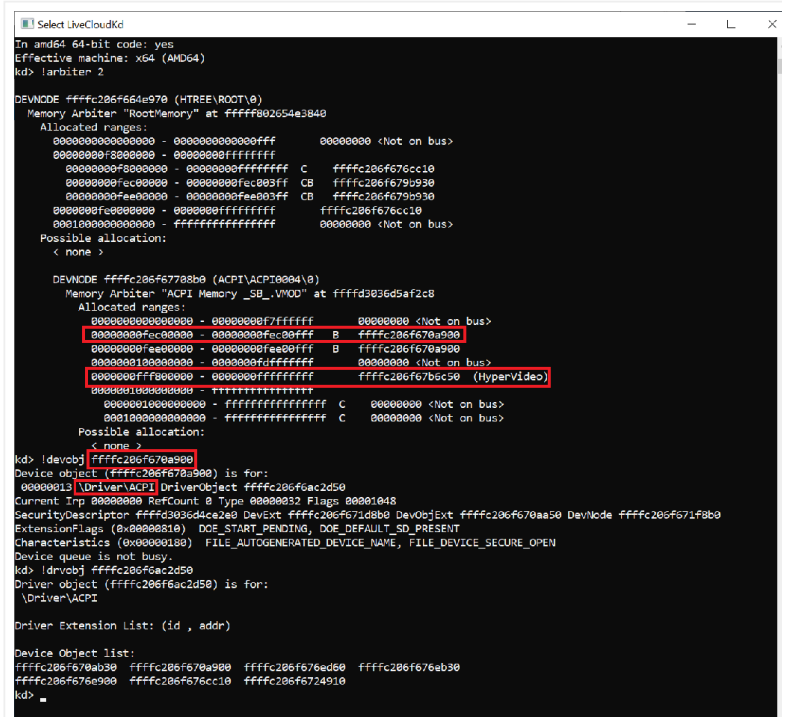
When we start Windows Server 2019 with 2300 Mb of RAM, we got:

1st call: rdx=0000000000000000 rsi=000000000008fc00
2nd call: rdx=00000000000f8000 rsi=0000000000000800
3rd call: rdx=0000000000fff800 rsi=000000000000024a

Call stack:

| 1st call | 2nd and 3rd calls |
|---|---|
| 00 winhvr!WinHvMapGpaPagesFromMbpArrayScanLargePages | # Call Site |
| 01 Vid!VsmmHvpMapGpasFromMbpArray | 00 winhvr!WinHvMapGpaPagesFromMbpArrayScanLargePages |
| 02 Vid!VsmmHvpMapGpasFromMemoryBlockRange | 01 Vid!VsmmHvpMapGpasFromMbpArray |
| 03 Vid!VsmmHvMapGpasFromMemoryBlock | 02 Vid!VsmmHvpMapGpasFromMemoryBlockRange |
| 04 Vid!VsmmAdjustGpaSpaceForMemoryBlockRange | 03 Vid!VsmmHvMapGpasFromMemoryBlock |
| 05 Vid!VsmmCreateMemoryBlockGpaRange | 04 Vid!VsmmAdjustGpaSpaceForMemoryBlockRange |
| 06 Vid!VidIoControlPartition | 05 Vid!VsmmCreateMemoryBlockGpaRange |
| 07 Vid!VidIoControlDispatch | 06 Vid!VidIoControlPartition |
| 08 Vid!VidIoControlPreProcess | 07 Vid!VidIoControlDispatch |
| ……WDF Calls…… | 08 Vid!VidIoControlPreProcess |
| 0d nt!IofCallDriver | ……WDF Calls…… |
| 0e nt!IopSynchronousServiceTail | 0d nt!IofCallDriver |
| 0f nt!IopXxxControlFile | 0e nt!IopSynchronousServiceTail |
| 10 nt!NtDeviceIoControlFile | 0f nt!IopXxxControlFile |
| 11 nt!KiSystemServiceCopyEnd | 10 nt!NtDeviceIoControlFile |
| 12 ntdll!NtDeviceIoControlFile | 11 nt!KiSystemServiceCopyEnd |
| 13 vid_7ffb4de20000!VidCreateMemoryBlockGpaRange | 12 ntdll!NtDeviceIoControlFile |
| 14 vmwp!GpaRangeMbBacked::Initialize | 13 vid_7ffb4de20000!VidCreateMemoryBlockGpaRange |
| 15 vmwp!MemoryManager::CreateGpaRangeInternal | 14 vmwp!MemoryManager::CreateMemoryBlockGpaRange |
| 16 vmwp!MemoryManager::CreateMemoryBlock | 15 vmwp!VmbComGpaRange::VmbComGpaRange |
| 17 vmwp!MemoryManager::CreateRamMemoryBlocks | 16 vmwp!Vml::VmComMultiInstanceObject<VmbComGpaRange>::CreateInstance |
| 18 **vmwp!MemoryManager::CreateRam** | 17 vmwp!Vml::CreateComObject<VmbComGpaRange,IMemoryManager |
| 19 vmwp!VirtualMachine::ConstructGuestRam | 18 vmwp!VmbComMemoryBlock::CreateGpaRange |
| 1a vmwp!WorkerTaskStarting::RunCleanStartSteps | 19 **vmuidevices!VideoSynthDevice::SetupVramGpaRange** |
| 1b vmwp!WorkerTaskStarting::RunTask | 1a vmuidevices!VideoSynthDevice::SynthVidOnVramLocation |
| 1c vmwp!WorkerAsyncTask<VmPerf::Vmwp::StartingTask>::Execute | 1b vmuidevices!VideoSynthDevice::OnMessageReceived |
| 1d vmwp!VirtualMachine::DoStateChangeTask | 1c vmuidevices!VMBusPipeIO::OnReadCompletion |
| 1e vmwp!VirtualMachine::StartInternal | 1d vmuidevices!VMBusPipeIO::ProcessCompletionList |
| | 1e vmuidevices!VMBusPipeIO::HandleCompletions |
| | 1f vmuidevices!VMBusPipeIO::OnCompletion |

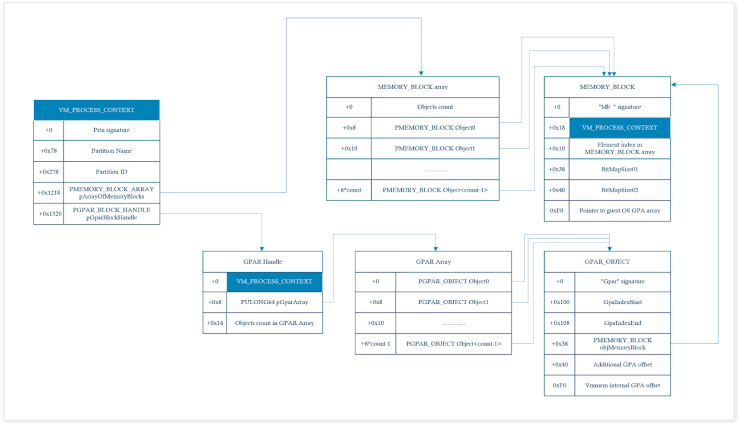
The last memory block is mapped memory of video adapter. A one-page-size block is used for an ACPI devices.




Among other things driver hvmm.sys is needed to remove vmwp.exe protection, that prevent dll injection to that process. That driver works with partition handle with Prtn-signature (VM_PROCESS_CONTEXT), but there is second type, that supporting by vid.sys - EXO-partitions. EXO-partitions can be created using WinHv Platform API Library (https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/hypervisor-platform), which allows third-party developers to make their virtualization solutions compatible with Hyper-V and run it simultaneously with native Hyper-V VMs. Currently VirtualBox, Qemu, Bochs (f.e. in applepie implementation) have this supporting. VMware, one year after the appearance of these APIs in Windows 1803, finally added support to its VMware Workstation product too. Probably, a new assembly of VMware will be released after the release of Windows 10, build 1908.

However, it is still possible to use the vid.dll interface without a driver in Windows Server 2016 and earlier. API execution lock is missing in vid.sys in that OS, and driver hvmm.sys is not needed in that environment. But WDAG and Windows Sandbox containers are presenting in Windows 10 only, where API is locked.

What structures will be needed to work with Guest OS memory? I tried to visualize them in a diagram. In the future, while reading the article, it should become clearly, how they are using.




Objects:

- Partition handle (VM_PROCESS_CONTEXT);
- GPAR-handle (GPAR - Guest physical Address Range);
- Array of GPAR elements (GPAR Array);
- Array of MBlock-objects (MBlock Array. MBlock – memory block GPA range);
- GPAR-object (GPAR_OBJECT);
- MBlock-object (MEMORY_BLOCK).

Partition handle is the main object, which is used by hvmm driver. When user mode section of partition handle is created, its kernel mode part contains all the necessary information about the created partition. The search algorithm for the user mode component hasn't changed since Windows Server 2008 R2, and this component can be obtained by enumeration of handles, opened by the vmwp.exe process. For this, find all open file descriptors with the names like \Device\000000 and try to get partition name.

```
if (memcmp(pObjectNameInformation->Name.Buffer, L"\\Device\\000000", sizeof(L"\\Device\\000000") - sizeof(
{
    if (SdkHvmmGetPartitionFriendlyName(PartitionEntry, DuplicatedHandle) == TRUE)
    {
        PartitionEntry->OriginalVidPartitionHandle = Handle;
        Ret = TRUE;
    }
}
```

If the name can be obtained, it means, that we found a valid partition handle. In my practice, there are 3 similar objects for each Full VM or container. If we pass the obtained values to the kernel function nt!ObReferenceObjectByHandle, then in two cases it returns NULL, that means objects are invalid. For the current descriptor, we get the pointer to the partition handle.

Yes, object pointers offsets inside partition handle are fixed and differ for each version of Windows. But for same version of Windows they aren't changed, so the method is quite reliable.

Partition handle contains fields, that point to an array of MBlock objects (initialized in vid.sys!VsmmMemoryBlockpInitialize) and an array of GPAR objects (initialized in vid.sys!VsmmGpaRangepInitialize).

By the way, you do not need to confuse the partition handle with the Windows 10 memory partition structure, which !partition WinDBG command displays. This is the _MI_PARTITION structure, which contains basic information about current state of the operating system memory. This object is created without an active hypervisor (or active – no matter).

```
0: kd> !partition
Partition0 fffff806388418c0 MemoryPartition0
0: kd> !kdexts.partition fffff806388418c0
PartitionObject @ ffff948af3895dc0 (MemoryPartition0)
_MI_PARTITION 0 @ fffff806388418c0
    MemoryRuns: 0000000000000000
    MemoryNodeRuns: ffff948af3843330
    TotalHugeIoSpaceRanges: 0 GB
    AvailablePages:         0n969332 ( 3 Gb 714 Mb 464 K
    ResidentAvailablePages: 0n1205109 ( 4 Gb 611 Mb 468
       0 _MI_NODE_INFORMATION @ ffffe1800000e000
         TotalPagesEntireNode:   0x13fe64
                   Zeroed                    Free
         1GB       ________0 ( 0)                  ____
         2MB       ________0 ( 0)                  ____
         64KB      ________0 ( 0)                  ____
         4KB       ____80472 ( 314 Mb 352 Kb)      ____
         Node Free Memory:     ( 314 Mb 352 Kb )
         InUse Memory:         ( 4 Gb 708 Mb 48 Kb )
         TotalNodeMemory:      ( 4 Gb 1022 Mb 400 Kb )
       1 _MI_NODE_INFORMATION @ ffffe1800000e7c0
         TotalPagesEntireNode:          0
                   Zeroed                    Free
         1GB       ________0 ( 0)                  ____
         2MB       ________0 ( 0)                  ____
         64KB      ________0 ( 0)                  ____
         4KB       ________0 ( 0)                  ____
         Node Free Memory:     ( 0 )
         InUse Memory:         ( 0 )
```

You can read more about it in the 1st part of Windows Internals book (7th edition). I couldn't find that information in MSDN (current Microsoft Docs).

Containers and Full VM have different accessing memory methods, so let's look at memory reading examples for both. Let's start with Full VM based on Windows Server 2019.

**Full VM memory reading**

LiveCloudKd application passes the request to the driver for reading guest OS memory block. The data, required for the request, is packed into the GPA_INFO structure. This structure contains start memory address, number of bytes to read and service information about virtual machine partition (PID vmwp, partition id).

```
GpaInfo                                  _GPA
  PartitionInfo                          _PAR
    StartPage                0x8058000   unsi
    BytesCount               0x1000      unsi
```

First, get partition handle. To do this, just call the nt!ObReferenceObjectByHandle function with the passed descriptor.

```
Status = ObReferenceObjectByHandle(hVmPartition
    READ_CONTROL,
    *IoFileObjectType,
    KernelMode,
    &objVmPartition,
    NULL);
```

Type of getting object is FILE_OBJECT. To gain access to the body of the descriptor, you must get a pointer to FsContext.

```
pPartitionHandle = (PVM_PROCESS_CONTEXT)((PCHAR)objVmPartition->FsCont
```

Beginning part of partition handle looks like:

```
3: kd> dc FFFFCE08AE03E000 L30
ffffce08`ae03e000  6e747250 00000000 00000000 00000000  Prtn....
ffffce08`ae03e010  0480001c 00000000 00000000 00000000  ........
ffffce08`ae03e020  00000001 00000000 00000000 00000000  ........
ffffce08`ae03e030  00000000 00000000 00000000 00000000  ........
ffffce08`ae03e040  00000000 00000000 00000000 00000000  ........
ffffce08`ae03e050  00000000 00000000 004a0048 00000000  ........
ffffce08`ae03e060  adb93230 ffffce08 02000014 00000000  02......
ffffce08`ae03e070  ae03e078 ffffce08 00690057 0032006e  x.....
ffffce08`ae03e080  00310030 002d0039 00320030 00000000  0.1.9.-
```

```
if (objVmPartition->FsContext != NULL)
{
    pPartitionHandle = (PVM_PROCESS_CONTEXT)((PCHAR)objVmPartition->FsCo

    switch (pPartitionHandle->VmType)
    {
    case VidVmTypeDockerHyperVContainerUserName:
    case VidVmTypeDockerHyperVContainerGUID:
    case VidVmTypeContainer:

        Ret = VidGetContainerMemoryBlock(pPartitionHandle, pBuffer, len,

        break;

    case VidVmTypeFullWin10VM:
    case VidVmTypeFullWinSrvVMSecure:
    case VidVmTypeFullWinSrvVM:

        Ret = VidGetFullVmMemoryBlock(pPartitionHandle, pBuffer, len, GP
        break;

    default:
        break;
    }
}
```

The first 0x278 bytes contain section signature, the name and its identifier. The size of structure is not small (0x3EF0 for Windows Server 2019) and it is different for different operating systems. The exact size of partition handle can be found in vid.sys!VidCreatePartition (by the amount of memory allocated for it). We will not need it in driver.

When we get partition handle type (VmType), we can perform one of two procedures for memory blocks reading. There are actually quite a lot of possible VmType values, and moreover, they differ for different versions of operating systems. For example, VmType for Full VM in Windows 10 and Windows Server 2019 have different values. Not all of them have been investigated (especially for operating systems such as

Linux, because WinDBG, that launched by LiveCloudKd, doesn't work with them). But finally partitions of virtual machines were divided into two categories: container's partitions and Full VM partitions.

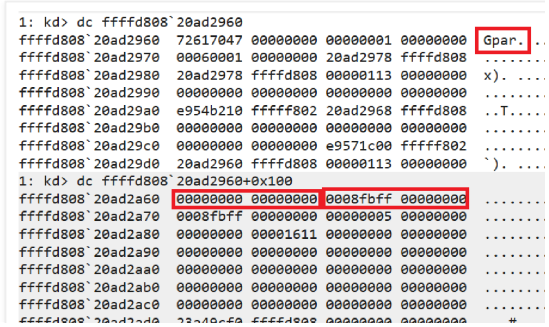
The hvmm.sys!VidGetFullVmMemoryBlock function at the input receives a section descriptor, a buffer in which to write the received data, the size of the buffer in bytes and the GPA of the virtual machine.

```
BOOLEAN VidGetFullVmMemoryBlock(PVM_PROCESS_CONTEXT pPartitionHandle, PCHAR pBuffer, ULONG len, ULONG64 GPA)
```

GPA – it is page number, which is calculated: GPA = GpaInfo.StartAddress / PAGE_SIZE;

The start address should be aligned on the page boundary, if the hvmm driver function is called directly (LiveCloudKdSdk prepared usermode buffer for that).

Next, we need to find GPAR object, that describes the requested GPA. Each GPA is included in the memory block, previously allocated by the hypervisor, and this memory block is described by the GPAR object. Fields GpaIndexStart and GpaIndexEnd are located, respectively, at the offsets 0x100 and 0x108 of the GPAR objects. You can understand whether the GPAR object describes the GPA or not, by the value of these fields. For example:




This GPAR object control GPA from 0 to 0x8fbff.

GPAR objects count in Full VM are much smaller than in containers. For example, Generation 2 Full VM has 3-4 GPAR objects, containers have about 780. Then guest OS has more memory, then more blocks it allocates with HvMapGpaPages* hypercalls and, correspondingly, there are greater numbers of GPAR objects. The maximum range of GPAs, described by GPAR object, that I met, was 0x96000 pages.

Let's get back to our driver. We can find GPAR object using hvmm.sys!VidGetGparObjectForGpa function. Partition handle and GPA are passed to the function. How does it work? As described above, each partition handle has a pointer to a GPA block descriptor. This is a structure, which, among other things, contains a pointer to the partition handle itself, a pointer to array with pointers to GPAR objects, and the count of elements in the array of GPAR objects (see the diagram of the relationship of structures above).

```
typedef struct _GPAR_BLOCK_HAN
    PVOID PartitionHandle;
    PGPAR_OBJECT GparArray;
    UINT32 Unknown01;
    UINT32 CountInGparArray;
} GPAR_BLOCK_HANDLE, *PGPAR_BL
```

```
typedef struct _GPAR_OBJECT {
    CHAR cGparSignature[0x8]; // "GPAR" signature - eq GPA Range
    CHAR Unknown01[0xF8];
    UINT64 GpaIndexStart; //offset +0x100, size 0x8
    UINT64 GpaIndexEnd;  //offset +0x108, size 0x8
    UINT64 UnknowParam01;
    UINT64 UnknowParam02;
    UINT32 KernelMemoryBlockGpaRangeFlags; //offset +0x120, size 0x4
    CHAR Unknown02[0x4C];
    PMEMORY_BLOCK objMBlock; //offset +0x170, size 0x8 //in Windows 10 20H1 up to
    ULONG64 SomeGpaOffset; //offset +0x178, size 0x8
    ULONG64 VmmMemGpaOffset;//offset +0x180, size 0x8
} GPAR_OBJECT, *PGPAR_OBJECT;
```

```
pGparBlockHandle = pPartitionHandle->pGparBlockHandle;

if (pGparBlockHandle == 0)
{
    pGparBlockHandle = pPartitionHandle->pGparBlockHandle20H1;
    if (pGparBlockHandle == 0) {
        KDbgPrintString("\tSomething wrong with offset of GparBlockHa
        return NULL;
    }
}
```

When we got this information, we can run cycle through the GPAR objects and find 1 GPAR the object, that is responsible for the GPA. Code is quite simple, as you can see. This is a simplified implementation of VsmmLookupMemoryBlockByHandle function of vid.sys driver.

Vid.sys driver also has additional procedure for encrypted memory reading - VsmmpSecureReadMemoryBlockPageRangeInternal. It uses AES XTS through BCryptEncrypt\BCryptDecrypt functions from ksecdd.sys driver. I can't find in what cases they are used, because even for Shielded VMs with TPM enabled, standard memory reading operations are working. Perhaps some special areas are encrypted, but they haven't been found still. Obviously, guest OS directly make reading/writing operations to the already allocated memory area without calling any functions from vid.sys. All exceptions must be caught and handled by the hypervisor. Accordingly, if the root OS encrypts some parts of the memory, then the guest OS will not be able to transparently access them.

Go back to the hvmm code.

When we found a suitable GPAR object, we exit from cycle.

```
Index = pGparBlockHandle->CountInGparArray;
pGparArray = (PUINT64)pGparBlockHandle->GparArray;

if (pGparArray == 0)
{
    KDbgPrintString("\tSomething wrong with offset of Gpar array");
    return NULL;
}

for (LONG i = Index - 1; i >= 0; i--)
{
    uElement = *((PUINT64)pGparArray + i);
    if (uElement != 0)
    {
        objGpar = (PGPAR_OBJECT)uElement;
        KDbgLog("pGparElement->GpaIndexStart", objGpar->GpaIndexStart);
        KDbgLog("pGparElement->GpaIndexEnd",objGpar->GpaIndexEnd);

        if ((GPA >= objGpar->GpaIndexStart) && (GPA <= objGpar->GpaIndexEnd))
        {
            return objGpar;
        }
    }
    else
    {
        KDbgLog("\tGpar Element is NULL, i = ", i);
    }
} // end for
```

There are GPAR objects exist, that don't describe the GPA, but instead of the necessary data, contain a pointer to a certain usermode structure inside the vmwp.exe process. They are tied to the memory allocated for virtual Hyper-V devices. Usually, there is 1 such GPAR object per partition (see content of that memory later in Docker part of that article).

```
if (objGpar->GpaIndexStart == objGpar->GpaIndexEnd) {
    KDbgPrintString("MBlock in GPAR object is vmwp.exe des
    return FALSE;
}
```

We don't need in that objects during memory reading operations.

What data is contained in the GPAR object and will help to read the data from the guest OS? This is another data type - an MBlock object (MEMORY_BLOCK). It contains guest PFN data and other useful information. A fairly large structure, at the beginning contains the signature "Mb  ".

```
typedef struct _MEMORY_BLOCK {
    CHAR cMblockString[0x8]; // "Mb  " sign
    PVOID PartitionHandle; // size 0x8
    CHAR Unknown01[0x8];
    ULONG MbHandle;
    CHAR Unknown02[0x1C];
    ULONG64 BitMapSize01; // offset 0x38, s
    ULONG64 BitMapSize02; // offset 0x40, s
    CHAR Unknown03[0xA8];
    PULONG64 pGuestGPAArray; //offset 0xF0,
} MEMORY_BLOCK, *PMEMORY_BLOCK;
```

From all the fields, we need only a pointer to the GPA array. Size of the array element is 16 bytes. One 8-byte part contains the GPA (in guest OS), and other 8-byte part contains the SPA information (in root OS).

```
1: kd> dq 0xffffcc8150400000
ffffcc81`50400000  00000000`00078c00 00f208
ffffcc81`50400010  00000000`00078c01 00f208
ffffcc81`50400020  00000000`00078c02 00f208
ffffcc81`50400030  00000000`00078c03 00f208
ffffcc81`50400040  00000000`00078c04 00f208
ffffcc81`50400050  00000000`00078c05 00f208
ffffcc81`50400060  00000000`00078c06 00f208
ffffcc81`50400070  00000000`00078c07 00f208
```

We can calculate SPA by following formula:

```
objMBlock = objGpar->objMBlock;

HostSPA = *(PULONG)((PCHAR)objMBlock->pGuestGPAArray + 0x10 * (GPA- objGpar->GpaInde
```

For SPA reading, we need mapped it to root OS virtual address space. Use MDL structure for this:

```
pMDL = IoAllocateMdl(VirtualAddress, PAGE_SIZE, FALSE, FA
```

There is an array of PFN at the end of each MDL structure. A pointer to it can be obtained using MmGetMdlPfnArray macro. When we received the pointer, we had wrote HostSPA index to it. Of course, it is possible to put in MDL more than one PFN at one time. But there is a chance to get to the border of GPAR blocks, therefore memory reading is done page by page. For Full VM, this is not very profitable, since the size of each block is large enough, but speed is still good.

```
MdlPfnArray = MmGetMdlPfnArray(pMDL);
*MdlPfnArray = HostSPA;

__try
{
    SourceAddress = MmMapLockedPagesSpecifyCache(pMDL, KernelMode, MmCached, NULL, FALSE, Normal

    if (!SourceAddress)
    {
        IoFreeMdl(pMDL);
        return FALSE;
    }

    RtlCopyMemory(pBuffer + i * PAGE_SIZE, SourceAddress, PAGE_SIZE);

    MmUnmapLockedPages(SourceAddress, pMDL);
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    KDbgLog("   RtlCopyMemory failed", GetExceptionCode());
}

IoFreeMdl(pMDL);
```

Next, we get virtual address using the nt!MmMapLockedPagesSpecifyCache function and use it to copy guest OS memory block using nt!RtlCopyMemory. Accordingly, reading is performed in a loop. 1 memory page is copied on 1 iteration. During copying, it is recommended to pause the virtual machine in order to avoid memory modification during reading. In LiveCloudKdSdk, the SdkControlVmState function is implemented for this. It suspends the execution of the virtual machine either by the usual powershell-cmdlets Suspend-VM\Resume-VM, or works with the special register of each virtual processor calling HvWriteVpRegister hypercall and set the HvRegisterExplicitSuspend register to 0 (resume) or 1 (suspend).

**Container memory reading**

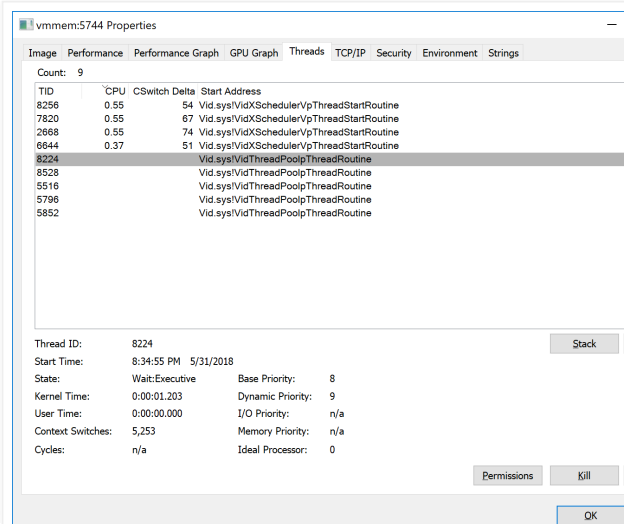
Consider reading the container's memory on Windows Defender Application Guard example (to use it, it's need install same name component in Windows 10. It has been present since the 1803 build). Access to memory of Windows Sandbox and docker container in Hyper-V isolation mode is same.

It made by next function of hvmm.sys driver:

```
BOOLEAN VidGetContainerMemoryBlock(PVM_PROCESS_CONTEXT pPartitionHandle, PCHAR pBuffer, ULONG len, ULONG64 GPA)
```

Before executing it, as for Full VM, we must get partition handle first. Then, we will additionally need vmmem process handle. This process is created, when containers work, and works in kernel mode only.

We can see it's threads, when launched container on a 4-processor PC (there are no user mode threads):




The vmmem process descriptor is present in the partition handle. We can find it, using 'scrP' signature (see the hvmm!VidFindVmmemHandle function for details).

We get a pointer to the GPAR object, as same way for reading memory in Full VM. Next we see differences - other fields of the GPAR structure are used to read blocks of memory. VmmMemGpaOffset - the main offset, which allows us convert GPA to SPA for a specific memory block. There is additional offset present (SomeGpaOffset), which can influence to final result, but during my experiments it was always 0.

```
typedef struct _GPAR_OBJECT {
    CHAR cGparSignature[0x8]; // "GPAR" signature - eq GPA Range
    CHAR Unknown01[0xF8];
    UINT64 GpaIndexStart; //offset +0x100, size 0x8
    UINT64 GpaIndexEnd;  //offset +0x108, size 0x8
    UINT64 UnknowParam01;
    UINT64 UnknowParam02;
    UINT32 KernelMemoryBlockGpaRangeFlags; //offset +0x120, size 0x4
    CHAR Unknown02[0x4C];
    PMEMORY_BLOCK objMBlock; //offset +0x170, size 0x8 //in Windows 10 20H1 up t
    ULONG64 SomeGpaOffset; //offset +0x178, size 0x8
    ULONG64 VmmMemGpaOffset;//offset +0x180, size 0x8
} GPAR_OBJECT, *PGPAR_OBJECT;
```

Next, we calculate source address, using the following formula and copy data block directly from the address space of vmmem process:

```
for (i = 0; i < uBlocks; i++)
{

    objGpar = VidGetGparObjectForGpa(pPartitionHandle, GPA+i);

    if (objGpar == NULL) {
        return FALSE;
    }

    SourceAddress = (GPA + i - objGpar->GpaIndexStart - objGpar->SomeGpaOffset) * PAGE_SIZE + objGpar->Vm
    if (objGpar->SomeGpaOffset != 0) {
        KDbgLog16("   objGpar->SomeGpaOffset", objGpar->SomeGpaOffset);
    }
    __try {
        KeReadProcessMemory((PEPROCESS)g_vmmemHandle, (PVOID)SourceAddress, pBuffer + i * PAGE_SIZE, PAG
        Ret = TRUE;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        KDbgLog("   KeReadProcessMemory excpetion", GetExceptionCode());
        Ret = FALSE;
    }
```

Now we can see key difference between reading container memory from reading Full VM memory: we need copy data from virtual memory of the vmmem process. There is no need for memory mapping using MDL.

**Hyper-V memory API**

Direct access to memory without corresponding exported Windows functions is interesting, but a more reliable method is to use some of APIs, which is provided by Microsoft. But for reliability you will have to pay the restrictions imposed by Microsoft on these APIs. In particular, for hypercalls they work only with Full VM and for containers they always return FALSE, additionally they read\write no more than 0x10 bytes at one time. The vid.dll function API is generally forbidden to be called from any module other than the vmwp.exe process in latest versions of Windows.

Vid.dll has next functions for reading\writing memory:

- VidTranslateGvaToGpa
- VidReadMemoryBlockPageRange (wrapper on vid.sys!VidReadWriteMemoryBlockPageRange)
- VidWriteMemoryBlockPageRange (wrapper on vid.sys!VidReadWriteMemoryBlockPageRange)

And hypercalls (it must be called from ring 0):

- HvTranslateVirtualAddress
- HvWriteGPA
- HvReadGPA

See it in more detailed.

**Reading\writing memory using hypercalls**

HvReadGpa using is quite simple, if you don't take, that memory block shouldn't fall on the page boundary. Otherwise, the reading operation will be broken and end of block, that must be read from the second page, will contain zero bytes. Blocking separation is implemented in the usermode part of LiveCloudKdSdk. Driver hvmm calls WinHvReadGPA - HvReadGpa wrapper from winhvr.sys driver. You can call HvReadGpa directly through vmcall, but before you will have to additionally perform operations to prepare hypercall parameters.

```
for (i = 0; i < uBlocks; i++)
{
    Status = WinHvReadGpa(GpaInfo.PartitionId, VpIndex, GpaInfo.StartPage+i*VID_READ_WRITE_GPA_BUFFER_SIZE, VID_READ_WRITE_GPA_BUFFER_SIZE, ControlFlags, &AccessResult, pBuffer + i * VID_READ_WRITE_

    KDbgLog("Status of WinHvReadGpa", Status);
    KDbgLog("AccessResult", AccessResult.ResultCode);
}
```

Boundary checking for writing operation was made in hvmm.sys driver.

```
if (PageBoundaryCheckLowerBorder == PageBoundaryCheckHighBorder)
{
    Status = WinHvWriteGpa(GpaInfo.PartitionId, VpIndex, GpaInfo.StartPage + i * VID_READ_WRITE_GPA_BUFFER_SIZE, VID_READ_WRITE_GPA_BUFFER_SIZE, ControlFlags, (PVOID)uPosition, &
}
else
{
    PageBoundaryCheck1WriteBlockSize = (PAGE_SIZE - ((GpaInfo.StartPage + i * VID_READ_WRITE_GPA_BUFFER_SIZE) & 0xFFF));
    PageBoundaryCheck2WriteBlockSize = VID_READ_WRITE_GPA_BUFFER_SIZE - PageBoundaryCheck1WriteBlockSize;

    Status = WinHvWriteGpa(GpaInfo.PartitionId, VpIndex, GpaInfo.StartPage + i * VID_READ_WRITE_GPA_BUFFER_SIZE, PageBoundaryCheck1WriteBlockSize, ControlFlags, (PVOID)uPosition,
    Status = WinHvWriteGpa(GpaInfo.PartitionId, VpIndex, GpaInfo.StartPage + i * VID_READ_WRITE_GPA_BUFFER_SIZE+ PageBoundaryCheck1WriteBlockSize, PageBoundaryCheck2WriteBlockSi
}
```

An additional check is performed before reading virtual address space using winhvr.sys!WinHvTranslateVirtualAddress. The function converts a virtual address into a physical one, using the current context of the CPU (and accordingly, CR3 register).

Possible validation options (LiveCloudKd uses only HV_TRANSLATE_GVA_VALIDATE_READ and HV_TRANSLATE_GVA_VALIDATE_WRITE).

```
#define HV_TRANSLATE_GVA_VALIDATE_READ       (0x0001)
#define HV_TRANSLATE_GVA_VALIDATE_WRITE      (0x0002)
#define HV_TRANSLATE_GVA_VALIDATE_EXECUTE    (0x0004)
#define HV_TRANSLATE_GVA_PRIVILEGE_EXEMPT    (0x0008)
#define HV_TRANSLATE_GVA_SET_PAGE_TABLE_BITS (0x0010)
#define HV_TRANSLATE_GVA_TLB_FLUSH_INHIBIT   (0x0020)
#define HV_TRANSLATE_GVA_CONTROL_MASK        (0x003F)
```

WinDBG in memory dump mode works with physical addresses only (for debugger it is file offsets). Accordingly, it makes all the work for converting virtual address to physical, therefore we don't need to do additional hypercall for checking memory address.

**Microsoft Hyper-V Virtualization Infrastructure Driver Library (vid.dll) API**

First, see vid.dll!VidReadMemoryBlockPageRange

```
VIDDLLAPI
BOOL
WINAPI
VidReadMemoryBlockPageRange(
    __in PT_HANDLE Partition,
    __in MB_HANDLE MemoryBlock,
    __in MB_PAGE_INDEX StartMbp,
    __in UINT64 MbpCount,
    __out_bcount(BufferSize)
        PVOID ClientBuffer,
    __in UINT64 BufferSize
);
```

Partition parameter – it is user mode partition handle;
ClientBuffer – pointer to memory region, where result will be stored;
BufferSize – yes, buffer size, and nothing more;

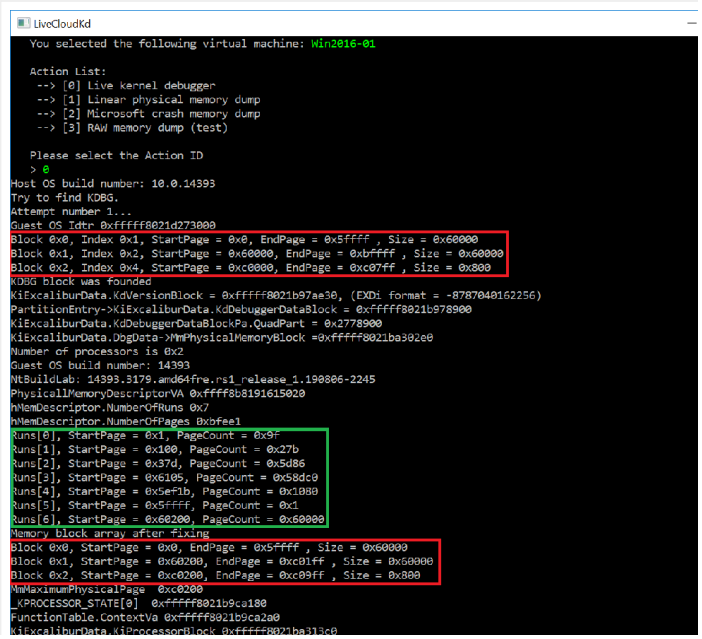
Two parameters can cause some questions: MemoryBlock and StartMbp. MemoryBlock is number of the MBlock object from which data will be read. In Windows Server 2008 R2 kernel-mode handle must be pointed as that parameter (yes, the user mode application contained kernel mode descriptor addresses - the original version of LiveCloudKd was built on this logic):

https://github.com/comaeio/LiveCloudKd/blob/07ac5901ff5cac5258033f1dd95cfc2bd0e06815/hvdd/memoryblock.c#L159 (buffer contains memory of vmwp.exe)

```
if ((Buffer[i] >= MmNonPagedPoolStart) && (Buffer[i] < MmNo
{
    for (j = 0; j < BlockIndex; j++)
    {
        if (Blocks[j].MemoryHandle == (MB_HANDLE)Buffer[i])
        {
            Blocks[j].Hits += 1;
            break;
        }
```

StartMbp is index, which is equal to physical memory page number. We just need to get the GPA and divide it into PAGE_SIZE (0x1000). The page size in this case is virtual. For example, when ntoskrnl.exe image memory page is usually 2 Mb LARGE_PAGE, but the page numbers will still be 4 Kb granular for that region. Buffer can be specified less, then less data will be written to it. Everything is clear, with one exception - this index is relative to the beginning of MB_HANDLE MemoryBlock. For example, for the first memory block, index will match with physical memory

page number. If blocks are placed continuously, index of second block will be equal to page number minus first block size. Index of third block will be equal to page number minus the size of the first block and minus the size of the second block. Everything seems to be clear. The main problem is that physical memory blocks are not continuous. Moreover, these boundaries cannot be easily determined from the user mode. Microsoft didn't provide such APIs even from the time of Windows Server 2008 R2.



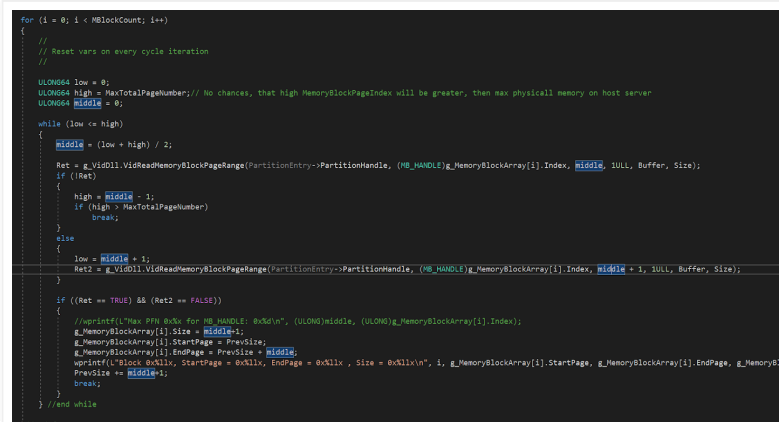

Matt used a separate function for searching descriptors in memory, but Microsoft closed this opportunity by replacing the descriptors with their indexes in the table, located in kernel mode, and therefore I used vid.dll! VidReadMemoryBlockPageRange function.

```
for (i = 0; i < MAX_INDEX_BLOCK_NUMBER; i++)
{
    Ret = g_VidDll.VidReadMemoryBlockPageRange(PartitionEntry->PartitionHandle, (MB_HANDLE)i,MemoryBlockPageIndex,1ULL, Buf

    if (Ret == TRUE) {
        //wprintf(L"Valid MB_HANDLE: %d\n", (ULONG)i);
        MBlockCount++;
        IndexArray[i] = 1;
    }
```

First, we can get the HANDLE numbers by doing a simple search, reading first memory page of each block. If function returns TRUE – it means, that block exists, if FALSE - block doesn't exist. Based on practical experience, I determined the maximum size of the index to be 0x400. As

we saw above, a large number of indexes are observed only for containers such as WDAG and Windows Sandbox, due to the fact that each file is mapped in a separate block.

When we get array with indexes, we have could determine maximum block size by slightly modifying the binary searching algorithm in the array.



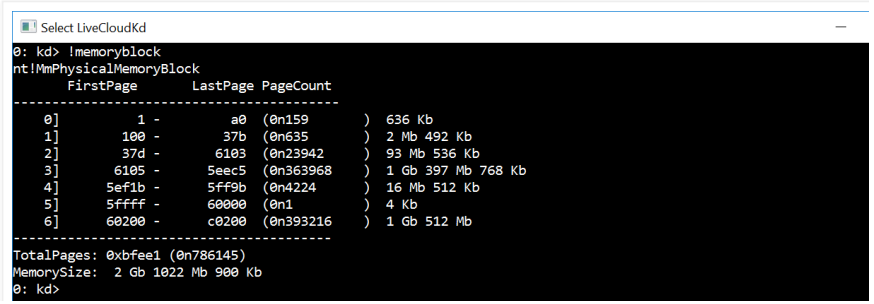
```
for (i = 0; i < MBlockCount; i++)
{
    //
    // Reset vars on every cycle iteration
    //

    ULONG64 low = 0;
    ULONG64 high = MaxTotalPageNumber;// No chances, that high MemoryBlockPageIndex will be greater, then max physicall memory on host server
    ULONG64 middle = 0;

    while (low <= high)
    {
        middle = (low + high) / 2;

        Ret = g_VidDll.VidReadMemoryBlockPageRange(PartitionEntry->PartitionHandle, (MB_HANDLE)g_MemoryBlockArray[i].Index, middle, 1ULL, Buffer, Size);
        if (!Ret)
        {
            high = middle - 1;
            if (high > MaxTotalPageNumber)
                break;
        }
        else
        {
            low = middle + 1;
            Ret2 = g_VidDll.VidReadMemoryBlockPageRange(PartitionEntry->PartitionHandle, (MB_HANDLE)g_MemoryBlockArray[i].Index, middle + 1, 1ULL, Buffer, Size);
        }

        if ((Ret == TRUE) && (Ret2 == FALSE))
        {
            //wprintf(L"Max PFN 0x%x for MB_HANDLE: 0x%d\n", (ULONG)middle, (ULONG)g_MemoryBlockArray[i].Index);
            g_MemoryBlockArray[i].Size = middle+1;
            g_MemoryBlockArray[i].StartPage = PrevSize;
            g_MemoryBlockArray[i].EndPage = PrevSize + middle;
            wprintf(L"Block 0x%llx, StartPage = 0x%llx, EndPage = 0x%llx , Size = 0x%llx\n", i, g_MemoryBlockArray[i].StartPage, g_MemoryBlockArray[i].EndPage, g_MemoryB
            PrevSize += middle+1;
            break;
        }
    } //end while
```

We know, that memory block is continuous, therefore we can determine its boundary by setting the condition: when reading a block, the subsequent block shouldn't be read. Accordingly, first we can scan the memory and build the initial memory mapping scheme. But, as I wrote above, there are gaps between the blocks, and therefore, to clarify the memory allocation, we will have to examine the _PHYSICAL_MEMORY_DESCRIPTOR structure in guest OS.

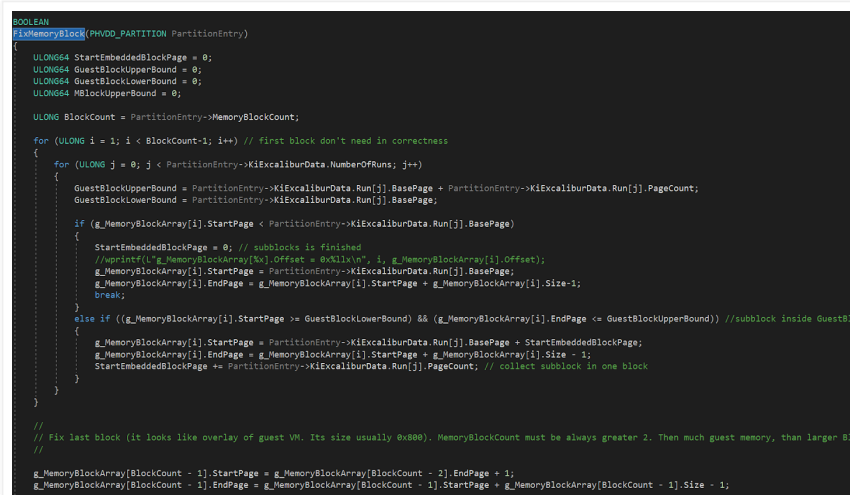
**0: kd> dt poi(nt!MmPhysicalMemoryBlock)  nt!_PHYSICAL_MEMORY_DESCRIPTOR**
+0x000 NumberOfRuns : 7
+0x008 NumberOfPages : 0xbfee1
+0x010 Run : [1] _PHYSICAL_MEMORY_RUN

**0: kd> dq poi(nt!MmPhysicalMemoryBlock) L20**
ffff8b81`91615020 00000000`00000007 00000000`000bfee1 – **all blocks count**, summary blocks size
ffff8b81`91615030 00000000`00000001 00000000`0000009f – **start position of block**, page count in block.
ffff8b81`91615040 00000000`00000100 00000000`0000027b
ffff8b81`91615050 00000000`0000037d 00000000`00005d86
ffff8b81`91615060 00000000`00006105 00000000`00058dc0
ffff8b81`91615070 00000000`0005ef1b 00000000`00001080
ffff8b81`91615080 00000000`0005ffff 00000000`00000001
ffff8b81`91615090 00000000`00060200 00000000`00060000

WinDBG has command to show PHYSICAL_MEMORY_DESCRIPTOR structure.




As you can see, part of the guest OS memory blocks fits in one block allocated by the hypervisor. And part of the blocks of the guest OS correspond to the blocks allocated by the hypervisor, with the same volume, but with some offset. Given that the offset is small, we can adjust our table:



```
BOOLEAN
FixMemoryBlock(PHVDD_PARTITION PartitionEntry)
{
    ULONG64 StartEmbeddedBlockPage = 0;
    ULONG64 GuestBlockUpperBound = 0;
    ULONG64 GuestBlockLowerBound = 0;
    ULONG64 MBlockUpperBound = 0;

    ULONG BlockCount = PartitionEntry->MemoryBlockCount;

    for (ULONG i = 1; i < BlockCount-1; i++) // first block don't need in correctness
    {
        for (ULONG j = 0; j < PartitionEntry->KiExcaliburData.NumberOfRuns; j++)
        {
            GuestBlockUpperBound = PartitionEntry->KiExcaliburData.Run[j].BasePage + PartitionEntry->KiExcaliburData.Run[j].PageCount;
            GuestBlockLowerBound = PartitionEntry->KiExcaliburData.Run[j].BasePage;

            if (g_MemoryBlockArray[i].StartPage < PartitionEntry->KiExcaliburData.Run[j].BasePage)
            {
                StartEmbeddedBlockPage = 0; // subblocks is finished
                //wprintf(L"g_MemoryBlockArray[%x].Offset = 0x%llx\n", i, g_MemoryBlockArray[i].Offset);
                g_MemoryBlockArray[i].StartPage = PartitionEntry->KiExcaliburData.Run[j].BasePage;
                g_MemoryBlockArray[i].EndPage = g_MemoryBlockArray[i].StartPage + g_MemoryBlockArray[i].Size-1;
                break;
            }
            else if ((g_MemoryBlockArray[i].StartPage >= GuestBlockLowerBound) && (g_MemoryBlockArray[i].EndPage <= GuestBlockUpperBound)) //subblock inside GuestB
            {
                g_MemoryBlockArray[i].StartPage = PartitionEntry->KiExcaliburData.Run[j].BasePage + StartEmbeddedBlockPage;
                g_MemoryBlockArray[i].EndPage = g_MemoryBlockArray[i].StartPage + g_MemoryBlockArray[i].Size - 1;
                StartEmbeddedBlockPage += PartitionEntry->KiExcaliburData.Run[j].PageCount; // collect subblock in one block
            }
        }
    }

    //
    // Fix last block (it looks like overlay of guest VM. Its size usually 0x800). MemoryBlockCount must be always greater 2. Then much guest memory, than larger B
    //

    g_MemoryBlockArray[BlockCount - 1].StartPage = g_MemoryBlockArray[BlockCount - 2].EndPage + 1;
    g_MemoryBlockArray[BlockCount - 1].EndPage = g_MemoryBlockArray[BlockCount - 1].StartPage + g_MemoryBlockArray[BlockCount - 1].Size - 1;
```

The first block isn't need for adjustment. Memory is mapping 1 in 1, which allows us to read data from the first block, where ntoskrnl.exe is located, in order to calculate the values ??of the _PHYSICAL_MEMORY_DESCRIPTOR structure later. After calculation, we can perform the offset correction. I described in driver code the case, when one guest block can consist of several blocks, allocated by the hypervisor,

but I haven't encountered such case in my stand. The last of the blocks with a size of 0x800 pages is used for video memory, as was explained above. In our case, in a virtual machine, the maximum physical address available for reading is greater than maximum address, specified in PHYSICAL_MEMORY_DESCRIPTOR. This block is not specified in PHYSICAL_MEMORY_DESCRIPTOR, so we just assume, that it goes sequentially after the last guest OS block. Offset of this block can't be determined without a driver in the host OS. We can assume, that this is memory used by the device, and it can be read, for example, by LiveCloudKd.

After correction, we can read all physical guest OS memory without the driver, excepting pages. Which was paged in pagefile.sys.
I complete code description on that point. The remaining details can be found in sources of hvmm driver.

**Additional details**

I wrote PyKD script ParsePrtnStructure.py for better visualization of GPAR objects and Mblock objects (link is given at the beginning of the article). For using it, you have to find partition handle first. To do this, run hvmm.sys driver, which outputs the value of this descriptor to the debugger and then inserted this value into the script.

Script output for Windows Server 2019 guest OS:

```
0: kd> !py @"F:\ida_files\ParsePrtnStructure.py"
Partition signature:  Prtn
Partition name:  Win2019-02
Partition id:  2
MBBlocks table address:  0xffffa8024f32bec0L
MBBlocks table element count:  2
Gpar block handle address:  0xffffa8024d426700L
Gpar Element Count:  3
pGparArray address:  0xffffa8024f0fd550L

GPAR Array content:
-------------------------------------------------------------------------------------------------------------------------
Index   Sign   StartPageNum  EndPageNum  UmFlag              MBlock              SomeGPA             VMMEM GPA
-------------------------------------------------------------------------------------------------------------------------
0       Gpar            0x0     0x8fbff       0   0xffffa80248f67d20L                 0x0                 0x0
1       Gpar        0xfec00     0xfec00       1       0x141f5c3c460L                  0x0                 0x0
2       Gpar       0xfff800    0xffffff       0   0xffffa8024f8bf920L                 0x0                 0x0

MBlock Array content:
-------------------------------------------------------------------------------------------------------------------------
Index   Sign  MBHandle    BitmapSize01    BitmapSize02            GPA Array
-------------------------------------------------------------------------------------------------------------------------
0       Mb           1         0x8fc00         0x8fc00   0xffffa8024fc00000L
1       Mb           2           0x800           0x800   0xffffa8024f8f7000L
0: kd> g
```

Count of GPAR and memory blocks for containers is much more:

```
0: kd> !py @"F:\ida_files\ParsePrtnStructure.py"
Partition signature:  Prtn
Partition name:  Virtual Machine
Partition id:  3
MBBlocks table address:  0xffff958b84cfd000L
MBBlocks table element count:  955
Gpar block handle address:  0xffff958b794c2c10L
Gpar Element Count:  956
pGparArray address:  0xffff958b7bcf9000L

GPAR Array content:
```

| Index | Sign | StartPageNum | EndPageNum | MemoryBlockGpaRangeFlag | MBlock | SomeGPA offset | Vmmem |
|---|---|---|---|---|---|---|---|
| 0 | Gpar | 0x0 | 0x3ffff | 0 | 0xffff958b71f2f4a0L | 0x0 | 0x19 |
| 1 | Gpar | 0x40000 | 0x7ffff | 0 | 0xffff958b74cd2010L | 0x0 | 0x19 |
| 2 | Gpar | 0x200000 | 0x201fff | 0 | 0xffff958b72344010L | 0x0 | 0x19 |
| 3 | Gpar | 0xfec00 | 0xfec00 | 1 | 0x23e91cfba40L | 0x0 | |
| 4 | Gpar | 0x100000 | 0x10001c | 0 | 0xffff958b71c9d620L | 0x0 | 0x19 |
| 5 | Gpar | 0x10001d | 0x10020c | 0 | 0xffff958b71d986a0L | 0x0 | 0x19 |
| 6 | Gpar | 0x10020d | 0x1003a6 | 0 | 0xffff958b734686a0L | 0x0 | 0x19 |
| 7 | Gpar | 0x1003a7 | 0x1003d7 | 0 | 0xffff958b71cf66a0L | 0x0 | 0x19 |
| 8 | Gpar | 0x1003d8 | 0x100407 | 0 | 0xffff958b71d866a0L | 0x0 | 0x19 |
| 9 | Gpar | 0x100408 | 0x10041c | 0 | 0xffff958b71c686a0L | 0x0 | 0x19 |
| 10 | Gpar | 0x10041d | 0x10042a | 0 | 0xffff958b720bd620L | 0x0 | 0x19 |
| 11 | Gpar | 0x10042b | 0x100434 | 0 | 0xffff958b72027620L | 0x0 | 0x19 |
| 12 | Gpar | 0x100435 | 0x10043e | 0 | 0xffff958b86cd3c00L | 0x0 | 0x19 |
| 13 | Gpar | 0x10043f | 0x1007b1 | 0 | 0xffff958b738978d0L | 0x0 | 0x19 |
| 14 | Gpar | 0x1007b2 | 0x1007c7 | 0 | 0xffff958b71c17c00L | 0x0 | 0x19 |
| 15 | Gpar | 0x1007c8 | 0x1007dd | 0 | 0xffff958b861d3970L | 0x0 | 0x19 |
| 16 | Gpar | 0x1007de | 0x1007ee | 0 | 0xffff958b74af1cb0L | 0x0 | 0x19 |

```
MBlock Array content:
```

| Index | Sign | MBHandle | BitmapSize01 | BitmapSize02 | GP |
|---|---|---|---|---|---|
| 0 | Mb | 1 | 0x40000 | 0x40000 | 0xffff958b6d |
| 1 | Mb | 2 | 0x1d | 0x1d | 0xffff958b73 |
| 2 | Mb | 3 | 0x1f0 | 0x1f0 | 0xffff958b73 |
| 3 | Mb | 4 | 0x19a | 0x19a | 0xffff958b75 |
| 4 | Mb | 5 | 0x31 | 0x31 | 0xffff958b72 |
| 5 | Mb | 6 | 0x30 | 0x30 | 0xffff958b73 |
| 6 | Mb | 7 | 0x15 | 0x15 | 0xffff958b74 |
| 7 | Mb | 8 | 0xe | 0xe | 0xffff958b8c |
| 8 | Mb | 9 | 0xa | 0xa | 0xffff958b80 |
| 9 | Mb | 10 | 0xa | 0xa | 0xffff958b80 |
| 10 | Mb | 11 | 0x373 | 0x373 | 0xffff958b73 |
| 11 | Mb | 12 | 0x16 | 0x16 | 0xffff958b87 |
| 12 | Mb | 13 | 0x16 | 0x16 | 0xffff958b87 |
| 13 | Mb | 14 | 0x11 | 0x11 | 0xffff958b72 |
| 14 | Mb | 15 | 0x1c | 0x1c | 0xffff958b72 |
| 15 | Mb | 16 | 0x11 | 0x11 | 0xffff958b72 |
| 16 | Mb | 17 | 0x26 | 0x26 | 0xffff958b71 |
| 17 | Mb | 18 | 0x59 | 0x59 | 0xffff958b6f |
| 18 | Mb | 19 | 0x13 | 0x13 | 0xffff958b87 |
| 19 | Mb | 20 | 0x1a | 0x1a | 0xffff958b87 |
| 20 | Mb | 21 | 0x2b | 0x2b | 0xffff958b83 |
| 21 | Mb | 22 | 0x14 | 0x14 | 0xffff958b73 |
| 22 | Mb | 23 | 0x40 | 0x40 | 0xffff958b75 |

In Hyper-V containers all Mblock objects contains zero. Like this:

**0: kd> dc 0xffff958b7f0d14d0**

ffff958b`7f0d14d0  00000000 00000000 00000000 00000000  ................

ffff958b`7f0d14e0  00000000 00000000 00000000 00000000  ................

ffff958b`7f0d14f0  00000000 00000000 00000000 00000000  ................

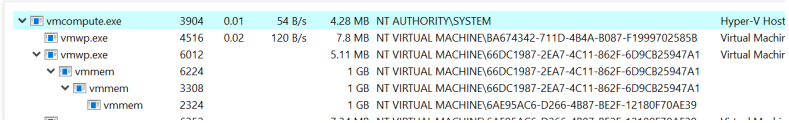
there is additional type of block inside vid.sys driver: reserve bucket block (VSMM_RESERVE_BUCKET)

```
0: kd> dc 0xffffa8024f417000+0x1390
ffffa802`4f418390  20206252 00000000 00000000 00000000
ffffa802`4f4183a0  4f4183a0 ffffa802 4f4183a0 ffffa802
ffffa802`4f4183b0  4f4183b0 ffffa802 4f4183b0 ffffa802
ffffa802`4f4183c0  4f4183c0 ffffa802 4f4183c0 ffffa802
ffffa802`4f4183d0  4f4183d0 ffffa802 4f4183d0 ffffa802
ffffa802`4f4183e0  4f4183e0 ffffa802 4f4183e0 ffffa802
ffffa802`4f4183f0  4f4183f0 ffffa802 4f4183f0 ffffa802
```

But it is not need for reading guest OS memory in standard case. We see that address is pointing to themselves (0x10 alignment).

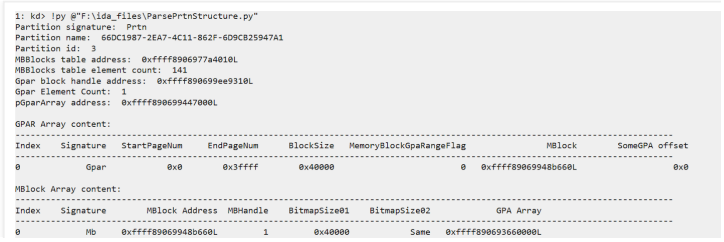
**Docker container with Hyper-V isolation mode**

Docker container in Hyper-V isolation mode creates quite a lot of processes (processes for 1 Windows Server 2019 nanoserver 1809 container):



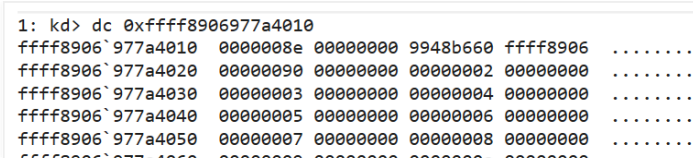
| | | | | | | |
|---|---|---|---|---|---|---|
| vmcompute.exe | 3904 | 0.01 | 54 B/s | 4.28 MB | NT AUTHORITY\SYSTEM | Hyper-V Host |
| vmwp.exe | 4516 | 0.02 | 120 B/s | 7.8 MB | NT VIRTUAL MACHINE\BA674342-711D-4B4A-B087-F1999702585B | Virtual Machir |
| vmwp.exe | 6012 | | | 5.11 MB | NT VIRTUAL MACHINE\66DC1987-2EA7-4C11-862F-6D9CB25947A1 | Virtual Machir |
| vmmem | 6224 | | | 1 GB | NT VIRTUAL MACHINE\66DC1987-2EA7-4C11-862F-6D9CB25947A1 | |
| vmmem | 3308 | | | 1 GB | NT VIRTUAL MACHINE\66DC1987-2EA7-4C11-862F-6D9CB25947A1 | |
| vmmem | 2324 | | | 1 GB | NT VIRTUAL MACHINE\6AE95AC6-D266-4B87-BE2F-12180F70AE39 | |

We see 2 partition handles (by the count of vmwp.exe processes). The name of 1st of them matches the name of the user in the context of which the process is running.



```
1: kd> !py @"F:\ida_files\ParsePrtnStructure.py"
Partition signature:  Prtn
Partition name:  66DC1987-2EA7-4C11-862F-6D9CB25947A1
Partition id:  3
MBBlocks table address:  0xffff8906977a4010L
MBBlocks table element count:  141
Gpar block handle address:  0xffff890699ee9310L
Gpar Element Count:  1
pGparArray address:  0xffff890699447000L

GPAR Array content:
-------------------------------------------------------------------------------------------------------------------------------
Index    Signature   StartPageNum     EndPageNum      BlockSize   MemoryBlockGpaRangeFlag                MBlock        SomeGPA offset
-------------------------------------------------------------------------------------------------------------------------------
0            Gpar            0x0        0x3ffff        0x40000                        0   0xffff89069948b660L                 0x0

MBlock Array content:
-------------------------------------------------------------------------------------------------------------------------------
Index    Signature      MBlock Address  MBHandle    BitmapSize01    BitmapSize02            GPA Array
-------------------------------------------------------------------------------------------------------------------------------
0            Mb    0xffff89069948b660L        1        0x40000           Same   0xffff890693660000L
```

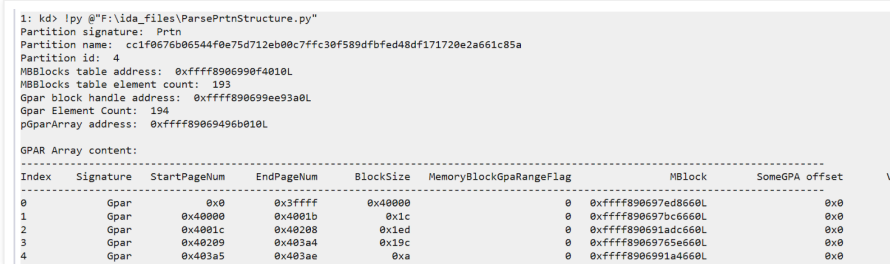
However, this partition has irrelevant table of MBlock objects:



```
1: kd> dc 0xffff8906977a4010
ffff8906`977a4010  0000008e 00000000 9948b660 ffff8906  ........
ffff8906`977a4020  00000090 00000000 00000002 00000000  ........
ffff8906`977a4030  00000003 00000000 00000004 00000000  ........
ffff8906`977a4040  00000005 00000000 00000006 00000000  ........
ffff8906`977a4050  00000007 00000000 00000008 00000000  ........
```

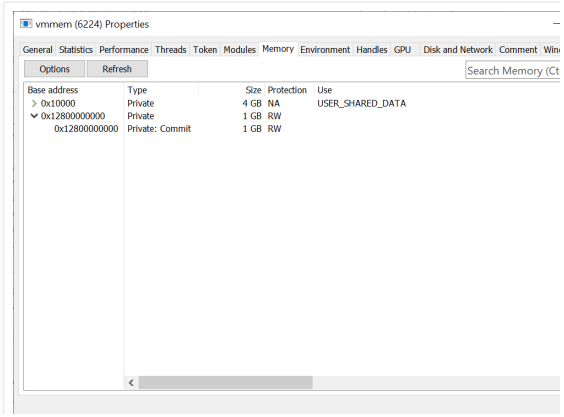
Elements count is 0x8e, but the MBlock object itself is only one, and it is empty.

Name of 2nd partition coincides with the identifier, created for container, and contains necessary Nt-kernel data, that can be used to access the memory of the container using WinDBG.

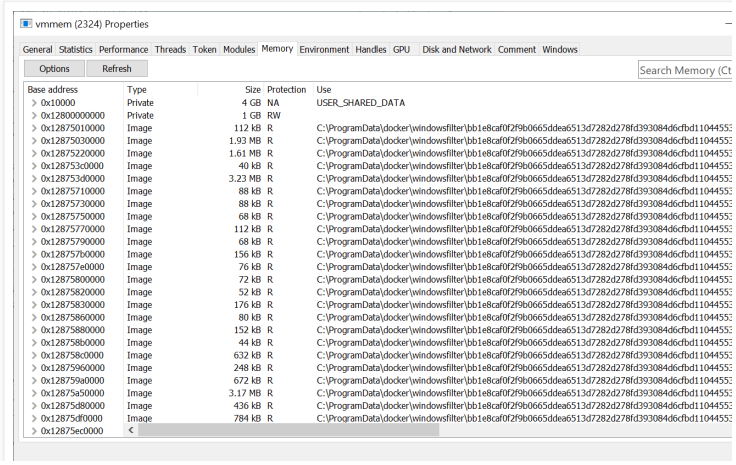


```
1: kd> !py @"F:\ida_files\ParsePrtnStructure.py"
Partition signature:  Prtn
Partition name:  cc1f0676b06544f0e75d712eb00c7ffc30f589dfbfed48df171720e2a661c85a
Partition id:  4
MBBlocks table address:  0xffff8906990f4010L
MBBlocks table element count:  193
Gpar block handle address:  0xffff890699ee93a0L
Gpar Element Count:  194
pGparArray address:  0xffff89069496b010L

GPAR Array content:
-------------------------------------------------------------------------------------------------------------------------------
Index    Signature   StartPageNum     EndPageNum      BlockSize   MemoryBlockGpaRangeFlag                MBlock        SomeGPA offset
-------------------------------------------------------------------------------------------------------------------------------
0            Gpar            0x0        0x3ffff        0x40000                        0   0xffff890697ed8660L                 0x0
1            Gpar        0x40000        0x4001b           0x1c                        0   0xffff890697bc6660L                 0x0
2            Gpar        0x4001c        0x40208          0x1ed                        0   0xffff890691adc660L                 0x0
3            Gpar        0x40209        0x403a4          0x19c                        0   0xffff89069765e660L                 0x0
4            Gpar        0x403a5        0x403ae            0xa                        0   0xffff8906991a4660L                 0x0
```

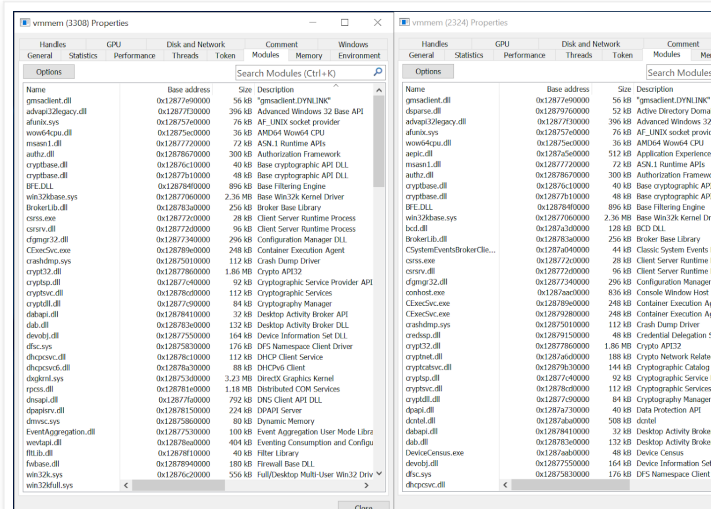
Base address is the same as the Vmmem GPA Offset parameter, which is used for reading memory block from the context of the vmmem process.




The offset of file mapping region in another vmmem instance are the same as VmmemGPA offset, using by hvmm.sys driver.




Different vmmem processes load different executables. But in the process, where there are fewer files, the number of active threads is 0.




```
1: kd> !process 0 0 vmmem
PROCESS ffff890697eab080
    SessionId: 0  Cid: 1850    Peb: 00000000  ParentCid: 177c
    DirBase: 119d00002  ObjectTable: ffffe707a7e384c0  HandleC
    Image: vmmem

PROCESS ffff890694fa90c0
    SessionId: 0  Cid: 0cec    Peb: 00000000  ParentCid: 1850
    DirBase: 37d00002  ObjectTable: ffffe707a54a81c0  HandleCo
    Image: vmmem

PROCESS ffff890697ac7300
    SessionId: 0  Cid: 0914    Peb: 00000000  ParentCid: 0cec
    DirBase: 133a00002  ObjectTable: ffffe707a54a9c80  HandleC
    Image: vmmem
```

The 2nd process of the vmmem docker container is not critical to execution. It can be killed through Process Hacker (the memory size will be several tens of kilobytes). The 1st vmmem process is also not critical for reading memory. The registers of the section to which the process is attached have the correct values, but when reading the kernel mode memory, zeros are returned.

After stopping the two aforementioned vmmem processes, you can still safely start processes inside the container through docker exec.

Call stack of vmmem creation (3 times per container starting process)

| 1st PsCreateMinimalProcess | 2nd PsCreateMinimalProcess | 3rd PsCreateMinimalProcess |
|---|---|---|
| **: kd> kcn** | **2: kd> kcn** | **0: kd> kcn** |
| # Call Site | # Call Site | # Call Site |
| 00 nt!PsCreateMinimalProcess | 00 nt!PsCreateMinimalProcess | 00 nt!PsCreateMinimalProcess |
| 01 nt!VmCreateMemoryProcess | 01 nt!VmCreateMemoryProcess | 01 nt!VmCreateMemoryProcess |
| 02 Vid!VsmmNtSlatMemoryProcessCreate | 02 Vid!VsmmNtSlatMemoryProcessCreate | 02 Vid!VsmmNtSlatMemoryProcessCreate |
| 03 **Vid!VsmmProcesspMicroVmSetup** | **03 Vid!VsmmClonepTemplateCreate** | **03 Vid!VsmmCloneTemplateApply** |
| …… | …… | …… |
| 14 vmwp!VidPartitionManager::Initialize | 13 vmwp!WorkerTaskSaving::StartSave | 13 vmwp!VidPartitionManager::Initialize |
| 15 vmwp!VidPartitionManager::CreateInstance | 14 vmwp!WorkerTaskSaving::RunSaveSteps | 14 vmwp!VidPartitionManager::CreateInstance |

| | | |
|---|---|---|
| | 15 vmwp!WorkerTaskSaving::RunTask | |

We again see a pseudo Gpar object pointing to a user mode structure (as seen above, this block is created for interaction with virtual devices):

```
188          Gpar       0x451e9          0x45217          0x2f                0   0xffff8906990f7d80L                0x0        0
189          Gpar       0x45218          0x4523f          0x28                0   0xffff890694ebe010L                0x0        0
190          Gpar       0x45240          0x45249           0xa                0   0xffff890694eee9d0L                0x0        0
191          Gpar       0x4524a          0x45570         0x327                0   0xffff89069817eb60L                0x0        0
192          Gpar       0x45571          0x455a8          0x38                0   0xffff890694aabd60L                0x0        0
193          Gpar       0xfec00          0xfec00           0x1                1         0x22f69043d20L               0x0
```

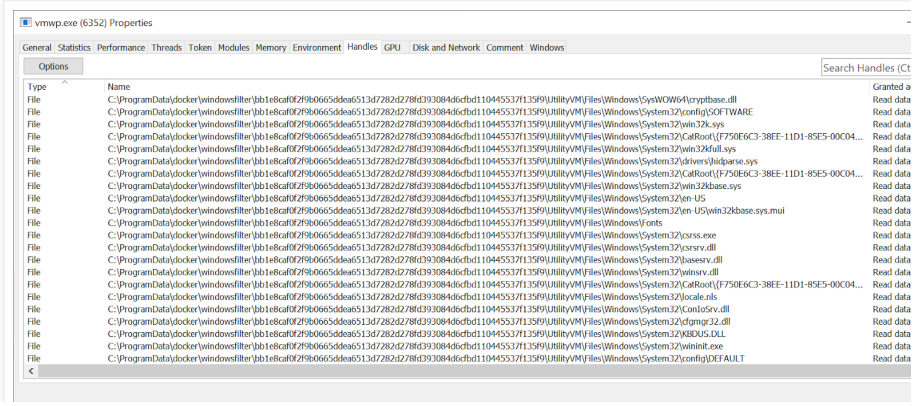
For reading memory inside this block we have to enter vmwp.exe context:

```
1: kd> !process 0 0 vmwp.exe
PROCESS ffff890697eda080
    SessionId: 0  Cid: 11a4    Peb: d3895ed000  ParentCid: 0f40
    DirBase: 133dd0002  ObjectTable: ffffe7079f786280  HandleCount: 427.
    Image: vmwp.exe

PROCESS ffff8906981ec080
    SessionId: 0  Cid: 177c    Peb: 6b3af21000  ParentCid: 0f40
    DirBase: 120d00002  ObjectTable: ffffe707a7388d00  HandleCount: 273.
    Image: vmwp.exe

PROCESS ffff890691b40080
    SessionId: 0  Cid: 18d0    Peb: 8752dd5000  ParentCid: 0f40
    DirBase: 12af00002  ObjectTable: ffffe707a54a8740  HandleCount: 981.
    Image: vmwp.exe

1: kd> .process ffff890691b40080
Implicit process is now ffff8906`91b40080
WARNING: .cache forcedecodeuser is not enabled
1: kd> .reload
Connected to Windows 10 17763 x64 target at (Tue Sep  3 09:06:53.768 2019 (UTC + 3:00)
Loading Kernel Symbols
...............................................................
................................................................
..............................
Loading User Symbols
...............................................................
Loading unloaded module list
........
1: kd> dps 0x22f69043d20
0000022f`69043d20  00007ff6`14e6a458 vmwp!VND_HANDLER_CONTEXT::`vftable'
0000022f`69043d28  00000000`00000000
0000022f`69043d30  01000001`07000001
0000022f`69043d38  00000000`00000000
0000022f`69043d40  00000000`00000000
0000022f`69043d48  00000000`00000000
0000022f`69043d50  0000022f`68ec5a60
0000022f`69043d58  0000022f`68ec6140
0000022f`69043d60  00000000`00000000
0000022f`69043d68  00000000`00000000
0000022f`69043d70  00007ff6`14e6a418 vmwp!VND_HANDLER_CONTEXT::`vftable'
0000022f`69043d78  00007ff6`14e6a3c0 vmwp!VND_HANDLER_CONTEXT::`vftable'
0000022f`69043d80  00007ff6`14e6a368 vmwp!VND_HANDLER_CONTEXT::`vftable'
0000022f`69043d88  00000000`00000000
0000022f`69043d90  00000000`00000090
```

```
1: kd> dps 00007ff6`14e6a458
00007ff6`14e6a458  00007ff6`14d05a10 vmwp!VND_HANDLER_CONTEXT::`vector deleting destructor'
00007ff6`14e6a460  00007ff6`14d08b70 vmwp!VND_HANDLER_CONTEXT::PrepareSelf
00007ff6`14e6a468  00007ff6`14d3ea00 vmwp!VND_HANDLER_CONTEXT::UnprepareSelf
00007ff6`14e6a470  00007ff6`14d3dbd0 vmwp!Vml::VmSharableObject::QuiesceSelf
00007ff6`14e6a478  00007ff6`14d0e110 vmwp!Vml::VmConnectionPointContainer<ComVirtualMachine>::~VmConnectionPointContainer<Cc
00007ff6`14e6a480  00007ff6`14d0e110 vmwp!Vml::VmConnectionPointContainer<ComVirtualMachine>::~VmConnectionPointContainer<Cc
00007ff6`14e6a488  00007ff6`14e9a5d0 vmwp!Vml::VmAutoLock::`RTTI Complete Object Locator'
00007ff6`14e6a490  00007ff6`14d7ade0 vmwp!Vml::VmAutoLock::`vector deleting destructor'
00007ff6`14e6a498  00007ff6`14e99590 vmwp!VmbComServiceAccess::`RTTI Complete Object Locator'
00007ff6`14e6a4a0  00007ff6`14d4cea0 vmwp!VmbComServiceAccess::`vector deleting destructor'
00007ff6`14e6a4a8  00007ff6`14d0e110 vmwp!Vml::VmConnectionPointContainer<ComVirtualMachine>::~VmConnectionPointContainer<Cc
00007ff6`14e6a4b0  00007ff6`14d0e110 vmwp!Vml::VmConnectionPointContainer<ComVirtualMachine>::~VmConnectionPointContainer<Cc
00007ff6`14e6a4b8  00007ff6`14d3dbd0 vmwp!Vml::VmSharableObject::QuiesceSelf
00007ff6`14e6a4c0  00007ff6`14d0e110 vmwp!Vml::VmConnectionPointContainer<ComVirtualMachine>::~VmConnectionPointContainer<Cc
00007ff6`14e6a4c8  00007ff6`14d0e110 vmwp!Vml::VmConnectionPointContainer<ComVirtualMachine>::~VmConnectionPointContainer<Cc
```

```
1: kd> dps 00007ff6`14e6a418
00007ff6`14e6a418  00007ff6`14d09750 vmwp!VND_HANDLER_CONTEXT::AddReference
00007ff6`14e6a420  00007ff6`14d09350 vmwp!VND_HANDLER_CONTEXT::RemoveReference
00007ff6`14e6a428  00007ff6`14d499e0 vmwp!VND_HANDLER_CONTEXT::GetCallbackBatch
00007ff6`14e6a430  00007ff6`14d494d0 vmwp!Vml::VmComLocalMemStream::GetBufferOffset
00007ff6`14e6a438  00007ff6`14d499f0 vmwp!ProcessorManager::GetVirtualProcessorCount
00007ff6`14e6a440  00007ff6`14d49b40 vmwp!ProcessorManager::GetProcessorOvercommitAllowed
00007ff6`14e6a448  00007ff6`14d49ab0 vmwp!ProcessorManager::GetCpuGroupId
00007ff6`14e6a450  00007ff6`14e99450 vmwp!VND_HANDLER_CONTEXT::`RTTI Complete Object Locator'
00007ff6`14e6a458  00007ff6`14d05a10 vmwp!VND_HANDLER_CONTEXT::`vector deleting destructor'
00007ff6`14e6a460  00007ff6`14d08b70 vmwp!VND_HANDLER_CONTEXT::PrepareSelf
00007ff6`14e6a468  00007ff6`14d3ea00 vmwp!VND_HANDLER_CONTEXT::UnprepareSelf
00007ff6`14e6a470  00007ff6`14d3dbd0 vmwp!Vml::VmSharableObject::QuiesceSelf
00007ff6`14e6a478  00007ff6`14d0e110 vmwp!Vml::VmConnectionPointContainer<ComVirtualMachine>::~VmConnectionPointContainer<ComVi
00007ff6`14e6a480  00007ff6`14d0e110 vmwp!Vml::VmConnectionPointContainer<ComVirtualMachine>::~VmConnectionPointContainer<ComVi
00007ff6`14e6a488  00007ff6`14e9a5d0 vmwp!Vml::VmAutoLock::`RTTI Complete Object Locator'
```

Vmwp.exe process of docker container contain descriptor of files, that used inside container:



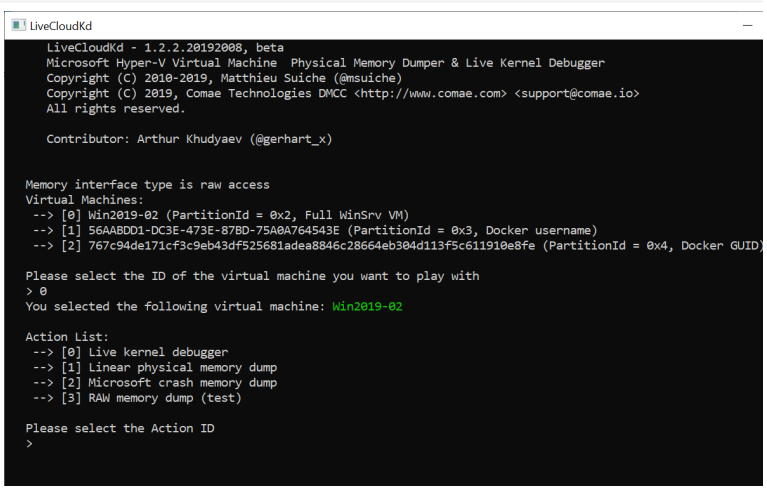

More information about docker containers internals you can see in video from Microsoft Ignite conference:
https://www.youtube.com/watch?time_continue=2291&v=tG8R5SQGPck (OS internals: Technical deep-dive into operating system innovations - BRK3365, starting from 38:11).

**Usage examples**

In which programs can we use the ability to read/write memory to the guest OS?

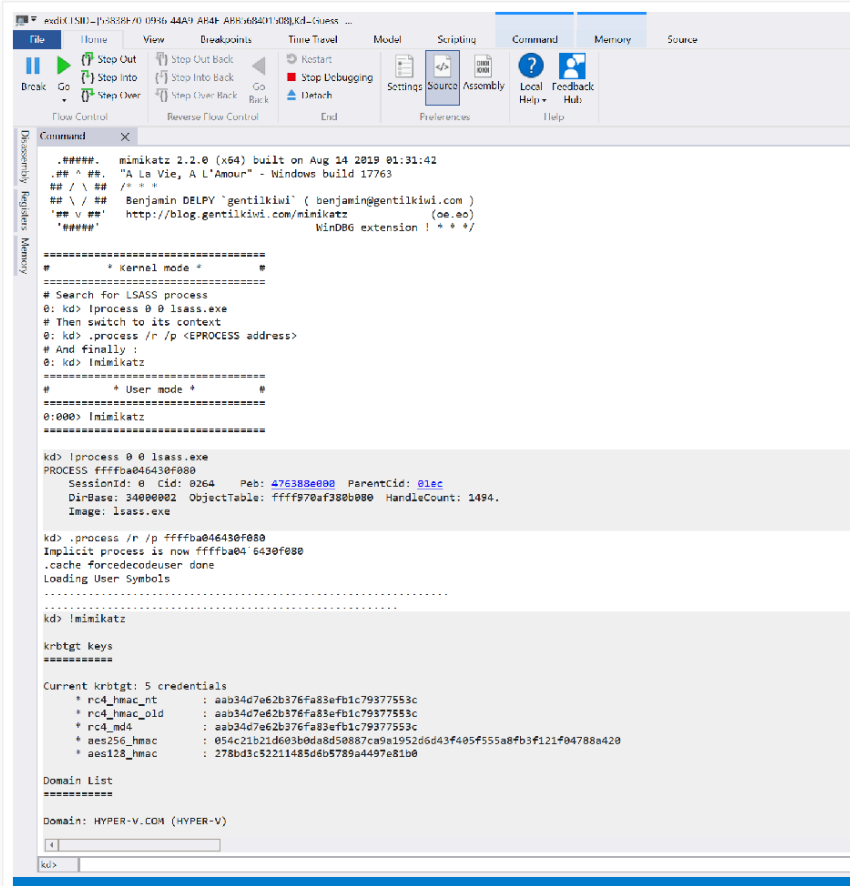
LiveCloudKd (as an alternative to Sysinternals LiveKd in the -hvl option part).
On screenshot, one Full VM with Windows Server 2019 and 1 Docker container in Hyper-V isolation mode are running on Hyper-V host server.

https://github.com/gerhart01/LiveCloudKd/releases



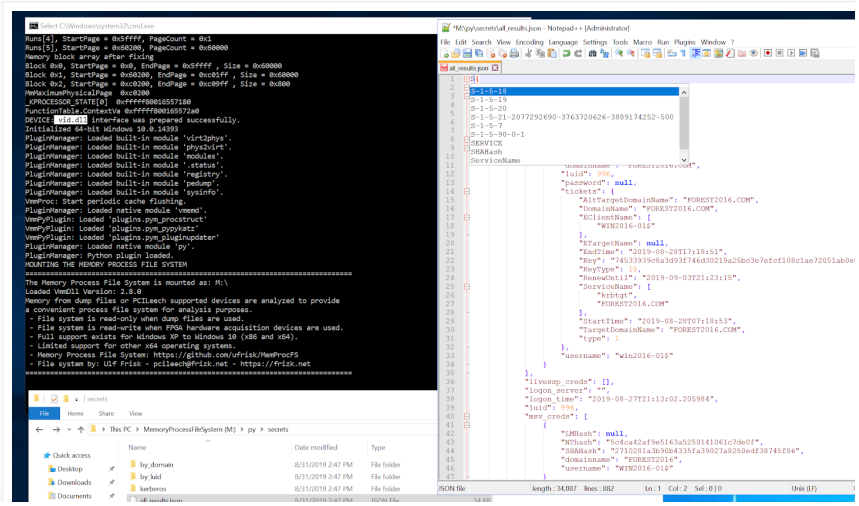

EXDi-plugin for WinDBG - the options are the same, but allows you to use legal functions for WinDBG integration (LiveCloudKd uses hooks of some functions inside WinDBG). It even works with WinDBG Preview, which itself runs in a separate container (UWP application). At the time of writing, EXDi-plugin plugin only works with Windows Server 2019\Windows 10 with the hvmm.sys driver loaded, since it requires a write operation to the guest OS. The screenshot shows the operation of WinDBG Preview in EXDi mode and the mimilb.dll plugin, which is part of the mimikatz utility.

https://github.com/gerhart01/LiveCloudKd/tree/master/ExdiKdSample




The plugin for the MemProcFs program (https://github.com/ufrisk/MemProcFS), which is integrated with pypykatz (https://github.com/skelsec/pypykatz) also allows you to scan the guest OS for hashes (in the screenshot, guest OS - domain controller, based on Windows Server 2016).

https://github.com/gerhart01/LiveCloudKd/tree/master/LeechCore



It is clear, that for using this method you need get access the host server with administrator rights. So, first of all, I position the utility as an opportunity to dig inside the OS when the debugger is long configured\too lazy or unable to connect (for example, the Secure Boot option is active).

**Conclusion**

The article described various ways to accessing memory of Hyper-V guest partitions, created in a variety of cases. I hope that working with Hyper-V memory has become a little more understandable. Hyper-V evolves very quickly and integrates more and more actively into the Windows kernel, while remaining virtually undocumented.

The information may be useful to those who want to understand the internal structure of Hyper-V, and possibly get transparent access to the guest OS memory, as well as make its modification. For LiveCloudKd usage it is necessary to have access to the root OS, where the virtual machines are located, and I don't think that it carries any security risk. However, for Windows Server 2016 such access can be obtained using only the user mode API, which is rather problematic to control. For protection, it is recommended to enable either the Shielded VM option (then, to bypass it, you will need to load the driver), or use Windows Server 2019, where Microsoft blocked the API call from vid.dll for third-party processes and turned on for vmwp.exe the prohibition of injecting libraries, that not signed by Microsoft. However, the latest work on introducing code into third-party processes, demonstrated in August 2019 at Blackhat in Las Vegas (report by Process Injection Techniques - Gotta Catch Them All from Itzik Kotler and Amit Klein from SafeBreach Labs), shows that there are ways to get around these restrictions from user mode (of course, this requires local administrator rights). The only reliable protection against such access to guest OS is Microsoft's Code Integrity in conjunction with the Shielded VM.

Posted by Gerhart X at 2:52 PM

## No comments:

## Post a Comment

Enter your comment...

Comment as: Google Accoun ▾

Publish Preview

Home Older Post

Subscribe to: Post Comments (Atom)

Simple theme. Powered by Blogger.