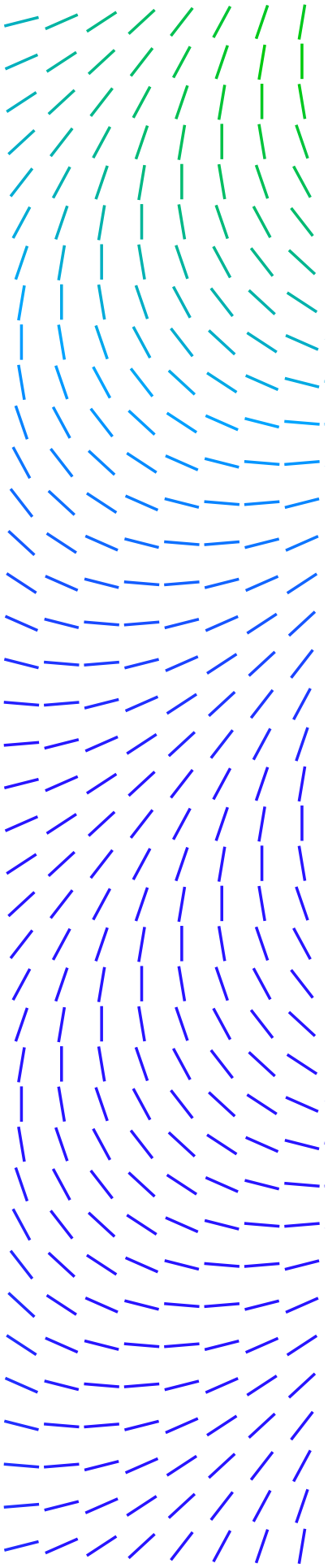


WHITE PAPER

Hyper-V Automation for Windows Patch Diffing

By: Kevin McGrath

Trellix

Table of Contents

I've often found that the best way to understand any given tool is to break it. Or at least see how other people broke it and try to understand that. In that spirit, this blog will walk through some automation developed to facilitate *patch diffing* -- the investigation of changes made by updates to a given library. This blog is a discussion of how to obtain the *before* and *after* files for a given patch, rather than the actual process of patch diffing. This is intentional, as the process of obtaining the files to investigate is often tedious, error-prone, and a frequent stumbling block.

Some assumptions are made:

- You are attempting this on an x86-64 based Windows machine that meets the requirements for Hyper-V (see below).
- You have at least one set of tools for binary diffing -- r2diaphora enables the free diaphora on r2 if you don't have a license for IDA-Pro.
- You can run PowerShell scripts as admin on your local machine. An admin PowerShell session is required due to the way Hyper-V modules work in PowerShell.
- You have some specific CVE you want to investigate and know the Knowledge Base (KB) ID of the patch.

After making use of this technique, you will have (at least) 2 files that you can compare -- what I call a pre-patch version and a post-patch version. One of the important considerations to keep in mind, if you are looking for a patch against a specific CVE, is that there will often be multiple DLLs impacted by the change...and not always the DLLs you might think.

What is patch diffing, and why would we do it?

Patch diffing is a technique in which you investigate a given CVE by looking at the binary in question both before and after the patch, looking for where the binary changed. It's an incredibly powerful approach to writing a proof-of-concept, but it's also used by malicious actors to weaponize a given vulnerability (if possible). This is possible primarily due to the fact that not everyone can or will patch, and once a patch is public, the cat is out of the bag, as it were.

This is a static analysis technique. You will not be comparing the running behavior of the code, but rather the raw opcodes which make up the binary. As mentioned above, there are many tools to help with this activity, and it's highly recommended that you leverage those -- sometimes the change is so small, it's easy to overlook.

This blog won't cover the actual diffing process as there are numerous tutorials available; instead, the focus is on an automated way to identify possible candidates for diffing, as well as making such files easily accessible to the tools you want to use.

Why Hyper-V?

Hyper-V is a type 1 hypervisor which runs bare metal on the host system (in essence, this means that the "host OS" is actually running within the hypervisor as well, albeit in a special fashion), and is freely available from Microsoft, so long as you are running Windows Professional, Enterprise, or Education[1,2].

As an aside, it should be noted that if you still want to run a type 2 hypervisor, such as VMware Workstation or VirtualBox, there are specific minimum versions required:

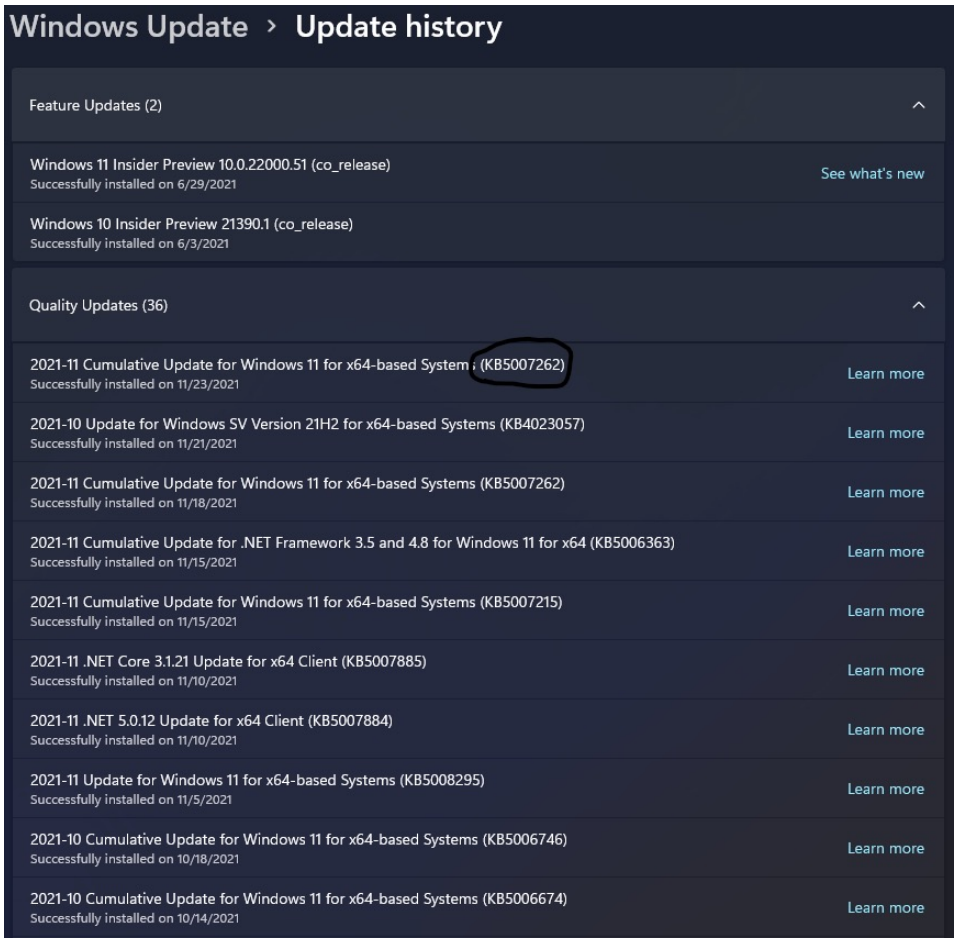
- VirtualBox 6+
- VMware Workstation 15.5.5+ In the case of VirtualBox, it will simply use the Hyper-V engine[3]. In the case of VMware, the change requires a move from a priviledged virtual machine monitor to a user-level monitor which leverages the Hyper-V API to run virtual machines[4].

Given that Hyper-V is freely available, and most importantly has a scriptable API from PowerShell, it seemed an ideal combination of tools that we could make something of. It's quite likely there is **qemu** equivalent on Linux, but I have not investigated that possibility, since I'm running Windows and I'm currently investigating Windows vulnerabilities.

Finding a patch to investigate

It's the second Tuesday of the month. You know what that means! Microsoft's Patch Tuesday is here! OK, maybe that isn't super exciting to people that aren't me; regardless, patches are available, and Microsoft publishes knowledge base (KB) articles about each and every patch set it deploys. Assuming you are reading this, you're interested in looking at what those patches actually *do*. So how can we figure that out?

We have a few ways. We can disable automatic patch installation and query the update servers to obtain the KB number of a patch we're interested in (maybe it touches on the kernel, or Hyper-V, or even MS Teams if that's your bag). We can also let the update install on our host machine, and then look at our update history, as seen here:



The circled value is the knowledge base entry which details this patch. So what can we use this value for?

Microsoft Update Catalog

This is where the Microsoft Update Catalog comes in. You can search by KB number, and it will find you the MSU (Microsoft update package) for the different variants of Windows (ARM64, ADM64). This site doesn't appear to have a usable scripting API, but you can just download the files manually and stick them somewhere safe.

OK, now we have the patch file that contains the fix we are interested in. Unfortunately, we can't just use that, as an MSU file is an archival format, typically containing of multiple CAB files, each of which contains some number of additional files. These additional files include manifests,

security catalogs, and several other files which we will mostly ignore. We are primarily interested in the new DLLs.

Except...there are many "DLL" files that aren't actually, you know, DLL files. They are too small, they are missing even a basic PE header, and IDA doesn't parse them as anything but raw binaries. Well, huh...there's definitely something going on here, what with these files always being in a **...\f**, **...\r**, or **...\n** folder, and often different sizes but all the same names. So, what's going on here?

Aside on Patch Formats

As seen in [5], there are 2 different forms of patches. Sometimes, MS will distribute full replacement DLL/EXE files for the component in question. Other times, it will make use of *deltas*, which are binary patches to the component. In an ideal world, after patch extraction, you would find the full DLL you need in **C:\Patches\MSU\x64** (or whatever path you choose to use when extracting). Unfortunately, this won't always be the case. Quite often, you will end up with a set of folders underneath a named component (of the form **...\<platform>_<component>_<checksum>**).

If you are wondering what the named-like-a-DLL-but-too-small-and-not-a-DLL files that you find in folders **...\f** and **...\r**, these are the delta files. There are three types:

- forward
- reverse
- null

Forward deltas move from a base binary (as shipped in a feature release of Windows) to the current version. Reverse deltas are exactly what they say on the tin: they take a current binary and revert it to the base version. Null deltas are essentially new files. While it is perfectly possible to write a Windows C++ program that makes use of the MSDELTA library, that would be tedious. And also, unnecessary. Fortunately for us, a GitHub user by the name of wumb0 has written a python script which leverages the MSDELTA library to apply a given pair of patches (forward and reverse) to a file, or to apply a null patch to obtain a new file.

If you are interested in obtaining the DLLs you're interested in without spinning up a new VM, for many components you can simply use the delta_patch.py script, included here for completeness (original source). You can obtain the deltas of applied patches in the **C:\Windows\WinSxS** directory, applying paired updates in a reverse-then-forward method to get to a specific patch level. Look at [5] for the full details on how you might want to do this.

OK, we know what we want to investigate. Now what?

The steps to investigate a given patch file are nearly always the same. We'll walk through the process of using a full MSU "by hand," then put it all together into an automation script.

The steps themselves are:

1. Build a Windows VM of the version of interest -- so if you want to investigate something that came out on Patch Tuesday of this month, snag the ISO from MSDN that was updated no later than last month.
2. Copy the .msu package of the given KB from the Microsoft Update Catalog. If you know the KB of the update you are interested in, you can search the catalog for it.
3. At this point, you need to extract the files from the patch file. You can do this manually with **C:\Windows\system32\expand.exe -F:* "C:\Patches\patch.msu"**, recursively as needed on any CAB files. That gets pretty old, and pretty cluttered. What I would suggest is making use of the PatchExtract.ps1 script. This is an interesting tool, if only for its provenance. It was originally released by Greg Linares (@Laughing_Mantis), possibly in a Twitter thread. Now, the only public sources I can find for it are other peoples' gits, and an article on working with patch files.[5]

```
Powershell -ExecutionPolicy Bypass -File C:\Patches\PatchExtract.ps1 -Patch C:\Patches\*.msu -Path
C:\Patches\MSU\
```

   This will create a series of folders for you within **C:\Patches\MSU**.

4. I'd suggest copying the entire contents of **C:\Patches\MSU\x64** to your host. This will give you the ability to manipulate the contents of that directory much more easily. There are a few ways you can do this:
   - You can copy/paste the whole directory from the VM, so long as you're connected via an "Enhanced Session"
   - You can mount a shared folder within the VM, and copy the files there
   - You can user the PowerShell Hyper-V API to move the files:

```
#copy the files from the VM to the host
Copy-Item -FromSession $s -Path "C:\Patches\MSU\x64" -Destination
"$patchPath\$CVE\patch_files\" -Recurse -ErrorAction SilentlyContinue
```

   This requires that you have a session **$s**, which can be created with:

```
> #create some credentials to log in -- in this case User/password (literally)
> $password = ConvertTo-SecureString "password" -AsPlainText -Force
> $cred = New-Object System.Management.Automation.PSCredential ("User", $password)
>
> #create a new session
> $s = New-PSSession -VMName $VMname -Credential $cred
```

   I'm sure there are more, as well, but this blog is about automation, after all.

5. In some fashion, you will need to generate a list of the DLL files which this update touches. To do it from within PowerShell on the host, you can run

```
> $dlls = (Get-ChildItem -Recurse -Filter *.dll -Path "$patchPath\$CVE\patch_files\" | Select-Object
-Property Name -Unique).Name
```

   These are my paths. Yours may differ.

6. Once you have the list of DLLs the update modifies, you can extract them from the VM. Again, there are a few ways to do this:
   - Copy *each and every one by hand*. In some updates, there are thousands. But you could do it, I suppose.
   - On the VM, use powershell to copy the full list to a new folder, then use one of the methods above to copy that folder to the host.
   - Or you can use more automation!

```
> $i = 0
> $base_path = "C:\Windows\System32"
> foreach ($dll in Get-Content ".\dlls.txt")
> {
>       Write-Progress -Activity "Copying DLLs" -ID 1 -Status 'Progress->' -PercentComplete
([int][Math]::Round(100 * $i / \ $dlls.count, [MidpointRounding]::AwayFromZero)) -
CurrentOperation "Copying $dll"
>
>     # suppress the copy-item preference dialog
>     $progresspreference = 'silentlyContinue'
>      Copy-Item -FromSession $s -Path $base_path\$dll -Destination $patchPath\$CVE\pre_patch -
ErrorAction SilentlyContinue
>     $progresspreference = 'Continue'
>     $i = $i + 1
> }
```

   It is worth pointing out, this assumes you have a session **$s** to work with. This snippet also displays a progress meter, based on the total number of DLLs we are attempting to copy, while silently ignoring missing files and any other errors. By setting **$progresspreference** to 'silentlyContinue' we eliminate the copy-file progress dialog, while still retaining our progress meter. **$patchPath** is the base path on your host where you want the DLLs while **$CVE** is exactly what it says. Now that you have the pre-patch DLLs copied to your host, it's time to apply the patch!

   - You can double click the .msu file
   - You can use the following command:

```
>wusa.exe C:\Patches\patch_file.msu /quiet /norestart
```

7. In either case, the patch will be installed. Now reboot the system.
8. Repeat the exercise above where you copy the DLLs, this time copying in to **$patchPath\$CVE\post_patch**.
9. As a final step, checksum each file in both **$patchPath\$CVE\pre_patch** and **$patchPath\$CVE\post_patch**, noting which DLLs actually changed. I put them in the **$patchPath\$CVE\dlls_of_interest** directory, suffixed by whether they are pre- or post-patch.
10. At this point, you are welcome to use whatever tool you prefer to actually diff the files!

OK, that's not too long a list. Let's do this! Do you have a cup of coffee? Maybe a sandwich or a piece of cake? Because you'll need at least that during this process.

1. Installing a VM to get to the right version takes a decent amount of time, especially if you have to download the ISO. Let's say 20 minutes, give or take.
2. Copying the MSU file takes under a minute, no big deal.
3. Extracting the MSU file...this can take a while. Large MSUs can easily take 20-30 minutes. Even smaller updates take a solid 5 minutes
4. Copying the patch files, give it a few minutes.
5. Copying the potentially modified DLLs can take anywhere from a few minutes to a few hours, depending on the approach you want. For a moderately sized update, even using the automation above, it takes 5-10 minutes.
6. Apply the update. 20-30 minutes, quite often.
7. Repeat step 5 for post-patch DLLs, another 5-10 minutes (at least)
8. Run the checksums, compare, and copy relavent DLLs to a new folder to make investigation easier. 2-3 minutes.
9. Total runtime: On the order of 75-90 minutes. Assuming you're paying attention and don't get distracted.

If this is something you do regularly, that's a lot of time, especially since it requires so much manual involvement. Turns out, we can remove that human component, which serves two purposes:

1. It removes the tedium of the process
2. It removes the common mistake points due to tedium.

There are a few places where manual intervention is still required, but they are minimized. This will be discussed in more detail below.

More Complete Automation

So how do we automate as much as possible? Turns out the majority of what we are doing can be invoked via the PowerShell CLI. Also, there's a PowerShell API which can be used to invoke commands on a remote session (the VM is the remote in this case).

Some more assumptions to bring up at this point:

- You have a base VM from which to clone. This means you have a VM that has had Windows installed, but little or nothing else done to it.
- You want to create a new VM for each CVE you are investigating. While not required, it is strongly encouraged.

OK, with that out of the way, how can we clone a VM from the command line? Below is a snippet of code which does just that. Comments are included in the code to explain the origin of variables, and what exactly this code does.

```
#create the directory where you want to store the new VM
if (!(Test-Path -Path "$VMPath")) {
    #PS native version of mkdir
    New-Item -ItemType "Directory" -Path $VMPath
}

#if the VM we want doesn't exist, create it
Write-Host "Cloning VM"
#check the list of known VMs for the one we need
if (! ((Get-VM).Name -contains $VMname)) {
    #import the VM, '-Copy -GenerateNewId' requires the import to create an entirely new VM, using the
copied VHDX file
    # $baseVMPath and $VMPath are command line arguments
    $VM = Import-VM -Path $baseVMPath -Copy -GenerateNewId -SnapshotFilePath "$VMPath" -VhdDestinationPath
"$VMPath" -VirtualMachinePath "$VMPath"

    #change the name of the VM to whatever we want it to be, and make checkpoints 'Standard' (see
https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/user-guide/checkpoints for details)
    Set-VM -VM $VM -NewVMName $VMname -CheckpointType Standard

    #move to the current generation VM
    Update-VMVersion -Name $VMname

    #configure processor count and memory requirements, as well as enabling paravirtualization
    Set-VMProcessor -VMName $VMname -Count 4 -ExposeVirtualizationExtensions $true
    Set-VMMemory -VMName $VMname -StartupBytes 8GB
}

#connect the VM to the network via the default switch -- change as necessary
Get-VMNetworkAdapter -VMName $VMname | Connect-VMNetworkAdapter -SwitchName 'Default Switch'

#check if VM running...
if (!((Get-VM -Name $VMname).State -eq 'Running')){
    #...and if not, start it!
    Start-VM -VMName $VMname
}
```

So, at this point, we have a new VM of the current virtualized hardware generation, configured to use 8GB of memory, 4 CPU cores, and the default switch for network access. If you make use of Hyper-V Manager (GUI application) you should see the new VM in the list of available VMs, and it should be in the "running" state.

While this isn't always required, for safety, this is a pause point which requires user interaction. Specifically, you need to log in to your shiny new VM and, if necessary, create a password for it. Regardless, the VM should be logged into, or (occasionally) the remote PS session will just…close. No explanation, just no connection, and therefore no actions within the automation will succeed from this point forward.

OK, you're logged in, you've obtained the MSU you want, now what? Well, this is where things get customizable. There are many features you can enable, software you can install, etc. It really depends on what you want to do with the VM. If you're looking to root cause a vulnerability, odds are good you want to have some development tools installed (**procmon, windbg, vscode, etc.**), but that all requires a lot of manual grunt work, doesn't it?

Well, not really, no. There are multiple tools out there which can help you with this step, such as choco, scoop, or winget. While the script presented here uses choco (mostly due to inertia), scoop and winget both bring some interesting features to the table. While choco requires administrator privileges, scoop installs in user mode. Winget is the new, native Windows package manager, and can install anything available in the Microsoft Store, as well as many other applications. Any of the three could be used, depending on whether they have the tools you wanted/needed on the VM.

And since we're all about doing this unattended, we can use the **Invoke-Command** commandlet from the Hyper-V API to invoke the package manager *from the host*.

```
#create some credentials so we can access the VM
$password = ConvertTo-SecureString "password" -AsPlainText -Force
$cred = New-Object System.Management.Automation.PSCredential ("User", $password)

#create a new session -- this basically lets you invoke all kinds of things on the VM
$s = New-PSSession -VMName $VMname -Credential $cred

$restartRequired = $false

#such as below! Invoke the commands you need for software installation
if ($choco -eq $true) {
    Invoke-Command -Session $s -ScriptBlock {
        Write-Host "Invoking commands on VM"
        Write-Host "Installing choco..."
        Set-ExecutionPolicy             Bypass             -Scope             Process             -Force;
[System.Net.ServicePointManager]::SecurityProtocol  =  [System.Net.ServicePointManager]::SecurityProtocol  -
bor                       3072;                       Invoke-Expression                       ((New-Object
System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))
        refreshenv
        choco install -y 7zip notepadplusplus chocolatey-core.extension powershell-core sysinternals python3
git microsoft-windows-terminal terminal-icons.powershell nerdfont-hack inconsolata firanf firefox firefox-
quantum-nox powertoys vscode
    }
}
```

To use **Invoke-Command**, you use the session created earlier, and then give it a **ScriptBlock**. This will be executed on the VM, as administrator, within PowerShell. This distinction is important, as that allows us to use native PowerShell in the **ScriptBlock**, rather than having to mix and match cmd commands with PowerShell. Also very important, the body of the **ScriptBlock** runs *on the VM, not the host*. For the most part, you can consider it to be very similar to a fully isolated scope: variables don't have their outside-the-**ScriptBlock** values, and no new variables will survive after the end of the **ScriptBlock**. Now, is this documented anywhere? Not that I could find -- this came out via a lot of experimentation with command line switches, setting values in the ScriptBlock that are magically not there, etc. Let my pain save you some trouble: assume nothing from the rest of the script exists within a **ScriptBlock** and you will find joy.

In this instance, there's a collection of tools installed on the VM, mostly development aids (git, notepad++, the new Terminal application, some others). Many VScode extensions are also available via choco **-- choco search vscode** will display the available options. **$choco** is a command

line switch to the larger script which defaults to **$false** to control whether you want tools installed or not. It does take a solid chunk of time and can be easily skipped if you prefer to do any and all analysis statically on your host.

Some other switches control other Windows optional features, specifically WSL2 and Hyper-V. See the full script for details on how this is done.

Finally, we have a fully configured VM, all outstanding reboots are dealt with (**StopVM** followed by a **StartVM**), and we are ready to handle the patch extraction.

```
#copy patch files to VM
Write-Host "Copying patch files..."
Copy-Item -Path $patchFile -Destination "C:\Patches\$(Split-Path -Path $patchFile -Leaf)" -ToSession $s
Copy-Item -Path $scriptPath -Destination "C:\Patches\PatchExtract.ps1" -ToSession $s

Write-Host "Running patch extract script..."
#extract patch files on VM
Invoke-Command -Session $s -ScriptBlock {
    New-Item -ItemType "Directory" -Path "C:\Patches\MSU"
    Powershell -ExecutionPolicy Bypass -File "C:\Patches\PatchExtract.ps1" -Patch "C:\Patches\$(Split-Path
-Path $patchFile -Leaf)" -Path C:\Patches\MSU
}
```

In order to get the files onto the VM, we make use of the **Copy-Item** commandlet, with a **-ToSession** parameter to signify directionality and the relative meaning of **-Path** (file source, from host) and **-Destination** (new location, on VM). Now that the files are there (in a location we know!), we can now run the **PatchExtract** script we talked about above. As mentioned, this will fully expand the MSU update so that we can determine all the DLLs which are touched by the update.

As mentioned, I like to pull the patch files over to the host machine, which can be done with a single command on the host:

```
Copy-Item -FromSession $s -Path "C:\Patches\MSU\x64" -Destination "$patchPath\$CVE\patch_files\" -Recurse -
ErrorAction SilentlyContinue
```

The reason for pulling the patch files to the host is so that we can quickly and easily generate a list of touched DLLs, then use that list to extract them from the VM onto our host.

```
#get the list -- $patchPath is on our host, `Select-Object` is used to pull out just the file names and
uniqify them
$dlls = (Get-ChildItem -Recurse -Filter *.dll -Path $patchPath\$CVE\patch_files | Select-Object -Property
Name -Unique).Name
$i = 0
$base_path = "C:\Windows\System32"

#copy each one, silently ignoring errors
foreach ($dll in $dlls)
{
    Write-Progress    -Activity    "Copying    DLLs"    -ID    1    -Status    'Progress->'    -PercentComplete
([int][Math]::Round(100 * $i / $dlls.count,[MidpointRounding]::AwayFromZero)) -CurrentOperation "Copying
$dll"
    #suppress progress output of actual copy operation
    $progresspreference = 'silentlyContinue'
    Copy-Item -FromSession $s -Path $base_path\$dll -Destination $patchPath\$CVE\pre_patch -ErrorAction
SilentlyContinue
    #re-enable progress so that our progress bar updates
    $progresspreference = 'Continue'
    $i = $i + 1
}
```

Now that we have all the pre-patched DLLs, we can use **Invoke-Command** once again to install the update:

```
Invoke-Command -Session $s -ScriptBlock {
    wusa.exe C:\Patches\*.msu /quiet /norestart
}
```

Follow this up with another reboot, and we are ready to pull all the post-patch DLLs from the VM. We do this exactly as above, simply changing the destination folder to be **post_patch**. This gives us the pre- and post-patch DLLs. Are they all the same? Let's check!

```
$dlls = (Get-ChildItem -Recurse -Filter *.dll -Path $patchPath\$CVE\post_patch\ | Select-Object -Property
Name -Unique).Name
foreach ($dll in $dlls)
{
    if (!(Test-Path -Path $patchPath\$CVE\pre_patch\$dll)) {
        Copy-Item -path $patchPath\post_patch\$dll -destination $patchPath/DLLs_of_interest/$(Split-Path -
Path $patchPath\$CVE\post_patch\$dll -LeafBase)_post.dll
        Write-Host "$dll copied to DLLs_of_interest"
        continue
    }
    if (!(Test-Path -Path $patchPath\$CVE\post_patch\$dll)) {
        Copy-Item               -path               $patchPath\$CVE\pre_patch\$dll               -destination
$patchPath\$CVE\DLLs_of_interest\$(Split-Path -Path $patchPath\$CVE\post_patch\$dll -LeafBase)_pre.dll
        Write-Host "$dll copied to DLLs_of_interest"
        continue
    }

    $pre = Get-FileHash -Algorithm SHA256 -Path $patchPath\$CVE\pre_patch\$dll
    $post = Get-FileHash -Algorithm SHA256 -Path $patchPath\$CVE\post_patch\$dll

    if ($pre.Hash -ne $post.Hash)
    {
        Write-Host "$dll copied to DLLs_of_interest"
        Copy-Item               -path               $patchPath\$CVE\pre_patch\$dll               -destination
$patchPath\$CVE\DLLs_of_interest\$(Split-Path -Path $patchPath\$CVE\post_patch\$dll -LeafBase)_pre.dll
        Copy-Item               -path              $patchPath\$CVE\post_patch\$dll               -destination
$patchPath\$CVE\DLLs_of_interest\$(Split-Path -Path $patchPath\$CVE\pre_patch\$dll -LeafBase)_post.dll
    }
}
```

While a given patch may touch thousands of DLLs in some fashion, a much smaller subset of all possible DLLs is actually modified in a way that their checksum changes. In order to construct this (typically much) smaller set of files to look at, we loop over the DLLs in the **post_patch** directory. If the file isn't in the **pre_patch** directory, it must be new, so add it to the **DLLs_of_interest** folder. Then we checksum both pre- and post-patch versions of a given DLL, and if the checksum differs, we move it to our **DLLs_of_interest** folder, suffixing the name of the DLL with its origin folder (**<DLL_basename>_pre.dll** or **<DLL_basename>_post.dll**).

Below you can see the entire script, combining all of the above snippets into a single script which can be invoked from an admin PowerShell session on the host.

It is quite possible this script can be modified for use with qemu, VirtualBox, or VMware workstation, but that's left as an exercise for the reader.

```powershell
#[CmdletBinding(PositionalBinding=$false)]
#This script can be used to perform the following operations:
#1. Clone a template VM to create a sandbox VM with specific VM settings
#2. The above, but also enable specific Windows features
#3. Extract a set of DLLs from a VM
#4. Patch a VM with a standalone MSU package -- get from Windows Update Catalog
#5. Extract patched DLLs from VM
#6. Make a folder with modified DLLs

param (
    [Parameter(Mandatory=$true)][string]$patchFile,                              #path to msu file we are
interested in
    [Parameter(Mandatory=$true)][string]$CVE,                                  #what CVE we're investigating
    [Parameter(Mandatory=$true)][string]$baseVMPath,                           #full path to base VM to clone
    [Parameter(Mandatory=$true)][string]$VMPath,                               #path to hold new VM clone
    [string]$scriptPath = "E:\HyperV\PatchExtract.ps1",                          #edit this to point to your
script
    [string]$VMname = "clone01",
    [string]$patchPath = "\\wsl.localhost\Ubuntu-20.04\home\dmcgrath\MSU",  #edit this to point to the base
directory of the destination of all files of interest ON HOST
    [switch]$HyperV = $false,
    [switch]$WSL = $false,
    [switch]$choco = $false
 )

#if you want to use the official MS dev VMs, uncomment the below, updating as necessary for expiration date
#expiration date 11/14/2021
# $Expiry = Get-Date -Month 11 -Day 14 -Year 2021 -Hour 00 -Minute 00 -Second 00

# #if we need a new image:
# if ($(Get-Date) -ge $Expiry -or Test-Path -Path "$VMPath") {
#     #download the file from URL
#     $ProgressPreference = 'silentlyContinue'
#     $VMURL = "https://aka.ms/windev_VM_hyperv"
#     Invoke-WebRequest -Uri $VMURL -OutFile "D:\HyperV\Dev-vm.zip"
#     $ProgressPreference = 'Continue'
#     Expand-Archive -Path "D:\HyperV\Dev-vm.zip" -DestinationPath $(Split-Path -Path $baseVMPath)
# }

#if directory doesn't exist, create it
if (!(Test-Path -Path "$VMPath")) {
    New-Item -ItemType "Directory" -Path $VMPath
}

#if it doesn't exist, create it
#E:\HyperV\MSU-VM\Virtual Machines\9502DC93-1634-49FE-AB2D-5AE78FE44E45.vmcx
Write-Host "Cloning VM"
if (! ((Get-VM).Name -contains $VMname)) {
    $VM = Import-VM -Path "$baseVMPath" -Copy -GenerateNewId -SnapshotFilePath "$VMPath" -VhdDestinationPath
"$VMPath" -VirtualMachinePath "$VMPath"
    Set-VM -VM $VM -NewVMName $VMname -CheckpointType Standard
    Update-VMVersion -Name $VMname
    Set-VMProcessor -VMName $VMname -Count 4 -ExposeVirtualizationExtensions $true
    Set-VMMemory -VMName $VMname -StartupBytes 8GB
}
Get-VMNetworkAdapter -VMName $VMname | Connect-VMNetworkAdapter -SwitchName 'Default Switch'

#check if VM running
if (!((Get-VM -Name $VMname).State -eq 'Running')){
    Start-VM -VMName $VMname
}

#tell user to create password on VM and request verification
Write-Host "Please create a password on the VM ('password' preferred) and press enter when you are done."
Write-Host "If you already have a password, please log in to the VM and press enter when you are done."
Read-Host

#create credentials for the VM -- edit as necessary if you didn't follow the preference from above

$password = ConvertTo-SecureString "password" -AsPlainText -Force
$cred = New-Object System.Management.Automation.PSCredential ("User", $password)

#create a new session
$s = New-PSSession -VMName $VMname -Credential $cred

$restartRequired = $false
```

```
#invoke the commands you need
if ($choco -eq $true) {
    Invoke-Command -Session $s -ScriptBlock {
        Write-Host "Invoking commands on VM"
        Write-Host "Installing choco..."
        Set-ExecutionPolicy             Bypass             -Scope             Process             -Force;
[System.Net.ServicePointManager]::SecurityProtocol  =  [System.Net.ServicePointManager]::SecurityProtocol  -
bor                        3072;                        Invoke-Expression                        ((New-Object
System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))
        refreshenv
        choco install -y 7zip notepadplusplus chocolatey-core.extension powershell-core sysinternals python3
git microsoft-windows-terminal terminal-icons.powershell nerdfont-hack inconsolata firanf firefox firefox-
quantum-nox powertoys
    }
}

if ($HyperV -eq $true) {
    $restartRequired = $true
    Invoke-Command -Session $s -ScriptBlock {
        #make sure required features are enabled -- edit as necessary
        Write-Host "Enabling Hyper-V features..."
        if (!((Get-WindowsOptionalFeature -online -FeatureName Microsoft-Hyper-V).State -eq 'Enabled')) {
            Enable-WindowsOptionalFeature -online -FeatureName Microsoft-Hyper-V -All -NoRestart
        }
        if   (!((Get-WindowsOptionalFeature   -online   -FeatureName   VirtualMachinePlatform).State   -eq
'Enabled')) {
            Enable-WindowsOptionalFeature -online -FeatureName VirtualMachinePlatform -All -NoRestart
        }
        if (!((Get-WindowsOptionalFeature -online -FeatureName HypervisorPlatform).State -eq 'Enabled')) {
            Enable-WindowsOptionalFeature -online -FeatureName HypervisorPlatform -All -NoRestart
        }
    }
}

if ($WSL -eq $true) {
    $restartRequired = $true
    Invoke-Command -Session $s -ScriptBlock {
        Write-Host "Enabling WSL features..."
        if (!((Get-WindowsOptionalFeature -online -FeatureName Microsoft-Windows-Subsystem-Linux).State -eq
'Enabled')) {
            Enable-WindowsOptionalFeature  -online  -FeatureName  Microsoft-Windows-Subsystem-Linux  -All  -
NoRestart
        }
        #install wsl2 with ubuntu-20.04
        wsl --install --distribution Ubuntu-20.04
        Write-Host "Invoking commands on VM complete"
    }
}

#disconnect the network adapter from the switch
Get-VMNetworkAdapter -VMName $VMname | Disconnect-VMNetworkAdapter

#reboot the VM if necessary
if ($restartRequired -eq $true) {
    Stop-VM -VMName $VMname
    Start-VM -VMName $VMname
    Remove-PSSession $s
    $password = ConvertTo-SecureString "password" -AsPlainText -Force
    $cred = New-Object System.Management.Automation.PSCredential ("User", $password)
    $s = New-PSSession -VMName $VMname -Credential $cred
    $restartRequired = $false

}

Invoke-Command -Session $s -ScriptBlock {
    New-Item -ItemType "Directory" -Path "C:\Patches\"
}


#copy patch files to VM
Write-Host "Copying patch files..."

Copy-Item -Path $patchFile -Destination "C:\Patches\$(Split-Path -Path $patchFile -Leaf)" -ToSession $s
Copy-Item -Path $scriptPath -Destination "C:\Patches\PatchExtract.ps1" -ToSession $s

Write-Host "Running patch extract script..."
#extract patch files on VM
Invoke-Command -Session $s -ScriptBlock {
    New-Item -ItemType "Directory" -Path "C:\Patches\MSU"
    Powershell -ExecutionPolicy Bypass -File "C:\Patches\PatchExtract.ps1" -Patch "C:\Patches\*.msu" -Path
C:\Patches\MSU
}
```

```
#copy patch files to host
Write-Host "Copying patch files to host..."
Copy-Item -FromSession $s -Path "C:\Patches\MSU\x64" -Destination "$patchPath\$CVE\patch_files\" -Recurse -
ErrorAction SilentlyContinue

#copy pre-patched DLLs from system32 to host
Write-Host "Copying pre-patched DLLs to host..."
$dlls = (Get-ChildItem -Recurse -Filter *.dll -Path "$patchPath\$CVE\patch_files\" | Select-Object -Property
Name -Unique).Name
$i = 0
$base_path = "C:\Windows\System32"
New-Item -ItemType "Directory" -Path "$patchPath\$CVE\pre_patch\"
foreach ($dll in $dlls)
{
    Write-Progress    -Activity    "Copying    DLLs"    -ID    1    -Status    'Progress->'    -PercentComplete
([int][Math]::Round(100 * $i / $dlls.count,[MidpointRounding]::AwayFromZero)) -CurrentOperation "Copying
$dll"
    #suppress progress output of actual copy operation
    $progresspreference = 'silentlyContinue'
    Copy-Item -FromSession $s -Path $base_path\$dll -Destination $patchPath\$CVE\pre_patch\ -ErrorAction
SilentlyContinue
    #re-enable progress so that our progress bar updates
    $progresspreference = 'Continue'
    $i = $i + 1
}

#apply patch -- wusa.exe .msu /quiet /norestart
Write-Host "Applying patch..."
Invoke-Command -Session $s -ScriptBlock {
    wusa.exe "C:\Patches\*.msu" /quiet /norestart
}

#reboot VM
Write-Host "Rebooting VM..."
Stop-VM -VMName $VMname
Start-VM -VMName $VMname

Remove-PSSession $s
$password = ConvertTo-SecureString "password" -AsPlainText -Force
$cred = New-Object System.Management.Automation.PSCredential ("User", $password)
$s = New-PSSession -VMName $VMname -Credential $cred


#copy patched DLLs from system32 to host
Write-Host "Copying patched DLLs to host..."
New-Item -ItemType "Directory" -Path "$patchPath\$CVE\post_patch\"
$i = 0
foreach ($dll in $dlls)
{
    Write-Progress    -Activity    "Copying    DLLs"    -ID    1    -Status    'Progress->'    -PercentComplete
([int][Math]::Round(100 * $i / $dlls.count,[MidpointRounding]::AwayFromZero)) -CurrentOperation "Copying
$dll"
    $progresspreference = 'silentlyContinue'
    Copy-Item -FromSession $s -Path $base_path\$dll -Destination $patchPath\$CVE\post_patch\ -ErrorAction
SilentlyContinue
    $progresspreference = 'Continue'
    $i = $i + 1
}

if (!(Test-Path -Path $patchPath\$CVE\DLLs_of_interest)) {
    New-Item -ItemType Directory -Path $patchPath\$CVE\DLLs_of_interest
}

#checksum all DLLs in pre- and post-patch folders, and copy those which differ to DLLs_of_interest folder
Write-Host "Comparing checksums and copying DLLs of interest..."
$dlls = (Get-ChildItem -Recurse -Filter *.dll -Path $patchPath\$CVE\post_patch\ | Select-Object -Property
Name -Unique).Name
foreach ($dll in $dlls)
{
    if (!(Test-Path -Path $patchPath\$CVE\pre_patch\$dll)) {
        Copy-Item -path $patchPath\post_patch\$dll -destination $patchPath/DLLs_of_interest/$(Split-Path -
Path $patchPath\$CVE\post_patch\$dll -LeafBase)_post.dll
        Write-Host "$dll copied to DLLs_of_interest"
        continue
    }
    if (!(Test-Path -Path $patchPath\$CVE\post_patch\$dll)) {
        Copy-Item              -path              $patchPath\$CVE\pre_patch\$dll              -destination
$patchPath\$CVE\DLLs_of_interest\$(Split-Path -Path $patchPath\$CVE\post_patch\$dll -LeafBase)_pre.dll
        Write-Host "$dll copied to DLLs_of_interest"
        continue
    }
```

```
    $pre = Get-FileHash -Algorithm SHA256 -Path $patchPath\$CVE\pre_patch\$dll
    $post = Get-FileHash -Algorithm SHA256 -Path $patchPath\$CVE\post_patch\$dll

    if ($pre.Hash -ne $post.Hash)
    {
        Write-Host "$dll copied to DLLs_of_interest"
        Copy-Item              -path              $patchPath\$CVE\pre_patch\$dll              -destination
$patchPath\$CVE\DLLs_of_interest\$(Split-Path -Path $patchPath\$CVE\post_patch\$dll -LeafBase)_pre.dll
        Copy-Item              -path              $patchPath\$CVE\post_patch\$dll              -destination
$patchPath\$CVE\DLLs_of_interest\$(Split-Path -Path $patchPath\$CVE\pre_patch\$dll -LeafBase)_post.dll
    }
}


Remove-PSSession $s
```


1. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-requirements
2. While you may be aware that WSL2 uses Hyper-V, it does so in a special fashion. More specifically, the Hyper-V *role* is the limiting factor to creating Hyper-V VMs on an unsupported OS variant.
3. https://docs.oracle.com/en/virtualization/virtualbox/6.0/admin/hyperv-support.html
4. https://blogs.vmware.com/workstation/2020/05/vmware-workstation-now-supports-hyper-v-mode.html
5. https://wumb0.in/extracting-and-diffing-ms-patches-in-2020.html

Trellix

Copyright © 2022 Musarubra US LLC
JANUARY 2022