Bootkitting Windows Sandbox

secret.club/2022/08/29/bootkitting-windows-sandbox.html

August 29, 2022

mrexodia, mgoodings

Aug 29, 2022

Introduction & Motivation

Windows Sandbox is a feature that Microsoft added to Windows back in May 2019. As Microsoft puts it:

> Windows Sandbox provides a lightweight desktop environment to safely run applications in isolation. Software installed inside the Windows Sandbox environment remains "sandboxed" and runs separately from the host machine.

The startup is usually very fast and the user experience is great. You can configure it with a `.wsb` file and then double click that file to start a clean VM.

The sandbox can be useful for malware analysis and as we will show in this article, it can also be used for kernel research and driver development. We will take things a step further though and share how we can intercept the boot process and patch the kernel during startup with a bootkit.

TLDR: Visit the SandboxBootkit repository to try out the bootkit for yourself.

Windows Sandbox for driver development

A few years back Jonas L tweeted about the undocumented command `CmDiag`. It turns out that it is almost trivial to enable test signing and kernel debugging in the sandbox (this part was copied straight from my StackOverflow answer).

First you need to enable development mode (everything needs to be run from an *Administrator* command prompt):

```
CmDiag DevelopmentMode -On
```

Then enable network debugging (you can see additional options with `CmDiag Debug`):

```
CmDiag Debug -On -Net
```

This should give you the connection string:

```
Debugging successfully enabled.

Connection string: -k net:port=50100,key=cl.ea.rt.ext,target=<ContainerHostIp> -v
```

Now start WinDbg and connect to `127.0.0.1`:

```
windbg.exe -k net:port=50100,key=cl.ea.rt.ext,target=127.0.0.1 -v
```

Then you start Windows Sandbox and it should connect:

```
Microsoft (R) Windows Debugger Version 10.0.22621.1 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Using NET for debugging
Opened WinSock 2.0
Using IPv4 only.
Waiting to reconnect...
Connected to target 127.0.0.1 on port 50100 on local IP <xxx.xxx.xxx.xxx>.
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 19041 x64 target at (Sun Aug  7 10:32:11.311 2022 (UTC +
2:00)), ptr64 TRUE
Kernel Debugger connection established.
```

Now in order to load your driver you have to copy it into the sandbox and you can use `sc create` and `sc start` to run it. Obviously most device drivers will not work/freeze the VM but this can certainly be helpful for research.

The downside of course is that you need to do quite a bit of manual work and this is not exactly a smooth development experience. Likely you can improve it with the `<MappedFolder>` and `<LogonCommand>` options in your `.wsb` file.

PatchGuard & DSE

Running Windows Sandbox with a debugger attached will disable PatchGuard and with test signing enabled you can run your own kernel code. Attaching a debugger every time is not ideal though. Startup times are increased by a lot and software might detect kernel debugging and refuse to run. Additionally it seems that the network connection is not necessarily stable across host reboots and you need to restart WinDbg every time to attach the debugger to the sandbox.

Tooling similar to EfiGuard would be ideal for our purposes and in the rest of the post we will look at implementing our own bootkit with equivalent functionality.

Windows Sandbox internals recap

Back in March 2021 a great article called Playing in the (Windows) Sandbox came out. This article has a lot of information about the internals and a lot of the information below comes from there. Another good resource is Microsoft's official Windows Sandbox architecture page.

Windows Sandbox uses VHDx layering and NTFS magic to allow the VM to be extremely lightweight. Most of the system files are actually NTFS reparse points that point to the host file system. For our purposes the relevant file is `BaseLayer.vhdx` (more details in the references above).

What the article did not mention is that there is a folder called `BaseLayer` pointing directly inside the mounted `BaseLayer.vhdx` at the following path on the host:

```
C:\ProgramData\Microsoft\Windows\Containers\BaseImages\<GUID>\BaseLayer
```

This is handy because it allows us to read/write to the Windows Sandbox file system without having to stop/restart `CmService` every time we want to try something. The only catch is that you need to run as `TrustedInstaller` and you need to enable development mode to modify files there.

When you enable development mode there will also be an additional folder called `DebugLayer` in the same location. This folder exists on the host file system and allows us to overwrite certain files (`BCD` , registry hives) without having to modify the `BaseLayer` . The configuration for the `DebugLayer` appears to be in `BaseLayer\Bindings\Debug` , but no further time was spent investigating. The downside of enabling development mode is that snapshots are disabled and as a result startup times are significantly increased. After modifying something in the `BaseLayer` and disabling development mode you also need to delete the `Snapshots` folder and restart `CmService` to apply the changes.

Getting code execution at boot time

To understand how to get code execution at boot time you need some background on UEFI. We released Introduction to UEFI a few years back and there is also a very informative series called Geeking out with the UEFI boot manager that is useful for our purposes.

In our case it is enough to know that the firmware will try to load `EFI\Boot\bootx64.efi` from the default boot device first. You can override this behavior by setting the `BootOrder` UEFI variable. To find out how Windows Sandbox boots you can run the following PowerShell commands:

```
> Set-ExecutionPolicy -ExecutionPolicy Unrestricted
> Install-Module UEFI
> Get-UEFIVariable -VariableName BootOrder -AsByteArray
0
0
> Get-UEFIVariable -VariableName Boot0000
VMBus File SystemVMBus\EFI\Microsoft\Boot\bootmgfw.efi
```

From this we can derive that Windows Sandbox first loads:

```
\EFI\Microsoft\Boot\bootmgfw.efi
```

As described in the previous section we can access this file on the host (as `TrustedInstaller`) via the following path:

```
C:\ProgramData\Microsoft\Windows\Containers\BaseImages\
<GUID>\BaseLayer\Files\EFI\Microsoft\Boot\bootmgfw.efi
```

To verify our assumption we can rename the file and try to start Windows Sandbox. If you check in Process Monitor you will see `vmwp.exe` fails to open `bootmgfw.efi` and nothing happens after that.

Perhaps it is possible to modify UEFI variables and change `Boot0000` (Hyper-V Manager can do this for regular VMs so probably there is a way), but for now it will be easier to modify `bootmgfw.efi` directly.

Bootkit overview

To gain code execution we embed a copy of our payload inside `bootmgfw` and then we modify the entry point to our payload.

Our `EfiEntry` does the following:

- Get the image base/size of the currently running module
- Relocate the image when necessary
- Hook the `BootServices->OpenProtocol` function
- Get the original `AddressOfEntryPoint` from the `.bootkit` section
- Execute the original entry point

To simplify the injection of `SandboxBootkit.efi` into the `.bootkit` section we use the linker flags `/FILEALIGN:0x1000 /ALIGN:0x1000`. This sets the `FileAlignment` and `SectionAlignment` to `PAGE_SIZE`, which means the file on disk and in-memory are mapped one-to-one.

Bootkit hooks

Note: Many of the ideas presented here come from the DmaBackdoorHv project by Dmytro Oleksiuk, go check it out!

The first issue you run into when modifying `bootmgfw.efi` on disk is that the self integrity checks will fail. The function responsible for this is called `BmFwVerifySelfIntegrity` and it directly reads the file from the device (e.g. it does not use the UEFI `BootServices` API). To bypass this there are two options:

1. Hook `BmFwVerifySelfIntegrity` to return `STATUS_SUCCESS`
2. Use `bcdedit /set {bootmgr} nointegritychecks on` to skip the integrity checks. Likely it is possible to inject this option dynamically by modifying the `LoadOptions`, but this was not explored further

Initially we opted to use `bcdedit`, but this can be detected from within the sandbox so instead we patch `BmFwVerifySelfIntegrity`.

We are able to hook into `winload.efi` by replacing the boot services `OpenProtocol` function pointer. This function gets called by `EfiOpenProtocol`, which gets executed as part of `winload!BlInitializeLibrary`.

In the hook we walk from the return address to the `ImageBase` and check if the image exports `BlImgLoadPEImageEx`. The `OpenProtocol` hook is then restored and the `BlImgLoadPEImageEx` function is detoured. This function is nice because it allows us to modify `ntoskrnl.exe` right after it is loaded (and before the entry point is called).

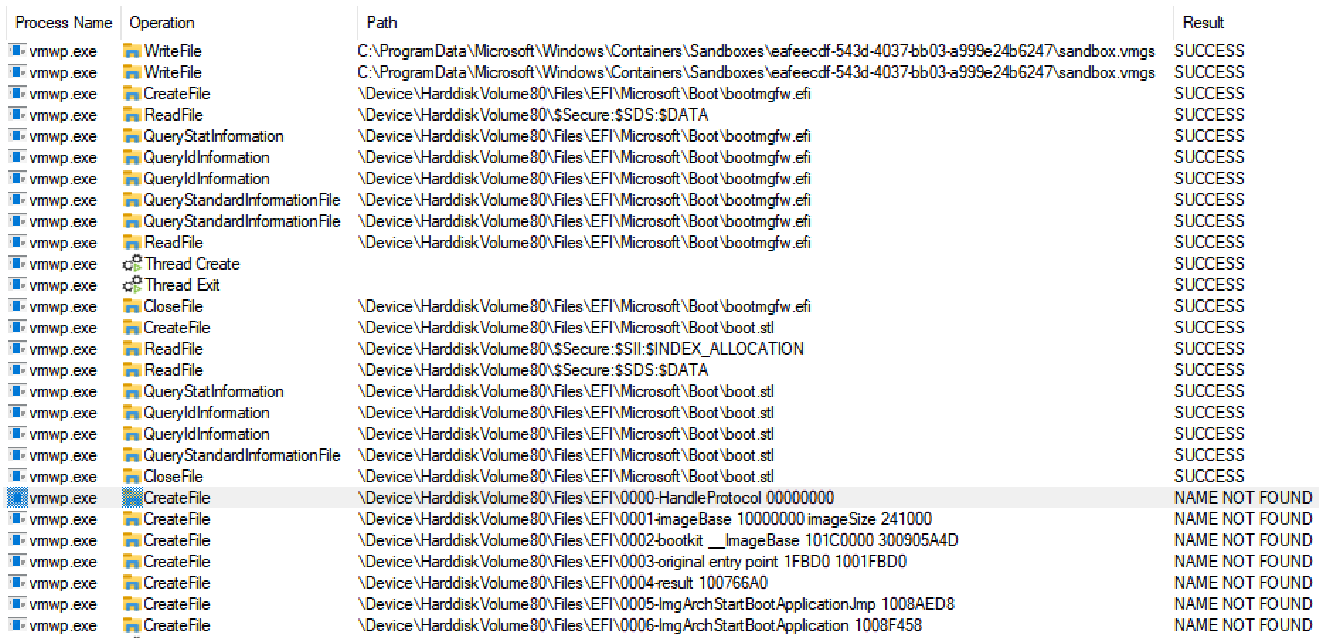
If we detect the loaded image is `ntoskrnl.exe` we call `HookNtoskrnl` where we disable PatchGuard and DSE. EfiGuard patches very similar locations so we will not go into much detail here, but here is a quick overview:

- Driver Signature Enforcement is disabled by patching the parameter to `CiInitialize` in the function `SepInitializeCodeIntegrity`
- PatchGuard is disabled by modifying the `KeInitAmd64SpecificState` initialization routine

Bonus: Logging from Windows Sandbox

To debug the bootkit on a regular Hyper-V VM there is a great guide by tansadat. Unfortunately there is no known way to enable serial port output for Windows Sandbox (please reach out if you know of one) and we have to find a different way of getting logs out.

Luckily for us Process Monitor allows us to see sandbox file system accesses (filter for `vmwp.exe`), which allows for a neat trick: accessing a file called `\EFI\my log string`. As long as we keep the path length under 256 characters and exclude certain characters this works great!



Process Name	Operation	Path	Result
vmwp.exe	WriteFile	C:\ProgramData\Microsoft\Windows\Containers\Sandboxes\eafeecdf-543d-4037-bb03-a999e24b6247\sandbox.vmgs	SUCCESS
vmwp.exe	WriteFile	C:\ProgramData\Microsoft\Windows\Containers\Sandboxes\eafeecdf-543d-4037-bb03-a999e24b6247\sandbox.vmgs	SUCCESS
vmwp.exe	CreateFile	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\bootmgfw.efi	SUCCESS
vmwp.exe	ReadFile	\Device\HarddiskVolume80\$Secure:$SDS:$DATA	SUCCESS
vmwp.exe	QueryStatInformation	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\bootmgfw.efi	SUCCESS
vmwp.exe	QueryIdInformation	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\bootmgfw.efi	SUCCESS
vmwp.exe	QueryIdInformation	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\bootmgfw.efi	SUCCESS
vmwp.exe	QueryStandardInformationFile	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\bootmgfw.efi	SUCCESS
vmwp.exe	QueryStandardInformationFile	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\bootmgfw.efi	SUCCESS
vmwp.exe	ReadFile	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\bootmgfw.efi	SUCCESS
vmwp.exe	Thread Create		SUCCESS
vmwp.exe	Thread Exit		SUCCESS
vmwp.exe	CloseFile	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\bootmgfw.efi	SUCCESS
vmwp.exe	CreateFile	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\boot.stl	SUCCESS
vmwp.exe	ReadFile	\Device\HarddiskVolume80\$Secure:$SII:$INDEX_ALLOCATION	SUCCESS
vmwp.exe	ReadFile	\Device\HarddiskVolume80\$Secure:$SDS:$DATA	SUCCESS
vmwp.exe	QueryStatInformation	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\boot.stl	SUCCESS
vmwp.exe	QueryIdInformation	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\boot.stl	SUCCESS
vmwp.exe	QueryIdInformation	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\boot.stl	SUCCESS
vmwp.exe	QueryStandardInformationFile	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\boot.stl	SUCCESS
vmwp.exe	CloseFile	\Device\HarddiskVolume80\Files\EFI\Microsoft\Boot\boot.stl	SUCCESS
vmwp.exe	CreateFile	\Device\HarddiskVolume80\Files\EFI\0000-HandleProtocol 00000000	NAME NOT FOUND
vmwp.exe	CreateFile	\Device\HarddiskVolume80\Files\EFI\0001-imageBase 10000000 imageSize 241000	NAME NOT FOUND
vmwp.exe	CreateFile	\Device\HarddiskVolume80\Files\EFI\0002-bootkit __ImageBase 101C0000 300905A4D	NAME NOT FOUND
vmwp.exe	CreateFile	\Device\HarddiskVolume80\Files\EFI\0003-original entry point 1FBD0 1001FBD0	NAME NOT FOUND
vmwp.exe	CreateFile	\Device\HarddiskVolume80\Files\EFI\0004-result 100766A0	NAME NOT FOUND
vmwp.exe	CreateFile	\Device\HarddiskVolume80\Files\EFI\0005-ImgArchStartBootApplicationJmp 1008AED8	NAME NOT FOUND
vmwp.exe	CreateFile	\Device\HarddiskVolume80\Files\EFI\0006-ImgArchStartBootApplication 1008F458	NAME NOT FOUND

A more primitive way of debugging is to just kill the VM at certain points to test if code is executing as expected:

```
void Die() {
    // At least one of these should kill the VM
    __fastfail(1);
    __int2c();
    __ud2();
    *(UINT8*)0xFFFFFFFFFFFFFFFFull = 1;
}
```

Bonus: Getting started with UEFI

The `SandboxBootkit` project only uses the headers of the EDK2 project. This might not be convenient when starting out (we had to implement our own `EfiQueryDevicePath` for instance) and it might be easier to get started with the VisualUefi project.

Final words

That is all for now. You should now be able to load a driver like TitanHide without having to worry about enabling test signing or disabling PatchGuard! With a bit of registry modifications you should also be able to load DTrace (or the more hackable implementation STrace) to monitor syscalls happening inside the sandbox.