DirtyMoe: Rootkit Driver

decoded.avast.io/martinchlumecky/dirtymoe-rootkit-driver/

August 11, 2021

by Martin ChlumeckýAugust 11, 202138 min read

Abstract

In the first post DirtyMoe: Introduction and General Overview of Modularized Malware, we have described one of the complex and sophisticated malware called DirtyMoe. The main observed roles of the malware are Cryptojacking and DDoS attacks that are still popular. There is no doubt that malware has been released for profit, and all evidence points to Chinese territory. In most cases, the PurpleFox campaign is used to exploit vulnerable systems where the exploit gains the highest privileges and installs the malware via the MSI installer. In short, the installer misuses Windows System Event Notification Service (SENS) for the malware deployment. At the end of the deployment, two processes (workers) execute malicious activities received from well-concealed C&C servers.

As we mentioned in the first post, every good malware must implement a set of protection, anti-forensics, anti-tracking, and anti-debugging techniques. One of the most used techniques for hiding malicious activity is using rootkits. In general, the main goal of the rootkits is to hide itself and other modules of the hosted malware on the kernel layer. The rootkits are potent tools but carry a high risk of being detected because the rootkits work in the kernel-mode, and each critical bug leads to BSoD.

The primary aim of this next article is to analyze rootkit techniques that DirtyMoe uses. The main part of this study examines the functionality of a DirtyMoe driver, aiming to provide complex information about the driver in terms of static and dynamic analysis. The key techniques of the DirtyMoe rootkit can be listed as follows: the driver can hide itself and other malware activities on kernel and user mode. Moreover, the driver can execute commands received from the user-mode under the kernel privileges. Another significant aspect of the driver is an injection of an arbitrary DLL file into targeted processes. Last but not least is the driver's functionality that censors the file system content. In the same way, we describe the refined routine that deploys the driver into the kernel and which anti-forensic method the malware authors used.

Another essential point of this research is the investigation of the driver's meta-data, which showed that the driver is code-signed with the certificates that have been stolen and revoked in the past. However, the certificates are widespread in the wild and are misused in other malicious software in the present.

Finally, the last part summarises the rootkit functionally and draws together the key findings of digital certificates, making a link between DirtyMoe and other malicious software. In addition, we discuss the implementation level and sources of the used rootkit techniques.

1. Sample

The subject of this research is a sample with SHA-256:
`AABA7DB353EB9400E3471EAAA1CF0105F6D1FAB0CE63F1A2665C8BA0E8963A05`
The sample is a windows driver that DirtyMoe drops on the system startup.

Note: VirusTotal keeps a record of 44 of 71 AV engines (62 %) which detect the samples as malicious. On the other hand, the DirtyMoe DLL file is detected by 86 % of registered AVs. Therefore, the detection coverage is sufficient since the driver is dumped from the DLL file.

Basic Information

- File Type: Portable Executable 64
- File Info: Microsoft Visual C++ 8.0 (Driver)
- File Size: 116.04 KB (118824 bytes)
- Digital Signature: Shanghai Yulian Software Technology Co., Ltd. (上海域联软件技术有限公司)

Imports

The driver imports two libraries `FltMgr` and `ntosrnl`. **Table 1** summarizes the most suspicious methods from the driver's point.

Routine	Description
`FltSetCallbackDataDirty`	A minifilter driver's pre or post operation calls the routine to indicate that it has modified the contents of the callback data structure.
`FltGetRequestorProcessId`	Routine returns the process ID for the process requested for a given I/O operation.
`FltRegisterFilter`	FltRegisterFilter registers a minifilter driver.

`ZwDeleteValueKey`
`ZwSetValueKey`
`ZwQueryValueKey`
`ZwOpenKey`
	Routines delete, set, query, and open registry entries in kernel-mode.

`ZwTerminateProcess`	Routine terminates a process and all of its threads in kernel-mode.
`ZwQueryInformationProcess`	Retrieves information about the specified process.

`MmGetSystemRoutineAddress`	Returns a pointer to a function specified by a routine parameter.
`ZwAllocateVirtualMemory`	Reserves a range of application-accessible virtual addresses in the specified process in kernel-mode.

Table 1. Kernel methods imported by the DirtyMoe driver

At first glance, the driver looks up kernel routine via `MmGetSystemRoutineAddress()` as a form of obfuscation since the string table contains routine names operating with `VirtualMemory` ; e.g., `ZwProtectVirtualMemory()` , `ZwReadVirtualMemory()` , `ZwWriteVirtualMemory()` . The kernel-routine `ZwQueryInformationProcess()` and strings such as `services.exe` , `winlogon.exe` point out that the rootkit probably works with kernel structures of the critical windows processes.

2. DirtyMoe Driver Analysis

The DirtyMoe driver does not execute any specific malware activities. However, it provides a wide scale of rootkit and backdoor techniques. The driver has been designed as a service support system for the DirtyMoe service in the user-mode.

The driver can perform actions originally needed with high privileges, such as writing a file into the system folder, writing to the system registry, killing an arbitrary process, etc. The malware in the user-mode just sends a defined control code, and data to the driver and it provides higher privilege actions.

Further, the malware can use the driver to hide some records helping to mask malicious activities. The driver affects the system registry, and can conceal arbitrary keys. Moreover, the system process `services.exe` is patched in its memory, and the driver can exclude arbitrary services from the list of running services. Additionally, the driver modifies the kernel structures recording loaded drivers, so the malware can choose which driver is visible or not. Therefore, the malware is active, but the system and user cannot list the malware records using standard API calls to enumerate the system registry, services, or loaded drivers. The malware can also hide requisite files stored in the file system since the driver implements a mechanism of the minifilter. Consequently, if a user requests a record from the file system, the driver catches this request and can affect the query result that is passed to the user.

The driverconsists of 10 main functionalities as **Table 2** illustrates.

Function	Description
Driver Entry	routine called by the kernel when the driver is loaded.
Start Routine	is run as a kernel thread restoring the driver configuration from the system registry.

Device Control	processes system-defined I/O control codes (IOCTLs) controlling the driver from the user-mode.
Minifilter Driver	routine completes processing for one or more types of I/O operations; *QueryDirectory* in this case. In other words, the routine filters folder enumerations.
Thread Notification	routine registers a driver-supplied callback that is notified when a new thread is created.
Callback of NTFS Driver	wraps the original callback of the NTFS driver for `IRP_MJ_CREATE` major function.
Registry Hiding	is hook method provides registry key hiding.
Service Hiding	is a routine hiding a defined service.
Driver Hiding	is a routine hiding a defined driver.
Driver Unload	routine is called by kernel when the driver is unloaded.

Table 2. Main driver functionality
Most of the implemented functionalities are available as public samples on internet forums. The level of programming skills is different for each driver functionality. It seems that the driver author merged the public samples in most cases. Therefore, the driver contains a few bugs and unused code. The driver is still in development, and we will probably find other versions in the wild.

2.1 Driver Entry

The *Driver Entry* is the first routine that is called by the kernel after driver loading. The driver initializes a large number of global variables for the proper operation of the driver. Firstly, the driver detects the OS version and setups required offsets for further malicious use. Secondly, the variable for pointing of the *driver image* is initialized. The *driver image* is used for hiding a driver. The driver also associates the following major functions:

1. `IRP_MJ_CREATE` , `IRP_MJ_CLOSE` – no interest action,
2. `IRP_MJ_DEVICE_CONTROL` – used for driver configuration and control,
3. `IRP_MJ_SHUTDOWN` – writes malware-defined data into the disk and registry.

The *Driver Entry* creates a symbolic link to the driver and tries to associate it with other malicious monitoring or filtering callbacks. The first one is a minifilter activated by the `FltRegisterFilter()` method registering the `FltPostOperation()` ; it filters access to the system drives and allows it to hide files and directories.

Further, the initialization method swaps a major function `IRP_MJ_CREATE` for `\FileSystem\Ntfs` driver. So, each API call of `CreateFile()` or a kernel-mode function `IoCreateFile()` can be monitored and affected by the malicious `MalNtfsCreatCallback()` callback.

Another *Driver Entry* method sets a callback method using `PsSetCreateThreadNotifyRoutine()`. The `NotifyRoutine()` monitors a kernel process creation, and the malware can inject malicious code into newly created processes/threads.

Finally, the driver tries to restore its configuration from the system registry.

2.2 Start Routine

The *Start Routine* is run as a kernel system thread created in the Driver Entry routine. The *Start Routine* writes the driver version into the SYSTEM registry as follows:

- Key: `HKLM\SYSTEM\CurrentControlSet\Control\WinApi\WinDeviceVer`
- Value: `20161122`

If the following `SOFTWARE` registry key is present, the driver loads artifacts needed for the process injecting:

`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Installer\Secure`

The last part of *Start Routine* loads the rest of the necessary entries from the registry. The complete list of the system registry is documented in Appendix A.

2.3 Device Control

The device control is a mechanism for controlling a loaded driver. A driver receives the `IRP_MJ_DEVICE_CONTROL` I/O control code (IOCTL) if a user-mode thread calls Win32 API `DeviceIoControl()`; visit [1] for more information. The user-mode application sends `IRP_MJ_DEVICE_CONTROL` directly to a specific device driver. The driver then performs the corresponding operation. Therefore, malicious user-mode applications can control the driver via this mechanism.

The driver supports approx. 60 control codes. We divided the control codes into 3 basic groups: **controlling**, **inserting**, and **setting**.

Controlling

There are 9 main control codes invoking driver functionality from the user-mode. The following **Table 3** summarizes controlling IOCTL that can be sent by malware using the Win32 API.

IOCTL	Description
0x222C80	The driver accepts other IOCTLs only if the driver is activated. Malware in the user-mode can activate the driver by sending this IOCTL and authorization code equal `0xB6C7C230`.

0x2224C0	The malware sends data which the driver writes to the system registry. A key, value, and data type are set by **Setting** control codes. *used variable: regKey, regValue, regData, regType*
0x222960	This IOCTL clears all data stored by the driver. *used variable: see **Setting**and **Inserting**variables*
0x2227EC	If the malware needs to hide a specific driver, the driver adds a specific driver name to the *listBaseDllName* and hides it using Driver Hiding.
0x2227E8	The driver adds the name of the registry key to the *WinDeviceAddress* list and hides this key using Registry Hiding. *used variable: WinDeviceAddress*
0x2227F0	The driver hides a given service defined by the name of the DLL image. The name is inserted into the *listServices* variable, and the Service Hiding technique hides the service in the system.
0x2227DC	If the malware wants to deactivate the Registry Hiding, the driver restores the original kernel `GetCellRoutine()`.
0x222004	The malware sends a process ID that wants to terminate. The driver calls kernel function `ZwTerminateProcess()` and terminates the process and all of its threads regardless of malware privileges.
0x2224C8	The malware sends data which driver writes to the file defined by *filePath* variable; see **Setting** control codes *used variable: filePath, fileData*

Table 3. Controlling IOCTLs

Inserting

There are 11 control codes inserting items into white/blacklists. The following **Table 4** summarizes variables and their purpose.

White/Black list	Variable	Purpose
Registry HIVE	WinDeviceAddress	Defines a list of registry entries that the malware wants to hide in the system.
Process Image File Name	WinDeviceMaker	Represents a whitelist of processes defined by *process image file name*. It is used in Callback of NTFS Driver, and grants access to the NTFS file systems. Further, it operates in Minifilter Driver and prevents hiding files defined in the *WinDeviceNumber* variable. The last use is in Registry Hiding; the malware does not hide registry keys for the whitelisted processes.

	WinDeviceMakerB	Defines a whitelist of processes defined by *process image file name*. It is used in Callback of NTFS Driver, and grants access to the NTFS file systems.
	WinDeviceMakerOnly	Specifies a blacklist of processes defined by the *process image file name*. It is used in Callback of NTFS Driver and refuses access to the NTFS file systems.
File names (*full path*)	WinDeviceName WinDeviceNameB	Determines a whitelist of files that should be granted access by Callback of NTFS Driver. It is used in combination with *WinDeviceMaker* and *WinDeviceMakerB*. So, if a file is on the whitelist and a requested process is also whitelisted, the driver grants access to the file.
	WinDeviceNameOnly	Defines a blacklist of files that should be denied access by Callback of NTFS Driver. It is used in combination with *WinDeviceMakerOnly*. So, if a file is on the blacklist and a requesting process is also blacklisted, the driver refuses access to the file.
File names (*containing number*)	WinDeviceNumber	Defines a list of files that should be hidden in the system by Minifilter Driver. The malware uses a name convention as follows: `[A-Z][a-z][0-9]+\.[a-z]{3}`. So, a file name includes a number.
Process ID	ListProcessId1	Defines a list of processes requiring access to NTFS file systems. The malware does not restrict the access for these processes; see Callback of NTFS Driver.
	ListProcessId2	The same purpose as *ListProcessId1*. Additionally, it is used as the whitelist for the registry hiding, so the driver does not restrict access. The Minifilter Driver does not limit processes in this list.
Driver names	listBaseDllName	Defines a list of drivers that should be hidden in the system; see Driver Hiding.
Service names	listServices	Specifies a list of services that should be hidden in the system; see Service Hiding.

Table 4. White and Black lists

Setting

The setting control codes store scalar values as a global variable. The following **Table 5** summarizes and groups these variables and their purpose.

Function	Variable	Description

File Writing (*Shutdown*)	filename1_for_ShutDown data1_for_ShutDown	Defines a file name and data for the first file written during the driver shutdown.
	filename2_for_ShutDown data2_for_ShutDown	Defines a file name and data for the second file written during the driver shutdown.
Registry Writing (*Shutdown*)	regKey1_shutdown regValue1_shutdown regData1_shutdown regType1	Specifies the first registry key path, value name, data, and type (REG_BINARY, REG_DWORD, REG_SZ, etc.) written during the driver shutdown.
	regKey2_shutdown regValue2_shutdown regData2_shutdown regType2	Specifies the second registry key path, value name, data, and type (REG_BINARY, REG_DWORD, REG_SZ, etc.) written during the driver shutdown.
File Data Writing	filePath	Determines filename which will be used to write data; see **Controlling**IOCTL `0x2224C8` .
Registry Writing	regKey regValue regType	Specifies registry key path, value name, and type (REG_BINARY, REG_DWORD, REG_SZ, etc.); see **Controlling**IOCTL `0x2224C0` .
Unknow (*unused*)	dwWinDevicePathA dwWinDeviceDataA	Keeps a path and data for file A.
	dwWinDevicePathB dwWinDeviceDataB	Keeps a path and data for file B.

Table 5. Global driver variables

The following **Table 6** summarizes variables used for the process injection; see Thread Notification.

Function	Variable	Description
Process to Inject	dwWinDriverMaker2 dwWinDriverMaker2_2	Defines two command-line arguments. If a process with one of the arguments is created, the driver should inject the process.
	dwWinDriverMaker1 dwWinDriverMaker1_2	Defines two process names that should be injected if the process is created.
DLL to Inject	dwWinDriverPath1 dwWinDriverDataA	Specifies a file name and data for the process injection defined by *dwWinDriverMaker2* or *dwWinDriverMaker1*.
	dwWinDriverPath1_2 dwWinDriverDataA_2	Defines a file name and data for the process injection defined by *dwWinDriverMaker2_2* or *dwWinDriverMaker1_2*.
	dwWinDriverPath2 dwWinDriverDataB	Keeps a file name and data for the process injection defined by *dwWinDriverMaker2* or *dwWinDriverMaker1*.

	dwWinDriverPath2_2 dwWinDriverDataB_2	Specifies a file name and data for the process injection defined by *dwWinDriverMaker2_2* or *dwWinDriverMaker1_2*.

Table 6. Injection variables

2.4 Minifilter Driver

The minifilter driver is registered in the Driver Entry routine using the `FltRegisterFilter()` method. One of the method arguments defines configuration (`FLT_REGISTRATION`) and callback methods (`FLT_OPERATION_REGISTRATION`). If the *QueryDirectory* system request is invoked, the malware driver catches this request and processes it by its `FltPostOperation()` .

The `FltPostOperation()` method can modify a result of the *QueryDirectory* operations (IRP). In fact, the malware driver can affect (hide, insert, modify) a directory enumeration. So, some applications in the user-mode may not see the actual image of the requested directory.

The driver affects the *QueryDirectory* results only if a requested process is not present in whitelists. There are two whitelists. The first whitelists (*Process ID* and *File names*) cause that the *QueryDirectory* results are not modified if the process ID or process image file name, requesting the given I/O operation (*QueryDirectory*), is present in the whitelists. It represents malware processes that should have access to the file system without restriction. The further whitelist is called *WinDeviceNumber*, defining a set of suffixes. The `FltPostOperation()` iterates each item of the *QueryDirectory*. If the enumerated item name has a suffix defined in the whitelist, the driver removes the item from the *QueryDirectory* results. It ensures that the whitelisted files are not visible for non-malware processes [2]. So, the driver can easily hide an arbitrary directory/file for the user-mode applications, including `explorer.exe` . The name of the *WinDeviceNumber* whitelist is probably derived from malware file names, e.g, `Ke145057.xsl` , since the suffix is a number; see Appendix B.

2.5 Callback of NTFS Driver

When the driver is loaded, the Driver Entry routine modifies the system driver for the NTFS filesystem. The original callback method for the `IRP_MJ_CREATE` major function is replaced by a malicious callback `MalNtfsCreatCallback()` as **Figure 1** illustrates. The malicious callback determines what should gain access and what should not.

```c
char UpdateNtfsCreateCallback()
{
  UNICODE_STRING ntfsDriver;
  DRIVER_OBJECT *Object;
  int *major_function_callback;

  major_function_callback = 0;
  if ( !OriginalNtfsCreateCallback )
    major_function_callback = &OriginalNtfsCreateCallback;
  RtlInitUnicodeString(&ntfsDriver, L"\\FileSystem\\Ntfs");
  ObReferenceObjectByName(&ntfsDriver, 0x40, 0, 0, IoDriverObjectType, 0, 0, (PVOID *)&Object);
  RewriteMajorFunctionCallback(&ntfsDriver, IRP_MJ_CREATE, MalNtfsCreatCallback, major_function_callback);
  return 1;
}
```

Figure 1. Rewrite IRP_MJ_CREATE callback of the regular NTFS driver

The malicious callback is active only if the Minifilter Driver registration has been done and the original callback has been replaced. There are whitelists and one blacklist. The whitelists store information about allowed *process image names*, *process ID*, and allowed *files*. If the process requesting the disk access is whitelisted, then the requested file must also be on the white list. It is double protection. The blacklist is focused on *processing image names*. Therefore, the blacklisted processes are denied access to the file system. **Figure 2** demonstrates the whitelisting of processes. If a process is on the whitelist, the driver calls the original callback; otherwise, the request ends with access denied.

```c
// white list of ProcessId (PID)
for ( k = 0; k < listProcessId_lenght; ++k )
{
  if ( CurrentProcessId == listProcessId[k] )
  {
    for ( m = 0; m < dWinDeviceName; ++m )
    {
      if ( string_compare(
             listWinDeviceName[m],
             CurrentWinDeviceName->FileName.Buffer,
             CurrentWinDeviceName->FileName.Length / 2) )
      {
        return OriginalNtfsCreateCallback(deviceObject, IRP);
      }
    }
  }
}
IRP->IoStatus.Status = STATUS_ACCESS_DENIED;
IRP->IoStatus.Information = 0;
status = IRP->IoStatus.Status;
IofCompleteRequest(IRP, 0);
return status;
```

Figure 2. Grant access to whitelisted processes

In general, if the malicious callback determines that the requesting process is authorized to access the file system, the driver calls the original `IRP_MJ_CREATE` major function. If not, the driver finishes the request as `STATUS_ACCESS_DENIED`.

2.6 Registry Hiding

The driver can hide a given registry key. Each manipulation with a registry key is hooked by the kernel method `GetCellRoutine()`. The configuration manager assigns a control block for each open registry key. The control block (`CM_KEY_CONTROL_BLOCK`) structure keeps all control blocks in a hash table to quickly search for existing control blocks. The `GetCellRoutine()`

hook method computes a memory address of a requested key. Therefore, if the malware driver takes control over the `GetCellRoutine()`, the driver can filter which registry keys will be visible; more precisely, which keys will be searched in the hash table.

The malware driver finds an address of the original `GetCellRoutine()` and replaces it with its own malicious hook method, `MalGetCellRoutine()`. The driver uses well-documented implementation [3, 4]. It just goes through kernel structures obtained via the `ZwOpenKey()` kernel call. Then, the driver looks for `CM_KEY_CONTROL_BLOCK`, and its associated HHIVE structured correspond with the requested key. The HHIVE structure contains a pointer to the `GetCellRoutine()` method, which the driver replaces; see **Figure 3**.

```
//  Hide Registry key
case 0x2227E8:
    Handle = OpenKey(keyToHide);
    if ( Handle )
    {
      // Get Control Block Handle
      ControBlock = GetKeyControlBlock(Handle);
      ZwClose(Handle);
      if ( ControBlock )
      {
        // Get HIVE
        hive = *(HHIVE **)&ControBlock->Keys[KeyHiveOffset];
        OriginalCellRoutine = hive->GetCellRoutine;
        hive->GetCellRoutine = (PGET_CELL_ROUTINE) MalGetCellRoutine;
      }
    }
    return_state = 0;
    goto endproc;

...

endproc:
// Process IRP
if ( return_state )
  irp->IoStatus.Information = 0;
else
  irp->IoStatus.Information = OutputBufferLength;
  irp->IoStatus.Status = return_state;
  IofCompleteRequest(irp, 0);
return return_state;
```

Figure 3. Overwriting GetCellRoutine

This method's pitfall is that offsets of looked structure, variable, or method are specific for each windows version or build. So, the driver must determine on which Windows version the driver runs.

The `MalGetCellRoutine()` hook method performs 3 basic operations as follow:

1. The driver calls the original kernel `GetCellRoutine()` method.
2. Checks whitelists for a requested registry key.
3. Catches or releases the requested registry key according to the whitelist check.

Registry Key Hiding

The hiding technique uses a simple principle. The driver iterates across a whole HIVE of a requested key. If the driver finds a registry key to hide, it returns the last registry key of the iterated HIVE. When the interaction is at the end of the HIVE, the driver does not return the last key since it was returned before, but it just returns NULL, which ends the HIVE searching.

The consequence of this principle is that if the driver wants to hide more than one key, the driver returns the last key of the searched HIVE more times. So, the final results of the registry query can contain duplicate keys.

Whitelisting

The `services.exe` and `System` services are whitelisted by default, and there is no restriction. Whitelisted *process image names* are also without any registry access restriction.

A decision-making mechanism is more complicated for Windows 10. The driver hides the request key only for `regedit.exe` application if the Windows 10 build is 14393 (July 2016) or 15063 (March 2017).

2.7 Thread Notification

The main purpose of the *Thread Notification* is to inject malicious code into created threads. The driver registers a thread notification routine via `PsSetCreateThreadNotifyRoutine()` during the Device Entry initialization. The routine registers a callback which is subsequently notified when a new thread is created or deleted. The suspicious callback (`PCREATE_THREAD_NOTIFY_ROUTINE`) `NotifyRoutine()` takes three arguments: *ProcessID*, *ThreadID*, and *Create* flag. The driver affects only threads in which *Create flag* is set to TRUE, so only newly created threads.

The whitelisting is focused on two aspects. The first one is an *image name*, and the second one is *command-line arguments* of a created thread. The *image name* is stored in *WinDriverMaker1*, and arguments are stored as a checksum in the *WinDriverMaker2* variable. The driver is designed to inject only two processes defined by a *process name* and two processes defined by *command line arguments*. There are no whitelists, just 4 global variables.

2.7.1 **Kernel Method Lookup**

The successful injection of the malicious code requires several kernel methods. The driver does not call these methods directly due to detection techniques, and it tries to obfuscate the required method. The driver requires the following kernel methods: `ZwReadVirtualMemory` , `ZwWriteVirtualMemory` , `ZwQueryVirtualMemory` , `ZwProtectVirtualMemory` , `NtTestAlert` , `LdrLoadDll`

The kernel methods are needed for successful thread injection because the driver needs to read/write process data of an injected thread, including program instruction.

Virtual Memory Method Lookup

The driver gets the address of the `ZwAllocateVirtualMemory()` method. As **Figure 4** illustrates, all lookup methods are consecutively located after `ZwAllocateVirtualMemory()` method in `ntdll.dll` .

```
.text:C910 ZwAllocateVirtualMemory
.text:C910                 mov     r10, rcx        ; NtAllocateVirtualMemory
.text:C913                 mov     eax, 18h
.text:C918                 test    byte ptr ds:7FFE0308h, 1
.text:C920                 jnz     short loc_18009C925
.text:C922                 syscall                 ; Low latency system call
.text:C924                 retn

.text:CA70 ZwQueryVirtualMemory
.text:CA70                 mov     r10, rcx        ; NtQueryVirtualMemory
.text:CA73                 mov     eax, 23h
.text:CA78                 test    byte ptr ds:7FFE0308h, 1
.text:CA80                 jnz     short loc_18009CA85
.text:CA82                 syscall                 ; Low latency system call
.text:CA84                 retn

.text:CD50 NtWriteVirtualMemory
.text:CD50                 mov     r10, rcx        ; NtWriteVirtualMemory
.text:CD53                 mov     eax, 3Ah
.text:CD58                 test    byte ptr ds:7FFE0308h, 1
.text:CD60                 jnz     short loc_18009CD65
.text:CD62                 syscall                 ; Low latency system call
.text:CD64                 retn

.text:D010 ZwProtectVirtualMemory
.text:D010                 mov     r10, rcx        ; NtProtectVirtualMemory
.text:D013                 mov     eax, 50h
.text:D018                 test    byte ptr ds:7FFE0308h, 1
.text:D020                 jnz     short loc_18009D025
.text:D022                 syscall                 ; Low latency system call
.text:D024                 retn
```

Figure 4. Code segment of ntdll.dll with VirtualMemory methods

The driver starts to inspect the code segments from the address of the `ZwAllocateVirtualMemory()` and looks up for instructions representing the constant move to `eax` (e.g. `mov eax, ??h`). It identifies the `VirtualMemory` methods; see **Table 7** for constants.

Constant	Method
0x18	`ZwAllocateVirtualMemory`
0x23	`ZwQueryVirtualMemory`
0x3A	`NtWriteVirtualMemory`
0x50	`ZwProtectVirtualMemory`

Table 7. Constants of Virtual Memory methods for Windows 10 (64 bit)

When an appropriate constants is found, the final address of a lookup method is calculated as follow:

`method_address = constant_address - alignment_constant`;
where `alignment_constant` helps to point to the start of the looked-up method.

The steps to find methods can be summarized as follow:

1. Get the address of `ZwAllocateVirtualMemory()`, which is not suspected in terms of detection.

2. Each sought method contains a specific constant on a specific address (`constant_address`).
3. If the `constant_address` is found, another specific offset (`alignment_constant`) is subtracted;
   the `alignment_constant` is specific for each Windows version.

The exact implementation of the Virtual Memory Method Lookup method is shown in **Figure 5**.

```
bool LookupVirtualMemoryMethods()
{
  bool status;

  wpZwReadVirtualMemory = GetVirtualMemoryMethod(ZwReadVM_offset);
  status = FALSE;
  if ( wpZwReadVirtualMemory )
  {
    wpZwWriteVirtualMemory = GetVirtualMemoryMethod(ZwWriteVM_offset);
    if ( wpZwWriteVirtualMemory )
    {
      ZwQueryVirtualMemory = GetVirtualMemoryMethod(ZwQueryVM_offset);
      if ( ZwQueryVirtualMemory )
      {
        wpZwProtectVirtualMemory = GetVirtualMemoryMethod(ZwProtectVM_offset);
        if ( wpZwProtectVirtualMemory )
          return TRUE;
      }
    }
  }
  return status;
}

PVOID GetVirtualMemoryMethod(int method_offset)
{
  int i, j;

  if ( method_offset != -1 && WinVersionOffset )
  {
    for ( i = 0; i < 100; ++i )
    {
      if ( *(&ZwAllocateVirtualMemory + i) == 0xB8 && *(&ZwAllocateVirtualMemory + i + 1) == WinVersionOffset )
      {
        for ( j = i; j < i + 100; ++j )
        {
          if ( *(&ZwAllocateVirtualMemory + j) == 0xB8 && *(&ZwAllocateVirtualMemory + j + 1) == WinVersionOffset + 1
          {
            if ( *(&ZwAllocateVirtualMemory + ((j - i) * (method_offset - WinVersionOffset)) + i) == 0xB8
              && *(&ZwAllocateVirtualMemory + ((j - i) * (method_offset - WinVersionOffset)) + i + 1) == method_offse
            {
              return &ZwAllocateVirtualMemory + ((j - i) * (method_offset - WinVersionOffset));
            }
            return NULL;
          }
        }
        return NULL;
      }
    }
  }
  return NULL;
}
```

Figure 5. Implementation of the lookup routine searching for the kernel VirtualMemory methods

The success of this obfuscation depends on the Window version identification. We found one Windows 7 version which returns different methods than the malware wants; namely, `ZwCompressKey()` , `ZwCommitEnlistment()` , `ZwCreateNamedPipeFile()` , `ZwAlpcDeleteSectionView()` .

The `alignment_constant` is derived from the current Windows version during the driver initialization; see the Driver Entry routine.

NtTestAlert and LdrLoadDll Lookup

A different approach is used for getting `NtTestAlert()` and `LdrLoadDll()` routines. The driver attaches to the `winlogon.exe` process and gets the pointer to the kernel structure `PEB_LDR_DATA` containing PE header and imports of all processes in the system. If the import table includes a required method, then the driver extracts the base address, which is the entry point to the sought routine.

2.7.2 **Process Injection**

The aim of the process injection is to load a defined DLL library into a new thread via kernel function `LdrLoadDll()`. The process injection is slightly different for x86 and x64 OS versions.

The x64 OS version abuses the original `NtTestAlert()` routine, which checks the thread's APC queue. The APC (Asynchronous Procedure Call) is a technique to queue a job to be done in the context of a specific thread. It is called periodically. The driver rewrites the instructions of the `NtTestAlert()` which jumps to the entry point of the malicious code.

Modification of NtTestAlert Code

The first step to the process injection is to find free memory for a code cave. The driver finds the free memory near the `NtTestAlert()` routine address. The code cave includes a shellcode as **Figure 6.** demonstrates.

```
mov   rax, NtTestAlert
push  rax
push  rcx
push  rdx
push   r8
push   r9
sub   rsp, 28h
mov   rcx, code_cave
mov   rdx, dll_loading_shellcode
jmp   rdx
```

Figure 6. Malicious payload overwriting the original NtTestAlert() routine

The shellcode prepares a parameter (`code_cave` address) for the malicious code and then jumps into it. The original `NtTestAlert()` address is moved into `rax` because the malicious code ends by the return instruction, and therefore the original `NtTestAlert()` is invoked. Finally, `rdx` contains the jump address, where the driver injected the malicious code. The next item of the code cave is a path to the DLL file, which shall be loaded into the injected process. Other items of the code cave are the original address and original code instructions of the `NtTestAlert()`.

The driver writes the malicious code into the address defined in `dll_loading_shellcode`. The original instructions of `NtTestAlert()` are rewritten with the instruction which just jumps to the shellcode. It causes that when the `NtTestAlert()` is called, the shellcode is activated and jumps into the malicious code.

Malicious Code x64

The malicious code for x64 is simple. Firstly, it recovers the original instruction of the rewritten `NtTestAlert()` code. Secondly, the code invokes the found `LdrLoadDll()` method and loads appropriate DLL into the address space of the injected process. Finally, the code executes the return instruction and jumps back to the original `NtTestAlert()` function.

The x86 OS version abuses the entry point of the injected process directly. The procedure is very similar to the x64 injection, and the x86 malicious code is also identical to the x64 version. However, the x86 malicious code needs to find a 32bit variant of the `LdrLoadDll()` method. It uses the similar technique described above (NtTestAlert and LdrLoadDll Lookup).

2.8 Service Hiding

Windows uses the Services Control Manager (SCM) to manage the system services. The executable of SCM is `services.exe`. This program runs at the system startup and performs several functions, such as running, ending, and interacting with system services. The SCM process also keeps all run services in an undocumented service record (`SERVICE_RECORD`) structure.

The driver must patch the service record to hide a required service. Firstly, the driver must find the process `services.exe` and attach it to this one via `KeStackAttachProcess()`. The malware author defined a byte sequence which the driver looks for in the process memory to find the service record. The `services.exe` keeps all run services as a linked list of `SERVICE_RECORD` [5]. The malware driver iterates this list and unlinks required services defined by *listServices* whitelist; see **Table 4**.

The used byte sequence for Windows 2000, XP, Vista, and Windows 7 is as follows: `{45 3B E5 74 40 48 8D 0D}`. There is another byte sequence `{48 83 3D ?? ?? ?? ?? ?? 48 8D 0D}` that is never used because it is bound to the Windows version that the malware driver has never identified; maybe in development.

The hidden services cannot be detected using PowerShell command `Get-Service`, Windows Task Manager, Process Explorer, etc. However, started services are logged via Windows Event Log. Therefore, we can enumerate all regular services and all logged services. Then, we can create differences to find hidden services.

2.9 Driver Hiding

The driver is able to hide itself or another malicious driver based on the IOCTL from the user-mode. The Driver Entry is initiated by a parameter representing a driver object (`DRIVER_OBJECT`) of the loaded driver image. The driver object contains an officially undocumented item called a driver section. The `LDR_DATA_TABLE_ENTRY` kernel structure stores information about the loaded driver, such as base/entry point address, image name, image size, etc. The driver section points to `LDR_DATA_TABLE_ENTRY` as a double-linked list representing all loaded drivers in the system.

When a user-mode application lists all loaded drivers, the kernel enumerates the double-linked list of the `LDR_DATA_TABLE_ENTRY` structure. The malware driver iterates the whole list and unlinks items (drivers) that should be hidden. Therefore, the kernel loses pointers to the hidden drivers and cannot enumerate all loaded drivers [6].

2.10 Driver Unload

The *Driver Unload* function contains suspicious code, but it seems to be never used in this version. The rest of the unload functionality executes standard procedure to unload the driver from the system.

3. Loading Driver During Boot

The DirtyMoe service loads the malicious driver. A driver image is not permanently stored on a disk since the service always extracts, loads, and deletes the driver images on the service startup. The secondary service aim is to eliminate evidence about driver loading and eventually complicate a forensic analysis. The service aspires to camouflage registry and disk activity. The DirtyMoe service is registered as follows:

```
Service name: Ms<volume_id>App ; e.g., MsE3947328App
Registry key: HKLM\SYSTEM\CurrentControlSet\services\<service_name>
ImagePath: %SystemRoot%\system32\svchost.exe -k netsvcs
ServiceDll: C:\Windows\System32\<service_name>.dll, ServiceMain
ServiceType: SERVICE_WIN32_SHARE_PROCESS
ServiceStart: SERVICE_AUTO_START
```

3.1 Registry Operation

On startup, the service creates a registry record, describing the malicious driver to load; see following example:

```
Registry key: HKLM\SYSTEM\CurrentControlSet\services\dump_E3947328
ImagePath: \??\C:\Windows\System32\drivers\dump_LSI_FC.sys
DisplayName: dump_E3947328
```

At first glance, it is evident that `ImagePath` does not reflect `DisplayName`, which is the Windows common naming convention. Moreover, `ImagePath` prefixed with `dump_` string is used for virtual drivers (loaded only in memory) managing the memory dump during the Windows crash. The malware tries to use the virtual driver name convention not to be so conspicuous. The principle of the Dump Memory using the virtual drivers is described in [7, 8].

`ImagePath` values are different from each windows reboot, but it always abuses the name of the system native driver; see a few instances collected during windows boot: `dump_ACPI.sys`, `dump_RASPPPOE.sys`, `dump_LSI_FC.sys`, `dump_USBPRINT.sys`, `dump_VOLMGR.sys`, `dump_INTELPPM.sys`, `dump_PARTMGR.sys`

3.2 Driver Loading

When the registry entry is ready, the DirtyMoe service dumps the driver into the file defined by `ImagePath`. Then, the service loads the driver via `ZwLoadDriver()`.

3.3 Evidence Cleanup

When the driver is loaded either successfully or unsuccessfully, the DirtyMoe service starts to mask various malicious components to protect the whole malware hierarchy.

The DirtyMoe service removes the registry key representing the loaded driver; see Registry Operation. Further, the loaded driver hides the malware services, as the Service Hiding section describes. Registry entries related to the driver are removed via the API call. Therefore, a forensics track can be found in the SYSTEM registry HIVE, located in `%SystemRoot%\system32\config\SYSTEM`. The API call just removes a relevant HIVE pointer, but unreferenced data is still present in the HIVE stored on the disk. Hence, we can read removed registry entries via RegistryExplorer.

The loaded driver also removes the dumped (`dump_` prefix) driver file. We were not able to restore this file via tools enabling recovery of deleted files, but it was extracted directly from the service DLL file.

Capturing driver image and register keys

The malware service is responsible for the driver loading and cleans up of loading evidence. We put a breakpoint into the `nt!IopLoadDriver()` kernel method, which is reached if a process wants to load a driver into the system. We waited for the wanted driver, and then we listed all the system processes. The corresponding service (`svchost.exe`) has a call stack that contains the kernel call for driver loading, but the corresponding service has been killed by EIP registry modifying. The process (service) was killed, and the whole Windows ended in BSoD. Windows made a crash dump, so the file system caches have been flushed, and the malicious service did not finish the cleanup in time. Therefore, we were able to mount a volume and read all wanted data.

3.4 Forensic Traces

Although the DirtyMoe service takes great pains to cover up the malicious activities, there are a few aspects that help identify the malware.

The DirtyMoe service and loaded driver itself are hidden; however, the Windows Event Log system records information about started services. Therefore, we can get additional information such as *ProcessID* and *ThreadID* of all services, including the hidden services.

WinDbg connected to the Windows kernel can display all loaded modules using the `lm` command. The module list can uncover non-virtual drivers with prefix `dump_` and identify the malicious drivers.

Offline connected volume can provide the DLL library of the services and other supporting files, which are unfortunately encrypted and obfuscated with VMProtect. Finally, the offline SYSTEM registry stores records of the DirtyMoe service.

4. Certificates

Windows Vista and later versions of Windows require that loaded drivers must be code-signed. The digital code-signature should verify the identity and integrity of the driver vendor [9]. However, Windows does not check the current status of all certificates signing a Windows driver. So, if one of the certificates in the path is expired or revoked, the driver is still loaded into the system. We will not discuss why Windows loads drivers with invalid certificates since this topic is really wide. The backward compatibility but also a potential impact on the kernel implementation play a role.

DirtyMoe drivers are signed with three certificates as follow:

Beijing Kate Zhanhong Technology Co.,Ltd.
Valid From: 28-Nov-2013 (2:00:00)
Valid To: 29-Nov-2014 (1:59:59)
SN: 3C5883BD1DBCD582AD41C8778E4F56D9
Thumbprint: 02A8DC8B4AEAD80E77B333D61E35B40FBBB010A0
Revocation Status: Revoked on 22-May-2014 (9:28:59)

Beijing Founder Apabi Technology Limited
Valid From: 22-May-2018 (2:00:00)
Valid To: 29-May-2019 (14:00:00)
SN: 06B7AA2C37C0876CCB0378D895D71041
Thumbprint: 8564928AA4FBC4BBECF65B402503B2BE3DC60D4D
Revocation Status: Revoked on 22-May-2018 (2:00:01)

Shanghai Yulian Software Technology Co., Ltd. (上海域联软件技术有限公司)
Valid From: 23-Mar-2011 (2:00:00)
Valid To: 23-Mar-2012 (1:59:59)
SN: 5F78149EB4F75EB17404A8143AAEAED7
Thumbprint: 31E5380E1E0E1DD841F0C1741B38556B252E6231
Revocation Status: Revoked on 18-Apr-2011 (10:42:04)

The certificates have been revoked by their certification authorities, and they are registered as stolen, leaked, misuse, etc. [10]. Although all certificates have been revoked in the past, Windows loads these drivers successfully because the root certificate authorities are marked as trusted.

5. Summarization and Discussion

We summarize the main functionality of the DirtyMoe driver. We discuss the quality of the driver implementation, anti-forensic mechanisms, and stolen certificates for successful driver loading.

5.1 Main Functionality

Authorization

The driver is controlled via IOCTL codes which are sent by malware processes in the user-mode. However, the driver implements the authorization instrument, which verifies that the IOCTLs are sent by authenticated processes. Therefore, not all processes can communicate with the driver.

Affecting the Filesystem

If a rootkit is in the kernel, it can do “anything”. The DirtyMoe driver registers itself in the filter manager and begins to influence the results of filesystem I/O operations; in fact, it begins to filter the content of the filesystem. Furthermore, the driver replaces the `NtfsCreatCallback()` callback function of the NTFS driver, so the driver can determine who should gain access and what should not get to the filesystem.

Thread Monitoring and Code injection

The DirtyMoe driver enrolls a malicious routine which is invoked if the system creates a new thread. The malicious routine abuses the APC kernel mechanism to execute the malicious code. It loads arbitrary DLL into the new thread.

Registry Hiding

This technique abuses the kernel hook method that indexes registry keys in HIVE. The code execution of the hook method is redirected to the malicious routine so that the driver can control the indexing of registry keys. Actually, the driver can select which keys will be indexed or not.

Service and Driver Hiding

Patching of specific kernel structures causes that certain API functions do not enumerate all system services or loaded drivers. Windows services and drivers are stored as a double-linked list in the kernel. The driver corrupts the kernel structures so that malicious services and drivers are unlinked from these structures. Consequently, if the kernel iterates these structures for the purpose of enumeration, the malicious items are skipped.

5.2 Anti-Forensic Technique

As we mentioned above, the driver is able to hide itself. But before driver loading, the DirtyMoe service must register the driver in the registry and dump the driver into the file. When the driver is loaded, the DirtyMoe service deletes all registry entries related to the driver loading. The driver deletes its own file from the file system through the kernel-mode. Therefore, the driver is loaded in the memory, but its file is gone.

The DirtyMoe service removes the registry entries via standard API calls. We can restore this data from the physical storage since the API calls only remove the pointer from HIVE. The dumped driver file is never physically stored on the disk drive because its size is too small and is present only in cache memory. Accordingly, the file is removed from the cache before cache flushing to the disk, so we cannot restore the file from the physical disk.

5.3 Discussion

The whole driver serves as an all-in-one super rootkit package. Any malware can register itself in the driver if knowing the authorization code. After successful registration, the malware can use a wide range of driver functionality. Hypothetically, the authorization code is hardcoded, and the driver's name can be derived so we can communicate with the driver and stop it.

The system loads the driver via the DirtyMoe service within a few seconds. Moreover, the driver file is never present in the file system physically, only in the cache. The driver is loaded via the API call, and the DirtyMoe service keeps a handler of the driver file, so the file manipulation with the driver file is limited. However, the driver removes its own file using kernel-call. Therefore, the driver file is removed from the file system cache, and the driver handler is still relevant, with the difference that the driver file does not exist, including its forensic traces.

The DirtyMoe malware is written using Delphi in most cases. Naturally, the driver is coded in native C. The code style of the driver and the rest of the malware is very different. We analyzed that most of the driver functionalities are downloaded from internet forums as public samples. Each implementation part of the driver is also written in a different style. The malware authors have merged individual rootkit functionality into one kit. They also merged known bugs, so the driver shows a few significant symptoms of driver presence in the system. The authors needed to adapt the functionality of the public samples to their purpose, but that has been done in a very dilettante way. It seems that the malware authors are familiar only with Delphi.

Finally, the code-signature certificates that are used have been revoked in the middle of their validity period. However, the certificates are still widely used for code signing, so the private keys of the certificates have probably been stolen or leaked. In addition, the stolen certificates have been signed by the certification authority which Microsoft trusts, so the certificates signed in this way can be successfully loaded into the system despite their revocation. Moreover, the trend in the use of certificates is growing, and predictions show that it will continue to grow in the future. We will analyze the problems of the code-signature certificates in the future post.

6. Conclusion

DirtyMoe driver is an advanced piece of rootkit that DirtyMoe uses to effectively hide malicious activity on host systems. This research was undertaken to inspect the rootkit functionally of the DirtyMoe driver and evaluate the impact on infected systems. This study set out to investigate and present the analysis of the DirtyMoe driver, namely its functionality, the ability to conceal, deployment, and code-signature.

The research has shown that the driver provides key functionalities to hide malicious processes, services, and registry keys. Another dangerous action of the driver is the injection of malicious code into newly created processes. Moreover, the driver also implements the minifilter, which monitors and affects I/O operations on the file system. Therefore, the content of the file system is filtered, and appropriate files/directories can be hidden for users. An implication of this finding is that malware itself and its artifacts are hidden even for AVs. More importantly, the driver implements another anti-forensic technique which removes the driver's evidence from disk and registry immediately after driver loading. However, a few traces can be found on the victim's machines.

This study has provided the first comprehensive review of the driver that protects and serves each malware service and process of the DirtyMoe malware. The scope of this study was limited in terms of driver functionality. However, further experimental investigations are needed to hunt out and investigate other samples that have been signed by the revoked certificates. Because of this, the malware author can be traced and identified using thus abused certificates.

IoCs

Samples (SHA-256)

```
550F8D092AFCD1D08AC63D9BEE9E7400E5C174B9C64D551A2AD19AD19C0126B1
AABA7DB353EB9400E3471EAAA1CF0105F6D1FAB0CE63F1A2665C8BA0E8963A05
B3B5FFF57040C801A4392DA2AF83F4BF6200C575AA4A64AB9A135B58AA516080
CB95EF8809A89056968B669E038BA84F708DF26ADD18CE4F5F31A5C9338188F9
EB29EDD6211836E6D1877A1658E648BEB749091CE7D459DBD82DC57C84BC52B1
```

References

[1] Kernel-Mode Driver Architecture
[2] Driver to Hide Processes and Files
[3] A piece of code to hide registry entries
[4] Hidden
[5] Opening Hacker's Door
[6] Hiding loaded driver with DKOM
[7] Crashdmp-ster Diving the Windows 8 Crash Dump Stack
[8] Ghost drivers named dump_*.sys
[9] Driver Signing
[10] Australian web hosts hit with a Manic Menagerie of malware

Appendix A

Registry entries used in the Start Routine

```
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDeviceAddress
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDeviceNumber
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDeviceId
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDeviceName
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDeviceNameB
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDeviceNameOnly
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverMaker1
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverMaker1_2
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverMaker2
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverMaker2_2
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDevicePathA
```

```
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDevicePathB
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverPath1
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverPath1_2
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverPath2
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverPath2_2
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDeviceDataA
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDeviceDataB
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverDataA
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverDataA_2
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverDataB
\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\WinApi\\WinDriverDataB_2
```

Appendix B

Example of registry entries configuring the driver

Key: `ControlSet001\Control\WinApi`
Value: `WinDeviceAddress`
Data: `Ms312B9050App` ;

Value: `WinDeviceNumber`
Data:
```
\WINDOWS\AppPatch\Ke601169.xsl;
\WINDOWS\AppPatch\Ke237043.xsl;
\WINDOWS\AppPatch\Ke311799.xsl;
\WINDOWS\AppPatch\Ke119163.xsl;
\WINDOWS\AppPatch\Ke531580.xsl;
\WINDOWS\AppPatch\Ke856583.xsl;
\WINDOWS\AppPatch\Ke999860.xsl;
\WINDOWS\AppPatch\Ke410472.xsl;
\WINDOWS\AppPatch\Ke673389.xsl;
\WINDOWS\AppPatch\Ke687417.xsl;
\WINDOWS\AppPatch\Ke689468.xsl;
\WINDOWS\AppPatch\Ac312B9050.sdb;
\WINDOWS\System32\Ms312B9050App.dll;
```

Value: `WinDeviceName`
Data:
```
C:\WINDOWS\AppPatch\Ac312B9050.sdb;
C:\WINDOWS\System32\Ms312B9050App.dll;
```

Value: `WinDeviceId`
Data: `dump_FDC.sys` ;

Tagged asDirtyMoe, Rootkit, series