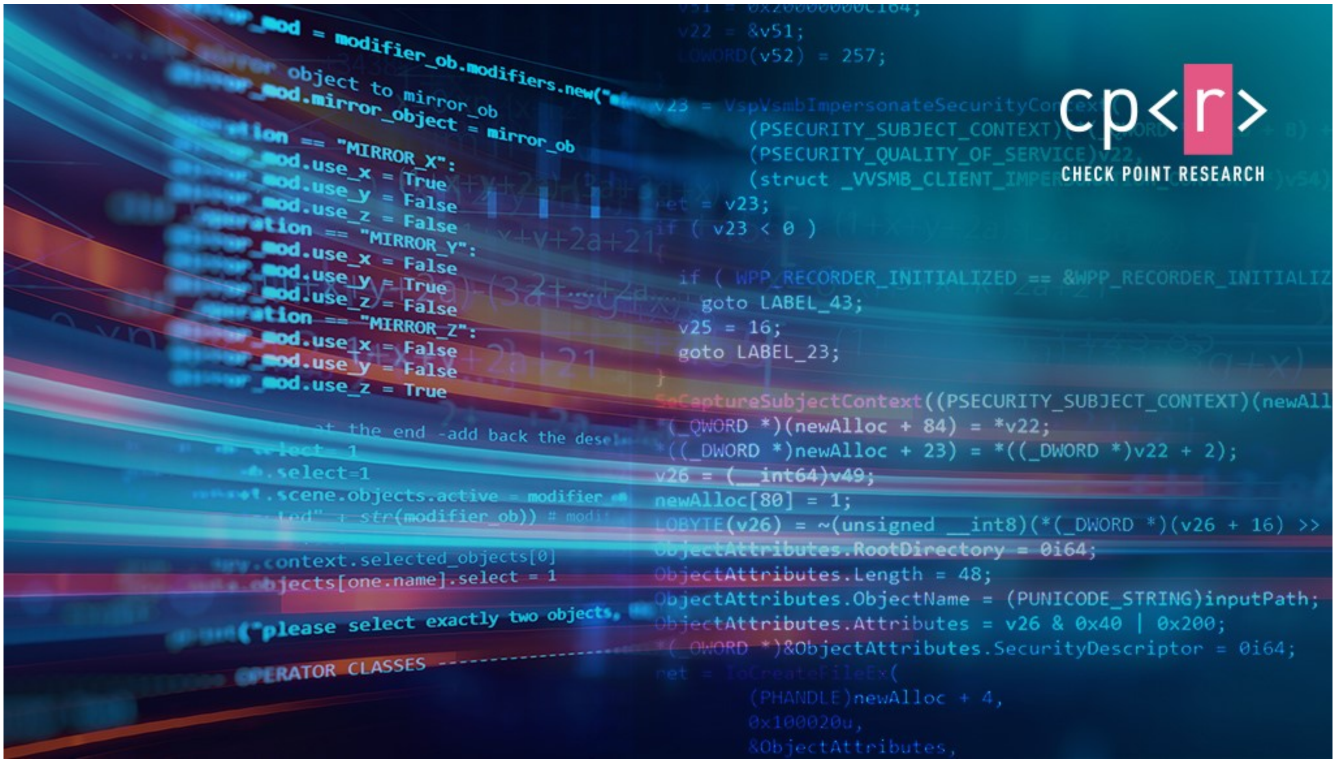
Playing in the (Windows) Sandbox

research.checkpoint.com/2021/playing-in-the-windows-sandbox/

March 11, 2021



March 11, 2021
Research By: Alex Ilgayev

Introduction

Two years ago, Microsoft released a new feature as a part of the **Insiders build 18305** – **Windows Sandbox.**

This sandbox has some useful specifications:

- Integrated part of Windows 10 (Pro/Enterprise).
- Runs on top of Hyper-V virtualization.
- Pristine and disposable – Starts clean on each run and has no persistent state.
- Configurable through a configuration file that has a dedicated format (WSB format). You can configure networking, vGPU, mapped folders, an automated script to run at user login, and many other options.
- The deployment is based on Windows Containers technology.

Judging by the accompanying technical blog post, we can say that Microsoft achieved a major technical milestone. The resulting sandbox presents the best of both worlds: on the one hand, the sandbox is based on Hyper-V technology, which means it inherits Hyper-V's strict virtualization security. On the other hand, the sandbox contains several features which allow sharing resources with the host machine to reduce CPU and memory consumption.

One of the interesting features is of particular importance, and we will elaborate on it here.

Dynamically Generated Image

The guest disk and filesystem are created dynamically, and are implemented using files in the host filesystem.

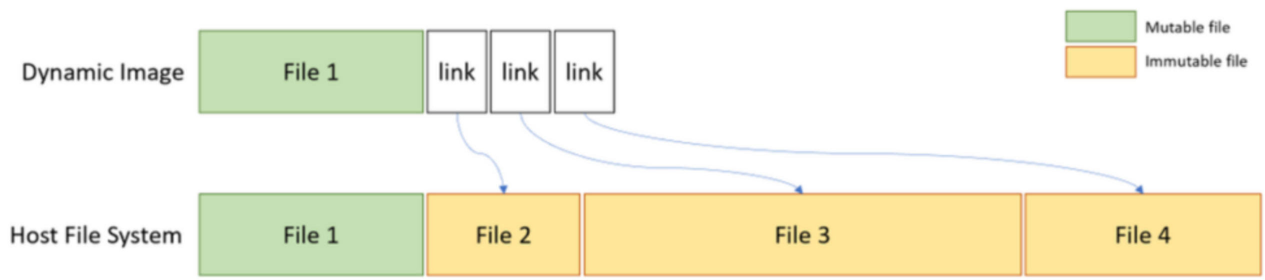


Figure 1 – Dynamically generated image (from Microsoft official documentation).

We decided to dig deeper into this technology for several reasons.

- **Lack of documentation** on its internal technicalities, both official and community-based. While it combines two widely documented technologies (Windows Containers and Hyper-V), we are still missing on how it all works together. For example, the technical blog refers to the Windows Containers technology, but in the official documentation, the creation and management of Windows Containers is done using the *Docker* utility for Windows, which isn't used in Windows Sandbox.
- Unfortunately, Microsoft does not allow any customization to the sandbox other than tweaking the WSB file. This means we can't install any program that requires a reboot, or create our own base image for the sandbox.

In this article, we break down several of the components, execution flow, driver support, and the implementation design of the dynamic image feature. We show that several internal technologies are involved, such as NTFS custom reparse tag, VHDx layering, container configuration for proper isolation, virtual storage drivers, vSMB over VMBus, and more. We also create a custom FLARE VM sandbox for malware analysis purposes, whose startup time is just 10 seconds.

General Components

The complex ecosystem of Hyper-V and its modules has already been researched extensively. Several vulnerabilities were found, such as the next *VmSwitch* RCE which can cause a full guest-to-host escape. A few years ago, Microsoft introduced Windows Containers (mainly for servers), a feature which allowed running Docker natively on Windows to ease software deployment.

Both these technologies were also introduced to the Windows 10 endpoint platform in the form of two components: ***WDAG*** (Windows Defender Application Guard), and most recently, **Windows Sandbox**. Lately, WDAG and another exciting feature for Office isolation were combined as MDAG – Microsoft Defender Application Guard.

In the *POC2018* conference, Yunhai Zhang had a presentation where he dived into the WDAG architecture and internals. As we demonstrate, Windows Sandbox shares the same technologies for its underlying implementation.

The sandbox can be divided into three components: two services – `CmService.dll` and `vmcompute.exe` – and the created worker process, `vmwp.exe` .

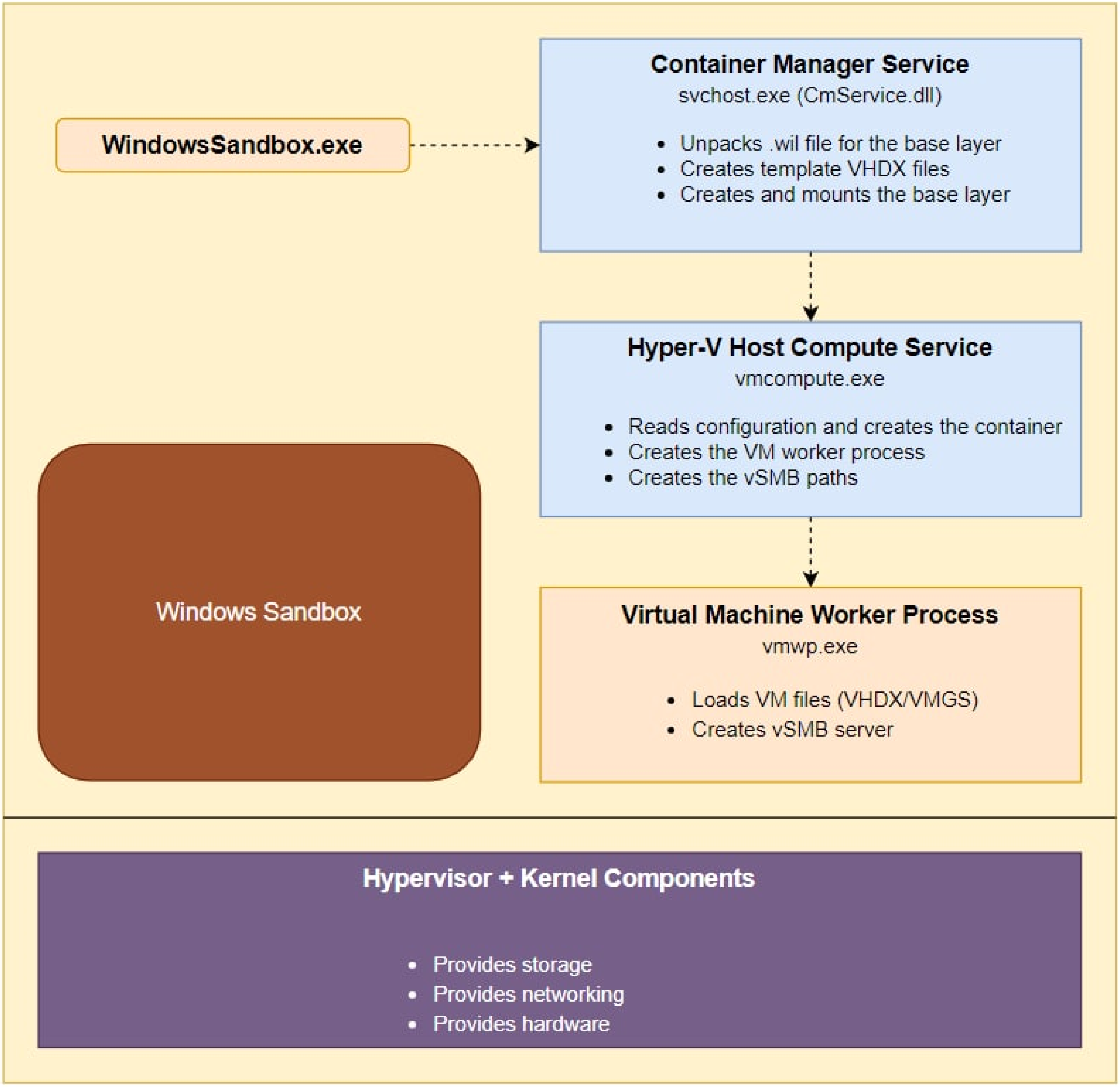


Figure 2 – Windows Sandbox general components.

Preparing the Sandbox

Behind every Hyper-V based VM there is a *VHDx* file, a virtual disk which is used by the machine. To understand how the disk is created, we looked at the working folder of an actively running sandbox: `%PROGRAMDATA%\Microsoft\Windows\Containers` . Surprisingly, we found more than 8 VHDx files.

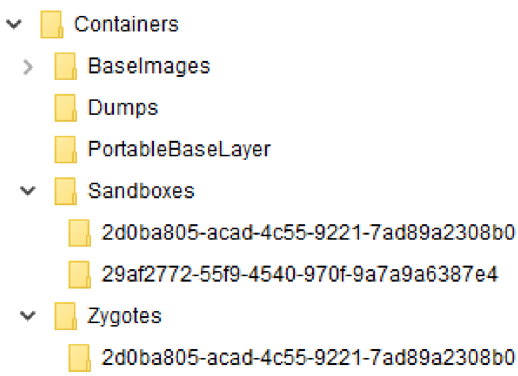


Figure 3 – Working folder structure.

We can track the main VHDx file by its dynamic size at the next path – `Sandboxes\29af2772-55f9-4540-970f-9a7a9a6387e4\sandbox.vhdx` , where the GUID is randomly generated on each sandbox run.

When we manually mount the VHDx file, we see that most of its filesystem is missing (this phenomenon is also visible in Zhang's WDAG research, mentioned previously).

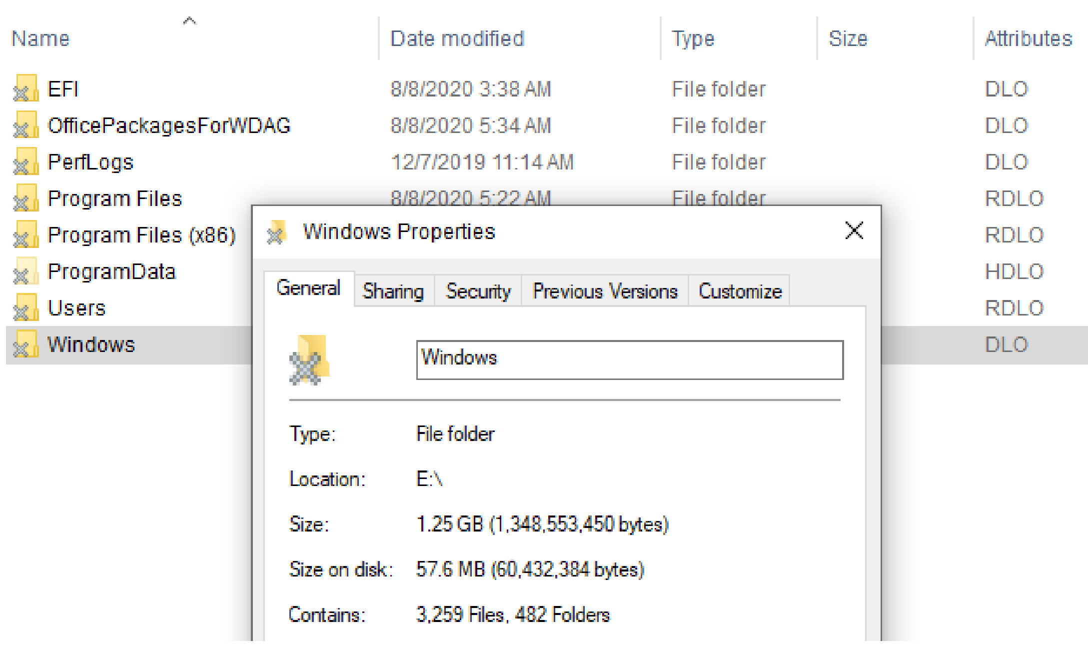


Figure 4 – Mounted sandbox VHDx.

We can immediately observe the "X" sign on the folder icon. If we turn on the "attributes" column in File Explorer, we can see two unusual NTFS attributes. These are explained here:

O – Offline

L – Reparse Point

Reparse Point is an extension to NTFS which allows it to create a "link" to another path. It also plays a role in other features, such as volume mounting. In our case, it makes sense that this feature is used as most of the files aren't "physically" present in the VHDx file.

To understand where the reparse points to and what's there, we delve deeper into the NTFS structure.

Parsing MFT Record

The *Master File Table* (MFT) stores the information required to retrieve files from an NTFS partition. A file may have one or more MFT records, and can contain one or more attributes. We can run the popular forensic tool, *Volatility*, with the `mftparser` option to parse all MFT records in the underlying file system. This can be done using the following command line:

```
volatility.exe -f sandbox.vhdx mftparser --output=body -D output --output-file=sandbox.body
```

When we search the `kernel32.dll` (a sample system file) record in the output, we encounter the following text:

```
0|[MFT FILE_NAME] Windows\System32\kernel32.dll (Offset: 0x3538c00)|1251|---a---S--o----
|0|0|764456|1604310972|1596874670|1603021550|1596874670
0|[MFT STD_INFO] Windows\System32\kernel32.dll (Offset: 0x3538c00)|1251|---a---Sr-o----
|0|0|764456|1606900209|1596874670|1603021550|1596874670
```

We can see similar reparse ("*S*") and offline ("*o*") attributes as we did earlier, but Volatility doesn't give us any additional information. We can use the offset of the MFT record, `0x3538c00`, to launch our own manual parse.

We used the next NTFS documentation for the parsing process. We do not provide a full specification of the MFT format, but to put it simply, MFT records contain a variable number of attributes, and each one has its own header and a payload. We are looking for the `$REPARSE_POINT` attribute, which is identified by the ordinal `0xC0`.

Table 4.2. Layout of a resident unnamed attribute header

Offset	Size	Value	Description
0x00	4		Attribute Type (e.g. 0x10, 0x60)
0x04	4		Length (including this header)
0x08	1	0x00	Non-resident flag
0x09	1	0x00	Name length
0x0A	2	0x00	Offset to the Name
0x0C	2	0x00	Flags
0x0E	2		Attribute Id (a)
0x10	4	L	Length of the Attribute
0x14	2	0x18	Offset to the Attribute
0x16	1		Indexed flag
0x17	1	0x00	Padding
0x18	L		The Attribute

Figure 5 – MFT attribute header structure.

Table 2.32. Layout of the $REPARSE_POINT (0xC0) attribute (Microsoft Reparse Point)

Offset	Size	Description
~	~	Standard Attribute Header
0x00	4	Reparse Type (and Flags)
0x04	2	Reparse Data Length
0x06	2	Padding (align to 8 bytes)
0x08	V	Reparse Data (a)

Figure 6 – `$REPARSE_POINT` attribute payload structure.

Our parsing effort with the structures listed above yields the following data:

```
$REPARSE_POINT Attribute
--------------- Attribute Header ---------------
C0 00 00 00 - Type ($REPARSE_POINT)
78 00 00 00 - Length
00          - Non-resident flag
00          - Name length
00 00       - Offset to the name
00 00       - Flags
03 00       - Attribute Id (a)
5C 00 00 00 - Length of the attribute
18 00       - Offset to the attribute
00          - Indexed flag
00          - Padding
---------------- Attribute Data ----------------
18 10 00 90 - Reparse tag
54 00       - Reparse data length
00 00       - Padding
----------------- Reparse Data -----------------
01 00 00 00 - Version ?
00 00 00 00 - Reserved ?
77 F6 64 82 B0 40 A5 4C BF 9A 94 4A C2 DA 80 87 - Referenced GUID
3A 00       - Path string size
57 00 69 00 6E 00 64 00 6F 00 77 00 73 00 5C 00
53 00 79 00 73 00 74 00 65 00 6D 00 33 00 32 00
5C 00 6B 00 65 00 72 00 6E 00 65 00 6C 00 33 00
32 00 2E 00 64 00 6C 00 6C 00 - Path string
```

A few important notes:

- We didn't find any public documentation for Microsoft's reparse data structure, but it wasn't too difficult to reverse-engineer.
- The reparse tag `0x90001018` is defined here as `IO_REPARSE_TAG_WCI_1` with the next description:

> "Used by the Windows Container Isolation filter. Server-side interpretation only, not meaningful over the wire."

- While reverse-engineering Windows modules in this research, several times we came across the referenced GUID `77 F6 64 82 B0 40 A5 4C BF 9A 94 4A C2 DA 80 87` as a hardcoded value. This value indicates a reference to the host base layer, which we talk about it later.
- The path in the reparse data shows the relative path of our sample file: `Windows\System32\kernel32.dll`

Based on the above information, we can conclude that files are "linked" by the underlying file system (probably to a designated FS filter), but many questions are still unanswered: how is the VHDx constructed, what is the purpose of other VHDx's, and what component is responsible for linking to the host files.

VHDx Layering

If we track *Procmon* logs during the sandbox creation, we notice a series of VHDx access attempts:

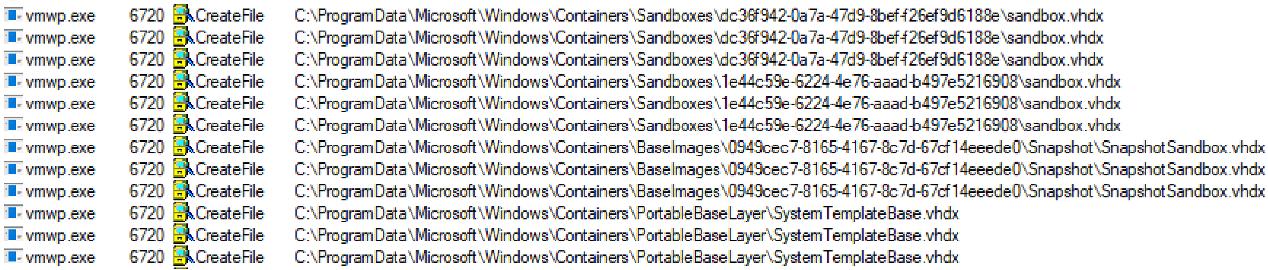


Figure 7 – VHDx layering lead.

While the first one is the "real" VHDx which we parsed previously, it is followed by 3 other VHDx accesses. We suspect that Microsoft used some sort of layering for the virtual disk templates.

Our theory is easily verified by inspecting the VHDx files using the binary editor:

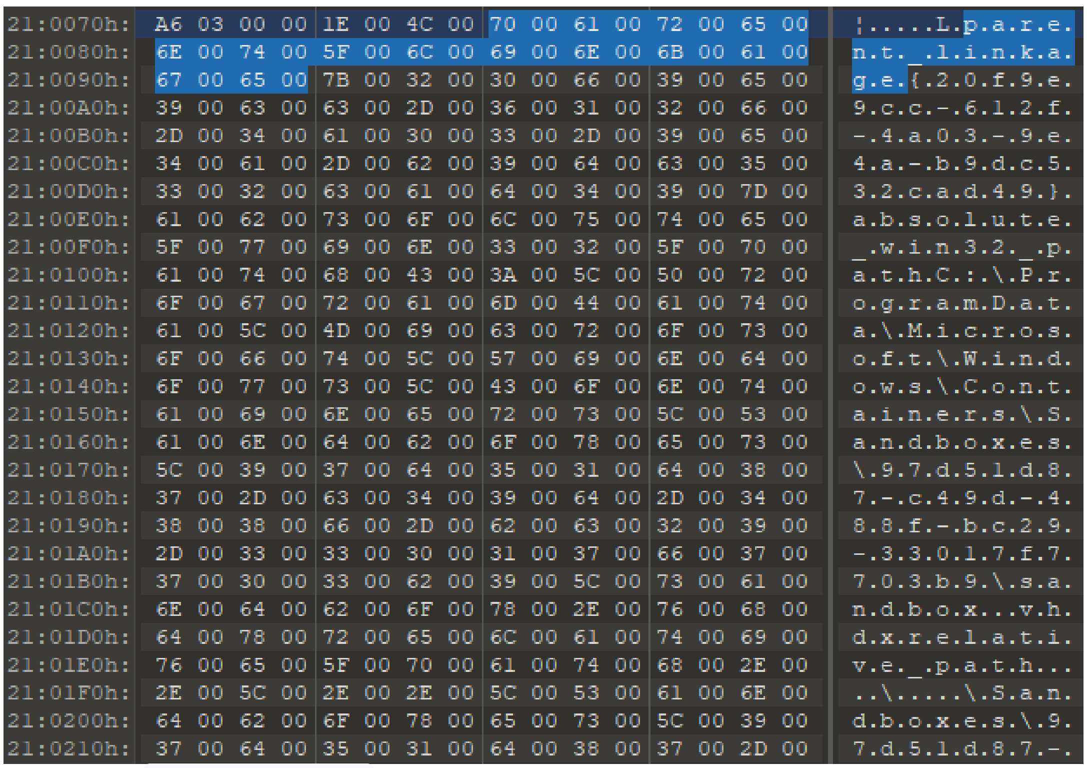


Figure 8 – `parent_linkage` tag in 010 Editor.

The parent locator in VHDx format can be given using multiple methods: absolute path, relative path, and volume path. The documentation can be found here.

With that knowledge, we can build the next layering:

- `Sandboxes\<new_sandbox_guid>\sandbox.vhdx` – The "real" VHDx.
- `Sandboxes\<constant_guid_per_installation>\sandbox.vhdx` – Created once per sandbox install.
- `BaseImages\0949cec7-8165-4167-8c7d-67cf14eeede0\Snapshot\SnapshotSandbox.vhdx` – Probably relevant to the base layer snapshot.
- `PortableBaseLayer\SystemTemplateBase.vhdx` – Base template.

When we browse these virtual disks, we notice files are still missing; some system folders are empty, as well as folders for Users/Program Files and various other files.

Playing with Procmon leads us to understand that another important layer is missing: the OS base layer.

OS Base Layer

The OS base layer main file exists in the sandbox working folder in the next path: `BaseImages\0949cec7-8165-4167-8c7d-67cf14eeede0\BaseLayer.vhdx` . By looking at the installation process through Procmon, we can see that the next `.wim` (Windows Imaging Format) file, `C:\Windows\Containers\serviced\WindowsDefenderApplicationGuard.wim` , is extracted into the `PortableBaseLayer` folder by the same name, and is copied and renamed into the base layer file above. This shows yet another similarity between WDAG and Windows Sandbox.

When we browsed the `BaseLayer.vhdx` disk, we could see the complete structure of the created sandbox, but system files were still “physically” missing. Parsing the MFT record for `kernel32.dll` like we did previously results in the same `$REPARSE_POINT` attribute but with a different tag: `0xA0001027` : `IO_REPARSE_TAG_WCI_LINK_1` . Remember this tag for later.

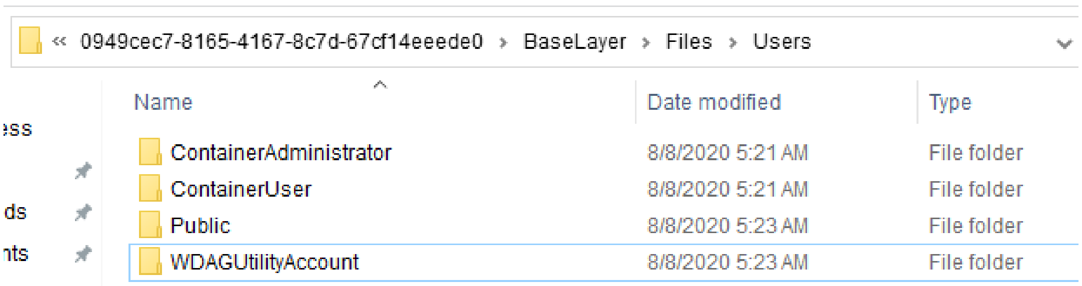


Figure 9 – Base layer user folders.

In addition, when we run `mountvol` command, we see that the base layer VHDx is mounted to the same directory where it exists:

```
\\?\Volume{629458e4-0000-0000-0000-010000000000}\
    C:\ProgramData\Microsoft\Windows\Containers\BaseImages\0949cec7-8165-4167-8c7d-67cf14eeede0\BaseLayer\

\\?\Volume{27451705-0000-0000-0000-402400000000}\
    C:\
```

Figure 10 – Mounted OS base layer.

The service in charge of mounting that volume, and all previous functionality we mentioned up to this point, is the *Container Manager Service* `CmService.dll` .

This service runs an executable named `cmimageworker.exe` , with one of the next command line parameters, `expandpbl/deploy/clean` , to perform these actions.

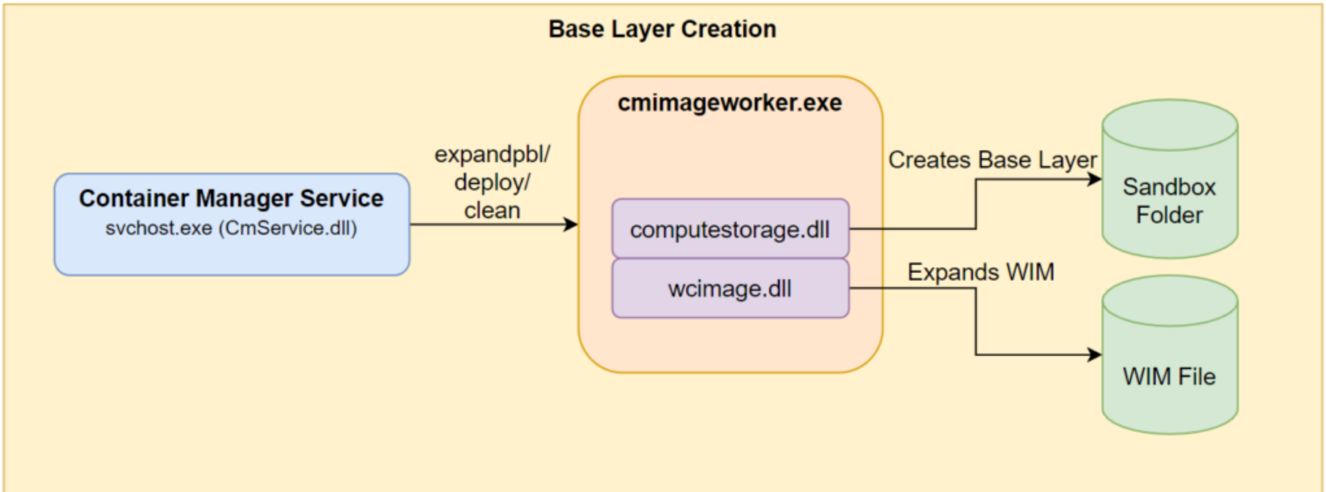


Figure 11 – `CmService` base layer creation.

We can observe the call to `computestorage!HcsSetupBaseOSLayer` in `cmimageworker.exe`, and part of the actual creation of the base layer in `computestorage.dll`.

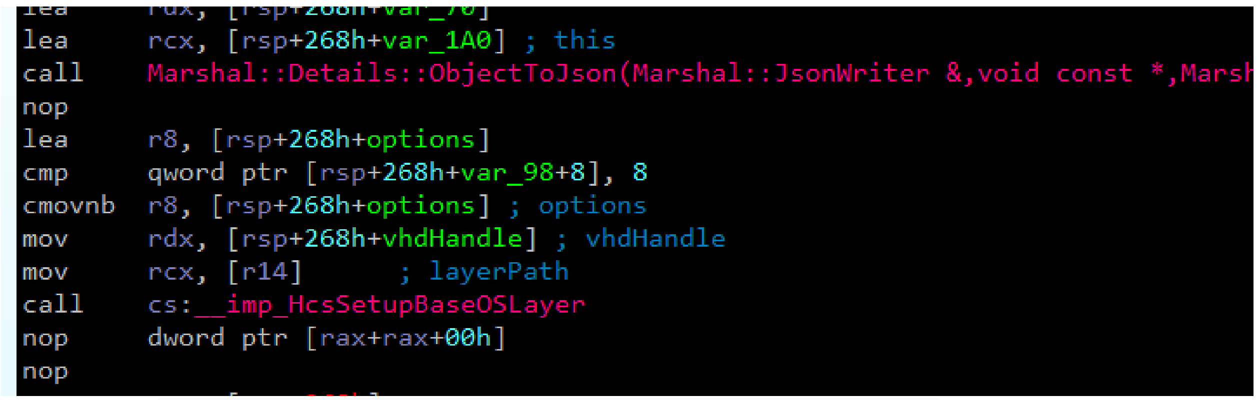


Figure 12 – `cmimageworker!Container::Manager::Hcs::ProcessImage` initiates base layer creation.

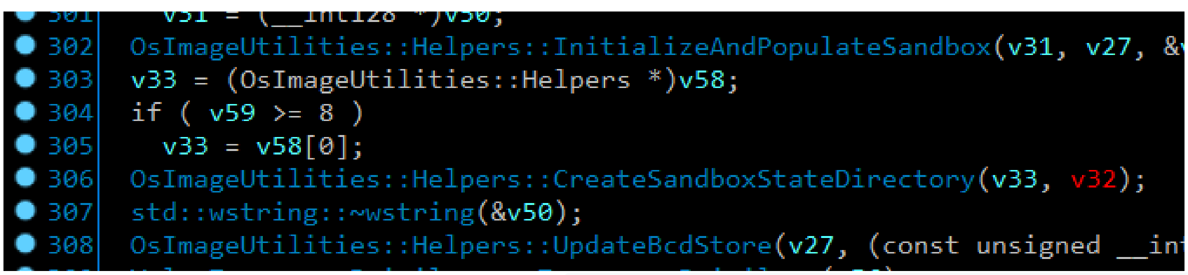


Figure 13 – Part of the base layer creation in `computestorage!OsImageUtilities::ProcessOsLayer`.

Microsoft issued the following statement regarding the sandbox:

> **Part of Windows** – everything required for this feature ships with Windows 10 Pro and Enterprise. No need to download a VHD!

So far, we understand crucial implementation details regarding that feature. Let's continue to see how the container is executed.

Running the Sandbox

Running the Windows Sandbox application triggers an execution flow which we won't elaborate on here. We just mention that the flow leads to `CmService` executing `vmcompute!HcsRpc_CreateSystem` through an RPC call. Another crucial service, `vmcompute.exe`, runs and orchestrates all compute systems (containers) on the host.

In our case, the *CreateSystem* command also receives the next configuration JSON which describes the desired machine:

Note – The JSON is cut for readability. You can access the full JSON in **Appendix A**.


```
{
    "Owner": "Madrid",
                ...
    "VirtualMachine": {
                                ...
        "Devices": {
            "Scsi": {
                "primary": {
                    "Attachments": {
                        "0": {
                            "Type": "VirtualDisk",
                            "Path":
"C:\\ProgramData\\Microsoft\\Windows\\Containers\\Sandboxes\\025b00c8-849a-4e00-bcb2-
c2b8ec698bab\\sandbox.vhdx",

...
                        }
                    }
                }
            },
                                                ...
            "VirtualSmb": {
                "Shares": [{
                    "Name": "os",
                    "Path": "C:\\ProgramData\\Microsoft\\Windows\\Containers\\BaseImages\\0949cec7-
8165-4167-8c7d-67cf14eeede0\\BaseLayer\\Files",
                                                                                  ...
                }],
`            },
                                                ...
        },
                                ...
        "RunInSilo": {
            "SiloBaseOsPath":
"C:\\ProgramData\\Microsoft\\Windows\\Containers\\BaseImages\\0949cec7-8165-4167-8c7d-
67cf14eeede0\\BaseLayer\\Files",
            "NotifySiloJobCreated": true,
            "FileSystemLayers": [{
                "Id": "8264f677-40b0-4ca5-bf9a-944ac2da8087",
                "Path": "C:\\",
                "PathType": "AbsolutePath"
            }]
        },
                                ...
    },
                ...
}
```


This JSON is created at `CmService!Container::Manager::Hcs::Details::GenerateCreateComputeSystemJson`. We didn't manage to track any file which helps build that configuration.

Before we start analyzing the interesting fields in the JSON, we want to mention this article by Palo Alto Networks. The article explains the container internals, and how ***Job*** and ***Silo*** objects are related.

The first interesting configuration tag is `RunInSilo`. This tag triggers a code flow in `vmcompute` which leads us to the next stack trace:

```
3: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff9a00`8da57648 fffff806`85d2b7fb     wcifs!WcPortMessage
01 ffff9a00`8da57650 fffff806`85d63499     FLTMGR!FltpFilterMessage+0xdb
... (REDUCTED)
0b 0000004d`4218dbf0 00007ffa`08c5363d     FLTLIB!FilterSendMessage+0x31
0c 0000004d`4218dc40 00007ffa`08c48686     wc_storage!WciSetupFilter+0x195
0d 0000004d`4218dcf0 00007ffa`22e06496     wc_storage!WcAttachFilterEx+0x156
0e 0000004d`4218dee0 00007ffa`22de5a66     container!container::FilesystemProvider::Setup+0x15e
0f 0000004d`4218dfc0 00007ffa`22ded4ad     container!container_runtime::CreateContainerObject+0x106
10 0000004d`4218e010 00007ffa`22decf3c     container!container::CreateContainer+0x10d
11 0000004d`4218e4a0 00007ff6`fcf0bc7f     container!WcCreateContainer+0x1c
12 0000004d`4218e4d0 00007ff6`fcf0c5c4
vmcompute!ComputeService::JobUtilities::ConvertJobObjectToContainer+0xcb
13 0000004d`4218e590 00007ff6`fce8573f
vmcompute!ComputeService::JobUtilities::CreateSiloForIsolatedWorkerProcess+0x4dc
14 0000004d`4218e8c0 00007ff6`fce875c5
vmcompute!ComputeService::Management::Details::PrepareJobForWorkerProcess+0x17b
15 0000004d`4218e9a0 00007ff6`fcee6cbb
vmcompute!ComputeService::Management::Details::ConstructVmWorker+0xfd5
... (REDUCTED)
```

From the stack, we can understand that whenever the compute system receives the silo configuration, it creates and configures a container through a `container!WcCreateContainer` call. As part of its configuration, it also communicates with the `wcifs.sys` driver through `FLTLIB!FilterSendMessage`. We explain this driver and its purpose shortly.

The second interesting feature is the `VirtualSmb` tag for creating the respective shares for the mounted base layer path we mentioned previously. We'll get back to this shortly as well.

Container Isolation

As we can see in the stack trace, the container creation includes opening the filter communication channel on port `\WcifsPort` with the `wcifs.sys` driver, *Windows Container Isolation FS Filter Driver*. This is a common method for a user mode code to communicate with filter drivers.

This mini-filter driver has an important part in the implementation of the container filesystem virtualization. This driver fills this role in **both the guest and the host**.

File system filter drivers are usually quite complex, and this one isn't an exception. Luckily, James Forshaw of Google Project Zero recently wrote a great article which explains the low-level design of Windows FS filter drivers, which helps us understand the logic in our case.

We can divide the driver logic into 2 parts:

- Driver configuration – The configuration depends on whether the driver runs on the guest or on the host system.
- Handling the operation callbacks, such as `WcPreCreate`, `WcPostCreate`, `WcPreRead`, and `WcPostRead`. These callbacks contain the main logic, data manipulation and proper redirections.

We'll explain some of the methods this driver uses to understand the ecosystem of the sandbox.

Initial Configuration

Guest Configuration

As we said previously, both the host, and the guest use this driver but in different ways.

The guest receives a set of parameters via the registry for its initial configuration. Some of these params are at `HKLM\SYSTEM\CurrentControlSet\Control` and `HKLM\SYSTEM\CurrentControlSet\Control\BootContainer` as we can see below:

BootContainerGuid	REG_SZ	{1b3979c8-279b-42eb-b2b9-750767ee9e3f}
BootDriverFlags	REG_DWORD	0x0000001c (28)
ContainerId	REG_SZ	cd6b0640-9a65-4d74-8e41-66085c102228
ContainerType	REG_DWORD	0x00000004 (4)

Figure 14 – `HKLM\SYSTEM\CurrentControlSet\Control` config values.

InstanceName	REG_SZ	wcifs Outer Instance
ReparseTag	REG_DWORD	0x90001018 (2415923224)

Figure 15 – `HKLM\SYSTEM\CurrentControlSet\Control\BootContainer` config values.

You might notice the `IO_REPARSE_TAG_WCI_1` (code `0x90001018`), which we saw earlier in the "real" VHDx file. This tag, together with `IO_REPARSE_TAG_WCI_LINK_1`, which we saw as a reparse tag in `BaseLayer.vhdx`, are hardcoded into the `wcifs!WcSetBootConfiguration` method:

```
if ( KeyValueInformation.Type == REG_DWORD )
{
  reparseTagWci = *KeyValueInformation.Data;
  if ( *KeyValueInformation.Data == IO_REPARSE_TAG_WCI )
  {
    reparseTagWciLink = 0xA0000027;          // IO_REPARSE_TAG_WCI_LINK
  }
  else if ( *KeyValueInformation.Data == IO_REPARSE_TAG_WCI_1 )
  {
    reparseTagWciLink = 0xA0001027;          // IO_REPARSE_TAG_WCI_LINK_1
  }
```

Figure 16 – Hardcoded reparse tag values in `WcSetBootConfiguration`.

The second, more important part of the guest configuration is in `wcifs!WcSetupVsmbUnionContext`, where it sets up a virtualized layer known as a ***Union Context***. Behind the scenes, the driver stores customized data on several context objects and accesses them with the proper NT API – `FltGetInstanceContext`, `PsGetSiloContext`, and `FltGetFileContext`. These custom objects contain AVL trees and hash tables to efficiently look up the virtualized layers.

The `WcSetupVsmbUnionContext` method has two more interesting artifacts. One is a vSMB path which is part of the layer, and another is the `HOST_LAYER_ID` GUID which we saw previously in the parsed MFT and in the JSON that describes the virtual machine:

```
    vmsbEntry_->params |= 0x20u;
  RtlInitUnicodeString(&vmsbEntry_->vsmbPath, L"\\Device\\vmsmb\\VSMB-{dcc079ae-60ba-4d07-847c-3493609c0870}\\os\\");
  guid = *guids_;
```

Figure 17 – Hardcoded vSMB path in `WcSetupVsmbUnionContext`.

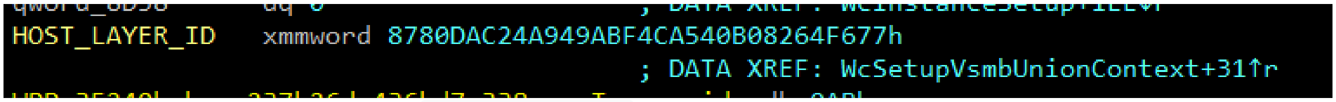


Figure 18 – Hardcoded GUID for `HOST_LAYER_ID`.

As we delve deeper, we see signs that a ***Virtual SMB*** method is used to share files between the guest and the host. Soon we'll see that vSMB is the **main method** for the base layer implementation and mapped folder sharing.

Host Configuration

For the host system, the main configuration happens when the parent compute process, `vmcompute`, initiates the container creation, and sends a custom message to `\WcifsPort`. This triggers `wcifs!WcPortMessage` which is a callback routine for any message sent to that specific port.

Below is a partial reconstruction of the message sent by the service to the filter driver:

```
struct WcifsPortMsg
{
  DWORD MsgCode;
  DWORD MsgSize;
  WcifsPortMsgSetUnion Msg;
};

struct WcifsPortMsgSetUnion
{
  DWORD MsgVersionOrCode;
  DWORD MsgSize;
  DWORD NumUnions;
  wchar_t InstanceName[50];
  DWORD InstanceNameLen;
  DWORD ReparseTag;
  DWORD ReparseTagLink;
  DWORD NotSure;
  HANDLE Job;
  BYTE ContextData[1];
};
```

The `ContextData` field also contains the device paths the union should map.

Operation Callbacks

During the registration, the filter driver supplies a set of callbacks for each operation it wants to intercept. The filter manager invokes these callbacks pre/post each file operation, as we can see below.

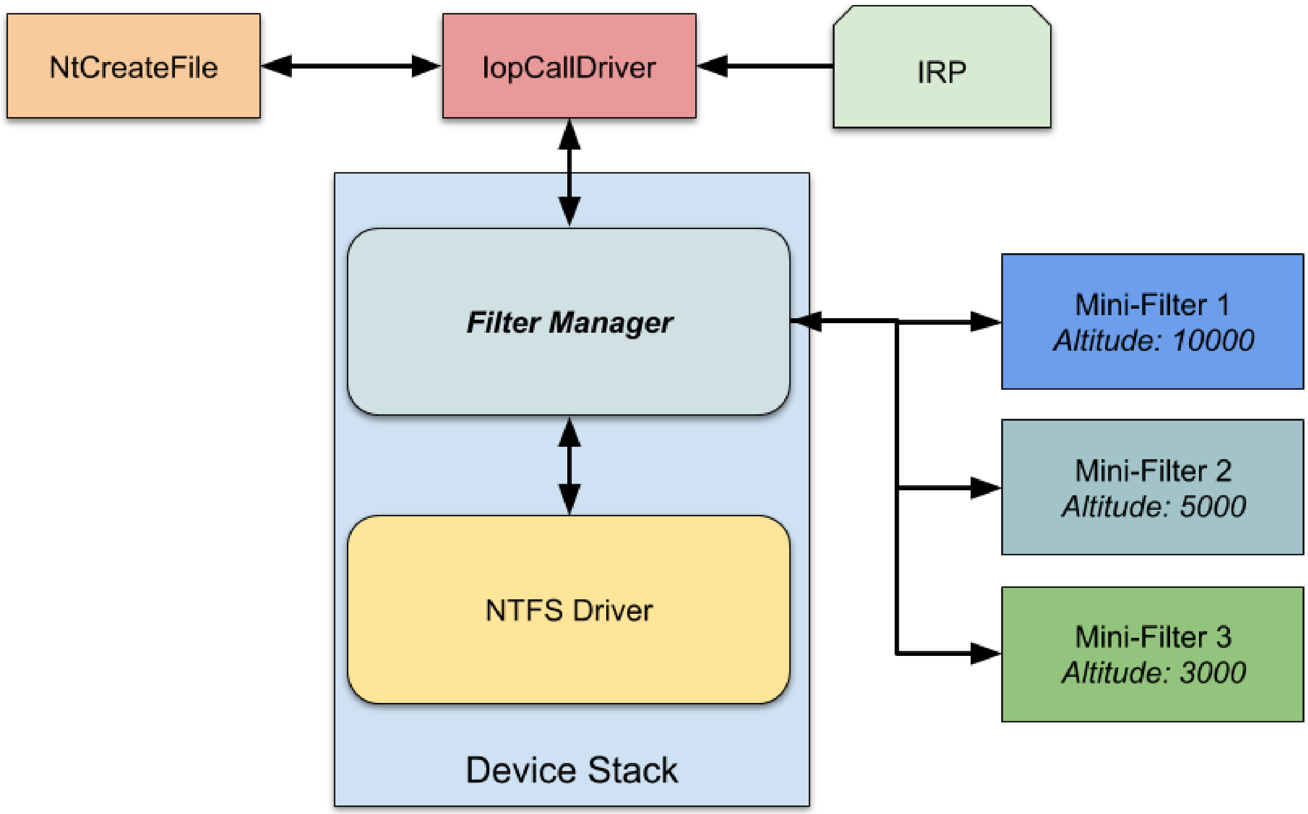


Figure 19 – Mini-filter architecture, courtesy of James Forshaw.

Without diving too much into the technical details, the driver defines and takes care of two custom reparse tags:

- **IO_REPARSE_TAG_WCI_1** – This is the main tag that indicates the file instance on the disk is virtual, and the real path can be found in its internal structures. Example uses of this "conversion":
  - The guest converts files from its native path `C:\Windows\system32\kernel32.dll` to vSMB path `\Device\vmsmb\VSMB-{dcc079ae-60ba-4d07-847c-3493609c0870}\os\Windows\System32\kernel32.dll` .
  - The host converts files from the base layer device path `C:\ProgramData\Microsoft\Windows\Containers\BaseImages\0949cec7-8165-4167-8c7d-67cf14eeede0\BaseLayer\Files\Windows\System32\en-US\apphelp.dll.mui` to the real path `C:\Windows\System32\en-US\apphelp.dll.mui` .
    This conversion is quite interesting, as it happens mainly in empty system folders in the base layer which contain this reparse tag (like the `en-US` folder).
- **IO_REPARSE_TAG_WCI_LINK_1** – This tag is used only on the host as far as we could tell, and links the system files from the base layer device path `C:\ProgramData\Microsoft\Windows\Containers\BaseImages\0949cec7-8165-4167-8c7d-67cf14eeede0\BaseLayer\Files\Windows\System32\kernel32.dll` to the real path `C:\Windows\System32\kernel32.dll` . Compared to the previous point, this example DLL file entry does exist in the base layer, and has this reparse tag.

The discovery that vSMB is the primary method for the OS base layer sharing was quite surprising. Now that we know it is a crucial communication method in the ecosystem the natural next step is to dig further inside.

(v)SMB File Sharing

During the sandbox installation, we noticed `vmcompute` creates several virtual shares by invoking `CreateFileW` to the storage provider device, and sends IOCTL `0x240328`. A sample path for such an invoke might look like this: `\??\STORVSP\VSMB\??\C:\ProgramData\Microsoft\Windows\Containers\BaseImages\0949cec7-8165-4167-8c7d-67cf14eeede0\BaseLayer\Files`.

The method that creates these shares is `vmcompute!ComputeService::Storage::OpenVsmbRootShare`. We can see its flow in the next stack trace:

```
3: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff9a00`8d48a178 fffff806`85fd6af8     storvsp!VspFileCreate
01 (Inline Function) --------`--------     Wdf01000!FxFileObjectFileCreate::Invoke+0x29
[minkernel\wdf\framework\shared\inc\private\common\FxFileObjectCallbacks.hpp @ 58]
... (REDUCTED)
11 0000004d`4210d690 00007ff6`fcf33700     KERNELBASE!CreateFileW+0x66
12 0000004d`4210d6f0 00007ff6`fceb8180
vmcompute!ComputeService::Storage::OpenVsmbRootShare+0x3ac
13 0000004d`4210d850 00007ff6`fceba0fc
vmcompute!ComputeService::VirtualMachine::Details::ConfigureVSMB+0x598
14 0000004d`4210da30 00007ff6`fceba908
vmcompute!ComputeService::VirtualMachine::Details::InitializeDeviceSettings+0x918
15 0000004d`4210eb90 00007ff6`fce86abd
vmcompute!ComputeService::VirtualMachine::CreateVirtualMachineConfiguration+0x68
16 0000004d`4210ebe0 00007ff6`fcee6cbb
vmcompute!ComputeService::Management::Details::ConstructVmWorker+0x4cd
... (REDUCTED)
```

In addition, when we map host folders to the guest using the WSB file configuration, the same method is called. For example, mapping the *Sysinternals* folder results in the next call to the driver: `\??\STORVSP\VSMB\??\C:\Users\hyperv-root\Desktop\SysinternalsSuite`.

Accessing Files via (v)SMB

After creating these shares, we can access them within the guest through the created alias. We can use the `type` command to print the `kernel32.dll` of the host with the next path `\\.\vmsmb\VSMB-{dcc079ae-60ba-4d07-847c-3493609c0870}\os\Windows\System32\kernel32.dll`:

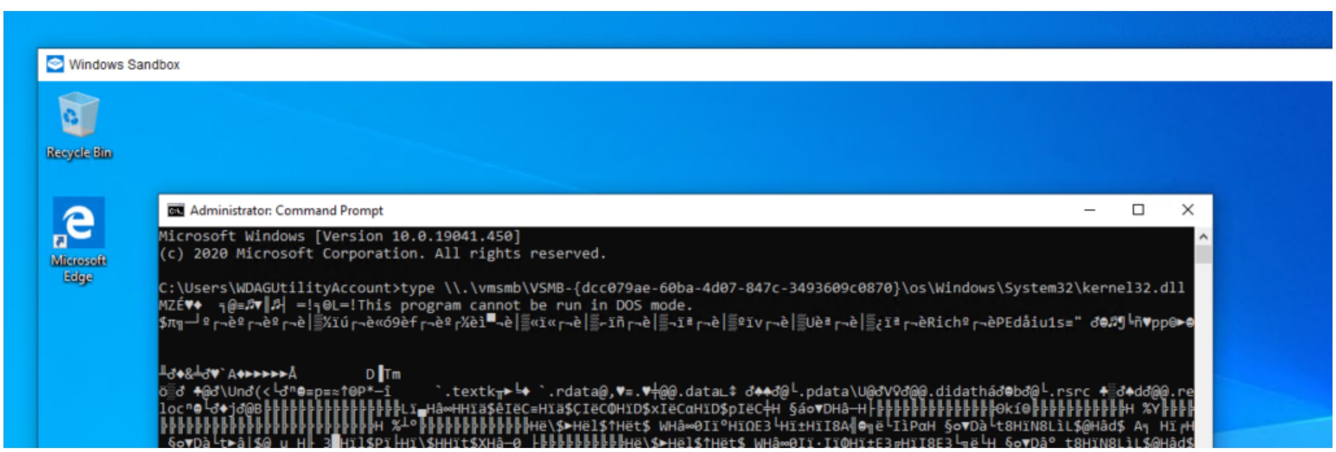


Figure 20 – Accessing the vSMB share.

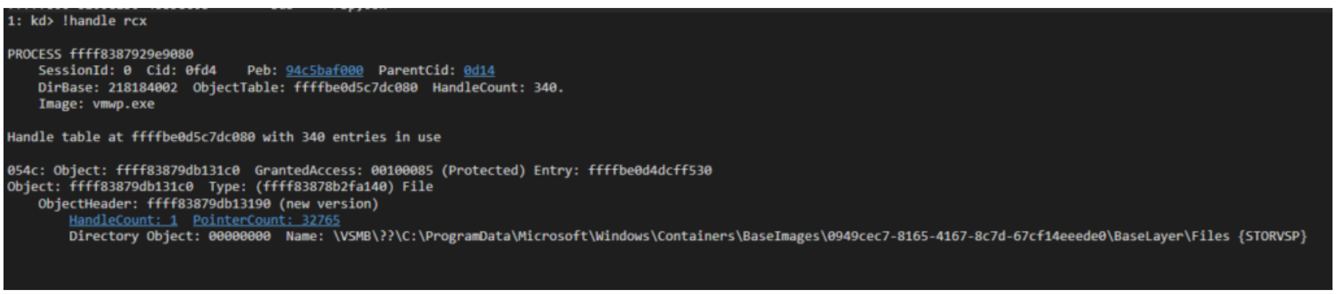
To serve the vSMB files, the `vmusrv` module, which is part of the VM worker process, creates a worker thread. This module is a user mode vSMB server which requests packets directly from the VMBus at the `vmusrv!VSmbpWorkerRecvLoop` routine, and then proceeds to process the packets.

Serving Create File Operation

Whenever `vmusrv` receives a *Create* SMB request, it initiates a new request to the storage provider driver. Such a call might look like this:

```
2: kd> k
 # Child-SP          RetAddr               Call Site
... (REDUCTED)
0c ffff9a00`8d9522e0 fffff806`892c4741     storvsp!VspVsmbCommonRelativeCreate+0x369
0d ffff9a00`8d952510 fffff806`892c3b7e     storvsp!VspVsmbHandleRelativeCreateFileRequest+0x321
0e ffff9a00`8d952790 fffff806`892c0f85     storvsp!VspVsmbDispatchIoControlForProcess+0x11e
0f ffff9a00`8d9527e0 fffff806`8100e522     storvsp!VspFastIoDeviceControl+0x175
... (REDUCTED)
13 000000ae`9c0ff298 00007ffa`110c0c0a     ntdll!NtDeviceIoControlFile+0x14
14 000000ae`9c0ff2a0 00007ffa`110c0456     vmusrv!CShare::OpenFileRelativeToShareRootInternal+0x306
15 000000ae`9c0ff3e0 00007ffa`110b9381     vmusrv!CShare::OpenFileRelativeToShareRoot+0x356
16 000000ae`9c0ff510 00007ffa`110b4451     vmusrv!CFSObject::CreateFileW+0x185
17 000000ae`9c0ff690 00007ffa`1109a568     vmusrv!CShare::Create+0x91
18 000000ae`9c0ff740 00007ffa`1109d74d     vmusrv!ProviderCallback_Create+0x30
19 000000ae`9c0ff780 00007ffa`1109c299     vmusrv!SrvCreateFile+0x331
1a 000000ae`9c0ff860 00007ffa`1109c6f0     vmusrv!Smb2ExecuteCreateReal+0x111
1b 000000ae`9c0ff940 00007ffa`110a08da     vmusrv!Smb2ExecuteCreate+0x30
1c 000000ae`9c0ff970 00007ffa`11098907     vmusrv!Smb2ExecuteProviderCallback+0x7e
1d 000000ae`9c0ff9d0 00007ffa`11088311     vmusrv!Smb2PacketProcessing+0x97
1e 000000ae`9c0ffa40 00007ffa`11087225     vmusrv!Smb2PacketProcessingCallback+0x11
... (REDUCTED)
```

The communication with the storage provider is done through an IOCTL with the code `0x240320`, while the referenced handle is the vSMB path opened on the initialization phase:


```
1: kd> !handle rcx

PROCESS ffff8387929e9080
    SessionId: 0  Cid: 0fd4    Peb: 94c5baf000  ParentCid: 0d14
    DirBase: 218184002  ObjectTable: ffffbe0d5c7dc080  HandleCount: 340.
    Image: vmwp.exe

Handle table at ffffbe0d5c7dc080 with 340 entries in use

054c: Object: ffff83879db131c0  GrantedAccess: 00100085 (Protected) Entry: ffffbe0d4dcff530
Object: ffff83879db131c0  Type: (ffff83878b2fa140) File
    ObjectHeader: ffff83879db13190 (new version)
        HandleCount: 1  PointerCount: 32765
        Directory Object: 00000000  Name: \VSMB\??\C:\ProgramData\Microsoft\Windows\Containers\BaseImages\0949cec7-8165-4167-8c7d-67cf14eeede0\BaseLayer\Files {STORVSP}
```

Figure 21 – The handle in which the IOCTL is referred.

If we look closely at `storvsp!VspVsmbCommonRelativeCreate`, we see that every execution is followed by a call to `nt!IoCreateFileEx`. This call contains the relative path of the desired file with an additional `RootDirectory` field which represents the `\Files` folder in the mounted base layer VHDx:

```
0: kd> g
Breakpoint 0 hit
storvsp!VspVsmbCommonRelativeCreate:
fffff807`3b072e98 48895c2410      mov     qword ptr [rsp+10h],rbx
0: kd> g
Breakpoint 1 hit
nt!IoCreateFileEx:
fffff807`34490000 488bc4          mov     rax,rsp
0: kd> r r8
r8=ffffc20eaef3f3d8
0: kd> dt nt!_OBJECT_ATTRIBUTES ffffc20eaef3f3d8
   +0x000 Length           : 0x30
   +0x008 RootDirectory    : 0xffffffff`80001e3c Void
   +0x010 ObjectName       : 0xffffc20e`aef3f5c8 _UNICODE_STRING "Windows\SystemApps\Microsoft.MicrosoftEdge_8wekyb3d8bbwe\Assets"
   +0x018 Attributes       : 0x240
   +0x020 SecurityDescriptor : (null)
   +0x028 SecurityQualityOfService : (null)
0: kd> !handle 0xffffffff`80001e3c

PROCESS ffffaa0db59e1080
    SessionId: 0  Cid: 054c    Peb: ebdd9c0000  ParentCid: 0cbc
    DirBase: 256cb002  ObjectTable: ffff800f3f9fccc0  HandleCount: 5608.
    Image: vmwp.exe

Kernel handle table at ffff800f32e28cc0 with 3220 entries in use

80001e3c: Object: ffffaa0db3699e70  GrantedAccess: 00100020 (Audit) Entry: ffff800f38bfb8f0
Object: ffffaa0db3699e70  Type: (ffffaa0dadcf7a60) File
    ObjectHeader: ffffaa0db3699e40 (new version)
        HandleCount: 1  PointerCount: 23725
        Directory Object: 00000000  Name: \Files {HarddiskVolume4}
```

Figure 22 – Execution of `IoCrateFileEx` by `storvsp.sys`.

Serving Read/Write Operation

Read/Write operations are executed by the worker thread in `vmusrv!CFSObject::Read/vmusrv!CFSObject::Write`. If the file is small enough, the thread simply executes `ReadFile/WriteFile` on the handle. Otherwise it maps the file to the memory, and transfers it efficiently through RDMA on top of VMBus. This transfer is executed at `vmusrv!SrvConnectionExecuteRdmaTransfer`, while the RDMA communication is done with the `RootVMBus` device (host VMBus device name) using IOCTL `0x3EC0D3` or `0x3EC08C`.

```
2: kd> k
... (REDUCTED)
06 ffffad0e`3bee7650 fffff800`36225b62     vmbusr!RootIoctlRdmaFileIoHandleMappingComplete+0x10f
07 ffffad0e`3bee7690 fffff800`361fee21     vmbusr!RootIoctlRdmaFileIo+0xf2
08 ffffad0e`3bee76f0 fffff800`339da977     vmbusr!RootIoctlDeviceControlPreprocess+0x191
... (REDUCTED)
12 00000009`ae27f7e8 00007ffe`281ce773     ntdll!NtDeviceIoControlFile+0x14
13 00000009`ae27f7f0 00007ffe`281dcbd2     vmusrv!SrvConnectionExecuteRdmaTransfer+0x24f
14 00000009`ae27f940 00007ffe`281d4874     vmusrv!CFile::ReadFileRdma+0xc2
15 00000009`ae27f9c0 00007ffe`281c218e     vmusrv!CFSObject::Read+0x94
16 00000009`ae27fa00 00007ffe`281c08da     vmusrv!Smb2ExecuteRead+0x1be
17 00000009`ae27fa60 00007ffe`281b8907     vmusrv!Smb2ExecuteProviderCallback+0x7e
18 00000009`ae27fac0 00007ffe`281a6a4e     vmusrv!Smb2PacketProcessing+0x97
19 00000009`ae27fb30 00007ffe`3bba6fd4     vmusrv!SmbWorkerThread+0xce
... (REDUCTED)
```

```
2: kd> !handle rcx

PROCESS ffff8387929e9080
    SessionId: 0  Cid: 0fd4    Peb: 94c5baf000  ParentCid: 0d14
    DirBase: 218184002  ObjectTable: ffffbe0d5c7dc080  HandleCount: 3179.
    Image: vmwp.exe

Handle table at ffffbe0d5c7dc080 with 3179 entries in use

0540: Object: ffff838792ec5690  GrantedAccess: 0012019f (Inherit) (Audit) Entry: ffffbe0d4dcff500
Object: ffff838792ec5690  Type: (ffff83878b2fa140) File
    ObjectHeader: ffff838792ec5660 (new version)
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \rdma\494 {RootVMBus}
```

Figure 23 – Communication with `\Device\RootVmBus\rdma\494` for the read/write operation.

Guest-to-Host Flow

Based on a few insights from this article explaining the `Storvsc.sys/Storvsp.sys` relationship, we can combine all previous technical blocks to the next file access flow.

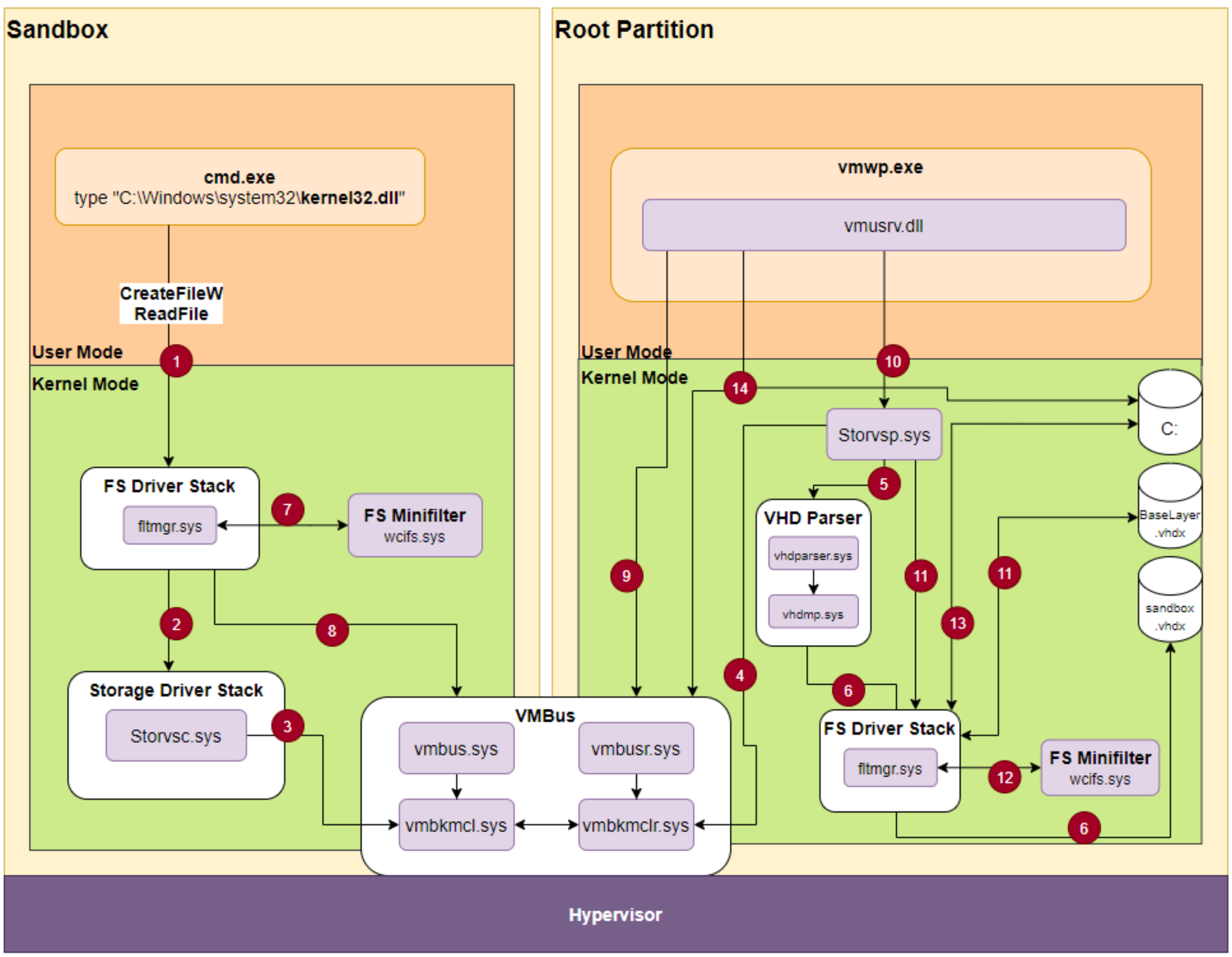


Figure 24 – File access flow.

1. We use the command `type` to open and print the content of the `kernel32.dll` file. This is a system file, and therefore the sandbox doesn't own its copy, but uses the host's copy.

2. The guest is not aware that the file doesn't exist, so it performs a normal file access through the filesystem driver stack up to the storage driver stack.
3. The Hyper-V storage consumer `Storvsc.sys` is a miniport driver, meaning it acts as the virtual storage for the guest. It receives and forwards SCSI requests over the VMBus.
4. The storage provider `Storvsp.sys` has a worker thread listening for new messages over the VMBus at `storvsp!VspPvtKmclProcessingComplete`.
5. The provider parses the VMBus request, and passes it to `vhdparser!NVhdParserExecuteScsiRequestDisk`, which executes `vhdmp.sys`, the VHD parser driver.
6. Eventually, `vhdmp.sys` accesses the physical instance of `sandbox.vhdx` through the filter manager, and performs read/write operation. In this case, it reads the data requested by the guest filesystem filter manager. That data is returned to the filter manager for further analysis.
7. As explained previously, the returned entry is tagged with a WCI reparse tag and with the host layer GUID. When `wcifs.sys` executes its post-create operation on the file, it looks for the union context for that device, and replaces the file object with the next one: `\Device\vmsmb\VSMB-{dcc079ae-60ba-4d07-847c-3493609c0870}\os\Windows\System32\kernel32.dll`
8. The `\Device\vmsmb` device was created as an SMB share, so the filter manager accesses it like any other normal share. Behind the scenes, it performs SMB requests over VMBus to the host.
9. The vSMB user-mode server `vmusrv.dll` polls the `\\.\VMbus\` device for new messages in its worker thread method `vmusrv!SmbWorkerThread`.
10. As we showed previously, in a create operation, the server communicates with the storage provider through IOCTL on the handle of mounted OS base layer: `\Device\STORVSP\VSMB\??\C:\ProgramData\Microsoft\Windows\Containers\BaseImages\0949cec7-8165-4167-8c7d-67cf14eeede0\BaseLayer\Files`
11. The storage provider executes the file request through `IoCreateFileEx`. That request is relative, and contains the `RootDirectory` of the mounted OS layer. This triggers the filter manager to open the file in the mounted OS layer.
12. Similar to step **(7)**, the returned entry contains a WCI reparse tag, which causes `wcifs.sys` to change the file object in the post-create method. It changes the file object to its physical path: `C:\Windows\System32\kernel32.dll`
13. Access the host `kernel32.dll` file, and return back to the guest.
14. For a `ReadFile` operation, the `wcifs.sys` driver saves a context state on top of the file object to help it perform a read/write operation. In addition, the worker thread `vmusrv` executes the read request either with direct access to the file, or through RDMA on top VMBus.

The actual process is much more complex, so we tried to focus on the components crucial to the virtualization.

The sandbox also allows mapping folders from host to guest through its configuration. Such folders receive a unique alias for the vSMB path, and the access is similar to the OS layer. The only difference is that the path is altered in the guest filter manager by `bindflt.sys`.

For example, if we map the *SysinternalsSuite* folder to the guest Desktop folder, the path `C:\Users\WDAGUtilityAccount\Desktop\SysinternalsSuite\Procmon.exe` is altered into `\Device\vmsmb\VSMB-{dcc079ae-60ba-4d07-847c-3493609c0870}\db64085bcd96aab59430e21d1b386e1b37b53a7194240ce5e3c25a7636076b67\Procmon.exe`, which leaves rest of the process the same.

Playing with the Sandbox

One of our targets in this research was to modify the base layer content according to our needs. Now that we understand the ecosystem, it appears to be quite easy.

The modification has a few simple steps:

1. Stop `CmService`, the service that creates and maintains the base layer. When the service is unloaded, it also removes the base layer mounting.
2. Mount the base layer (it is in the `C:\ProgramData\Microsoft\Windows\Containers\BaseImages\0949cec7-8165-4167-8c7d-67cf14eeede0\BaseLayer.vhdx` file). This can be done by double clicking, or using the `diskmgmt.msc` utility.
3. Make modifications to the base layer. In our case, we added all FLARE post-installation files.
4. Unmount the base layer.
5. Start `CmService`.

The moment we start the sandbox, we have our awesome FLARE VM!

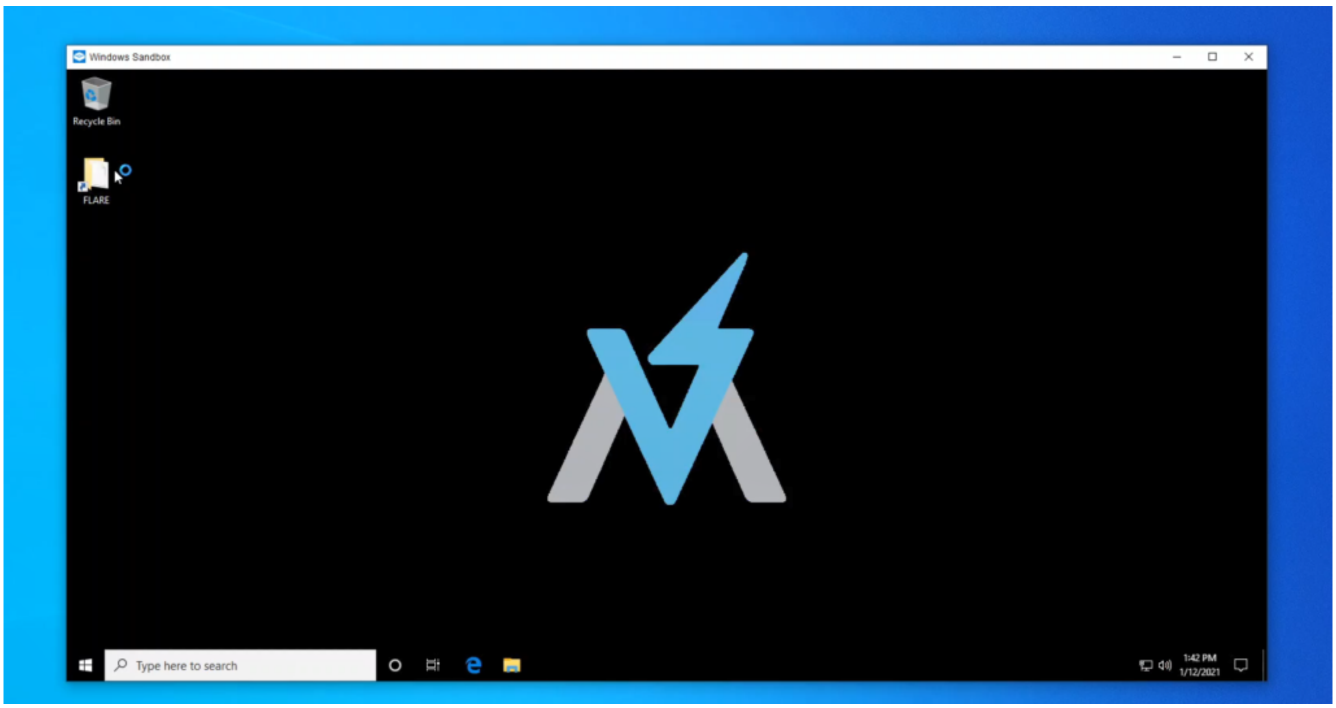


Figure 25 – FLARE VM on top of the Windows Sandbox.

Summary

When we started researching Windows Sandbox, we had no idea that such a "simple" operation boils down to a complex flow with several Microsoft internal undocumented technologies such as vSMB and Container Isolation.

We hope this article will help the community with further information gathering and bug hunting. For us, this was a big first step into researching and understanding virtualization related technologies.

For any technical feedback, feel free to reach out on twitter.

Links

Hyper-V VmSwitch RCE Vulnerability

https://www.youtube.com/watch?v=025r8_TrV8I

Windows Sandbox

https://techcommunity.microsoft.com/t5/windows-kernel-internals/windows-sandbox/ba-p/301849

Windows Sandbox WSB Configuration

https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-sandbox/windows-sandbox-configure-using-wsb-file

Windows Containers

NTFS Attributes

https://www.urtech.ca/2017/11/solved-all-ntfs-attributes-defined/

Reparse Point

https://docs.microsoft.com/en-us/windows/win32/fileio/reparse-points

NTFS Documentation

https://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf

NTFS Reparse Tags

https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-fscc/c8e77b37-3909-4fe6-a4ea-2b9d423b1ee4

VHDx Parent Locator

https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-vhdx/b6332a98-624d-46b8-bd0e-b77b573662f9

FS Filter Driver – Communication between User Mode and Kernel Mode

https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/communication-between-user-mode-and-kernel-mode

Hunting for Bugs in Windows Mini-Filter Drivers

https://googleprojectzero.blogspot.com/2021/01/hunting-for-bugs-in-windows-mini-filter.html

Hyper-V Storvsp.sys-Strovsc.sys Flow

https://www.linkedin.com/pulse/hyper-v-architecture-internals-pravin-gawale/

RDMA Explained by Microsoft

https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v-virtual-switch/rdma-and-switch-embedded-teaming

Appendix A

Windows Sandbox JSON configuration for vmwp

```
{
    "Owner": "Madrid",
    "SchemaVersion": {
        "Major": 2,
        "Minor": 1
    },
    "VirtualMachine": {
        "StopOnReset": true,
        "Chipset": {
            "Uefi": {
                "BootThis": {
                    "DeviceType": "VmbFs",
                    "DevicePath": "\\EFI\\Microsoft\\Boot\\bootmgfw.efi"
                }
            }
        },
        "ComputeTopology": {
            "Memory": {
                "SizeInMB": 1024,
                "Backing": "Virtual",
                "BackingPageSize": "Small",
                "FaultClusterSizeShift": 4,
                "DirectMapFaultClusterSizeShift": 4,
                "EnablePrivateCompressionStore": true,
                "EnableHotHint": true,
                "EnableColdHint": true,
                "SharedMemoryMB": 2048,
                "SharedMemoryAccessSids": ["S-1-5-21-2542268174-3140522643-1722854894-1001"],
                "EnableEpf": true,
                "EnableDeferredCommit": true
            },
            "Processor": {
                "Count": 4,
                "SynchronizeHostFeatures": true,
                "EnableSchedulerAssist": true
            }
        },
        "Devices": {
            "Scsi": {
                "primary": {
                    "Attachments": {
                        "0": {
                            "Type": "VirtualDisk",
                            "Path":
"C:\\ProgramData\\Microsoft\\Windows\\Containers\\Sandboxes\\025b00c8-849a-4e00-bcb2-
c2b8ec698bab\\sandbox.vhdx",
                            "CachingMode": "ReadOnlyCached",
                            "NoWriteHardening": true,
                            "DisableExpansionOptimization": true,
                            "IgnoreRelativeLocator": true,
                            "CaptureIoAttributionContext": true
                        }
                    }
                }
            },
            "HvSocket": {
                "HvSocketConfig": {
                    "DefaultBindSecurityDescriptor": "D:P(A;;FA;;;SY)",
                    "DefaultConnectSecurityDescriptor": "D:P(A;;FA;;;SY)",
                    "ServiceTable": {
                        "befcbc10-1381-45ab-946e-b1a12d6bce94": {
                            "BindSecurityDescriptor": "D:P(D;;FA;;;WD)",
                            "ConnectSecurityDescriptor": "D:P(D;;FA;;;WD)",
```

```
                        "AllowWildcardBinds": true
                    },
                    "7d2e0620-034a-4438-b0fd-ae27fc0172a1": {
                        "BindSecurityDescriptor": "D:P(A;;FA;;;SY)(A;;FA;;;S-1-5-83-0)",
                        "ConnectSecurityDescriptor": "D:P(D;;FA;;;WD)"
                    },
                    "a715ac94-b745-4889-9a0f-772d85a3cfa4": {
                        "BindSecurityDescriptor": "D:P(A;;FA;;;LS)",
                        "ConnectSecurityDescriptor": "D:P(A;;FA;;;LS)",
                        "AllowWildcardBinds": true
                    },
                    "7b3014c3-284a-40d4-a97f-9d23a75c6a80": {
                        "BindSecurityDescriptor": "D:P(D;;FA;;;WD)",
                        "ConnectSecurityDescriptor": "D:P(D;;FA;;;WD)",
                        "AllowWildcardBinds": true
                    },
                    "e97910d9-55bb-455e-9170-114fdfce763d": {
                        "BindSecurityDescriptor": "D:P(D;;FA;;;WD)",
                        "ConnectSecurityDescriptor": "D:P(D;;FA;;;WD)",
                        "AllowWildcardBinds": true
                    },
                    "e5afd2e3-9b98-4913-b37c-09de98772940": {
                        "BindSecurityDescriptor": "D:P(D;;FA;;;WD)",
                        "ConnectSecurityDescriptor": "D:P(D;;FA;;;WD)",
                        "AllowWildcardBinds": true
                    },
                    "abd802e8-ffcc-40d2-a5f1-f04b1d12cbc8": {
                        "BindSecurityDescriptor": "D:P(A;;FA;;;SY)(A;;FA;;;BA)(A;;FA;;;S-1-15-
3-3)(A;;FA;;;S-1-5-21-2542268174-3140522643-1722854894-1001)",
                        "ConnectSecurityDescriptor": "D:P(D;;FA;;;WD)"
                    },
                    "f58797f6-c9f3-4d63-9bd4-e52ac020e586": {
                        "BindSecurityDescriptor": "D:P(A;;FA;;;SY)",
                        "ConnectSecurityDescriptor": "D:P(A;;FA;;;SY)",
                        "AllowWildcardBinds": true
                    }
                }
            }
        },
        "EnhancedModeVideo": {
            "ConnectionOptions": {
                "AccessSids": ["S-1-5-21-2542268174-3140522643-1722854894-1001"],
                "NamedPipe": "\\\\.\\pipe\\025b00c8-849a-4e00-bcb2-c2b8ec698bab"
            }
        },
        "GuestCrashReporting": {
            "WindowsCrashSettings": {
                "DumpFileName":
"C:\\ProgramData\\Microsoft\\Windows\\Containers\\Dumps\\025b00c8-849a-4e00-bcb2-c2b8ec698bab.dmp",
                "MaxDumpSize": 4362076160,
                "DumpType": "Full"
            }
        },
        "VirtualSmb": {
            "Shares": [{
                "Name": "os",
                "Path": "C:\\ProgramData\\Microsoft\\Windows\\Containers\\BaseImages\\0949cec7-
8165-4167-8c7d-67cf14eeede0\\BaseLayer\\Files",
                "Options": {
                    "ReadOnly": true,
                    "TakeBackupPrivilege": true,
                    "NoLocks": true,
                    "ReparseBaseLayer": true,
```

```
                        "PseudoOplocks": true,
                        "PseudoDirnotify": true,
                        "SupportCloudFiles": true
                    }
                }],
                "DirectFileMappingInMB": 2048
            },
            "Licensing": {
                "ContainerID": "00000000-0000-0000-0000-000000000000",
                "PackageFamilyNames": []
            },
            "Battery": {},
            "KernelIntegration": {}
        },
        "GuestState": {
            "GuestStateFilePath":
"C:\\ProgramData\\Microsoft\\Windows\\Containers\\Sandboxes\\025b00c8-849a-4e00-bcb2-
c2b8ec698bab\\sandbox.vmgs"
        },
        "RestoreState": {
            "TemplateSystemId": "97d51d87-c49d-488f-bc29-33017f7703b9"
        },
        "RunInSilo": {
            "SiloBaseOsPath":
"C:\\ProgramData\\Microsoft\\Windows\\Containers\\BaseImages\\0949cec7-8165-4167-8c7d-
67cf14eeede0\\BaseLayer\\Files",
            "NotifySiloJobCreated": true,
            "FileSystemLayers": [{
                "Id": "8264f677-40b0-4ca5-bf9a-944ac2da8087",
                "Path": "C:\\",
                "PathType": "AbsolutePath"
            }]
        },
        "LaunchOptions": {
            "Type": "None"
        },
        "GuestConnection": {}
    },
    "ShouldTerminateOnLastHandleClosed": true
}
```