Subverting Bootkits using the Crash Dump Driver Stack

Aaron LeMasters

3/15/2012

Contents

Technical Overview

An operating system provides facilities that allow applications to carry out tasks without having to worry about details of underlying hardware. For example, when an application saves a file to disk, it is able to carry out that task without knowledge of the underlying disk geometry or transport protocol, because the operating system implements and controls the mechanisms for converting the high-level application request into a low-level hardware command. Applications rely heavily on the operating system for accessing low-level hardware resources, and computer forensic tools, for the most part, are no exception. Even popular disk forensic tools such as The Sleuth Kit [1], which provides raw disk access for manually extracting data from popular file system formats, still rely on various drivers provided by the operating system to access the data on disk. The operating system in turn relies on the hardware to operate correctly and conform to certain standards and rules. This trust relationship is a fundamental design principle of the Windows Driver Framework and is exhibited in all driver and device stacks in the operating system. Figure 1 illustrates the chain of trust for the mass storage device stack or "Normal I/O Path".

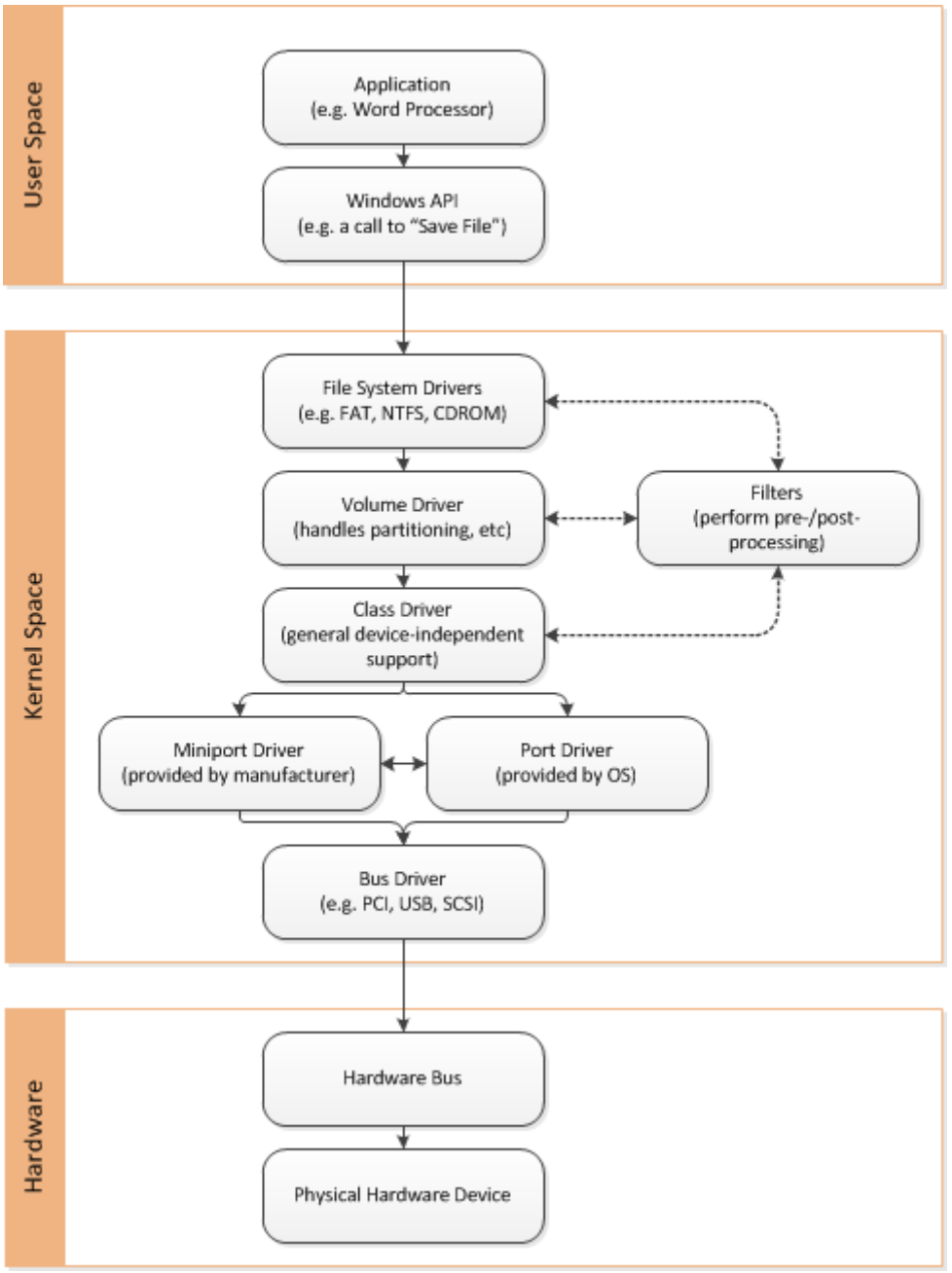


Figure 1: Mass storage device stack or "Normal I/O Path"

In this trust chain, an I/O request initiates in the user space layer and travels a long winding path through the normal I/O path, until it eventually reaches the physical disk in the hardware layer. Each driver in the chain performs its intended function and passes the request down to the next driver – all coordinated by the operating system I/O subsystem. The Host Bus Adapter (HBA), firmware on the physical disk, and miniport driver work together to complete the request and pass it back up the driver chain to the user.

Rootkits operate in the kernel space layer and install themselves into one or more of the components in this normal I/O path in order to manipulate disk contents (e.g., to hide files). Once the normal I/O path is contaminated in this manner, it becomes useless to any driver or application above it, since it is returning altered data. This includes forensic tools that might be relying on the normal I/O path to search for evidence on a hard drive. Contaminated evidence can cause an active investigation to be misled and is typically not admissible in court.

Numerous solutions have been proposed to this contamination problem over the past 20 years. Anti-virus and HIPS vendors attempt to uninstall the rootkits dynamically or put measures in place to prevent them from installing at all. This approach becomes a classic cat-and-mouse game, as rootkit authors simply install at some other junction in the normal I/O path or subvert the prevention measure itself. Additionally, it is often the case that "whoever is there first" wins this battle, since all software that operates at the kernel space layer has equal privileges. Microsoft itself has put measures in place to protect critical components in trust chains such as the normal I/O path, but rootkit authors outsmart these measures and manage to infect systems.

Since the battle for disk control is raging in the normal I/O path, this paper introduces a novel technique to control the hard disk through a completely different I/O path: what is referred to as the *crash dump I/O path*. The crash dump I/O path, illustrated in Figure 2, is completely separate from the normal I/O path and is largely hidden and undocumented by Microsoft. It is used by the operating system when the system becomes unstable (e.g., Blue Screen of Death) to save critical debugging information to disk (inside what is called a *crash dump file*). This process is known as the *crash dump process*.

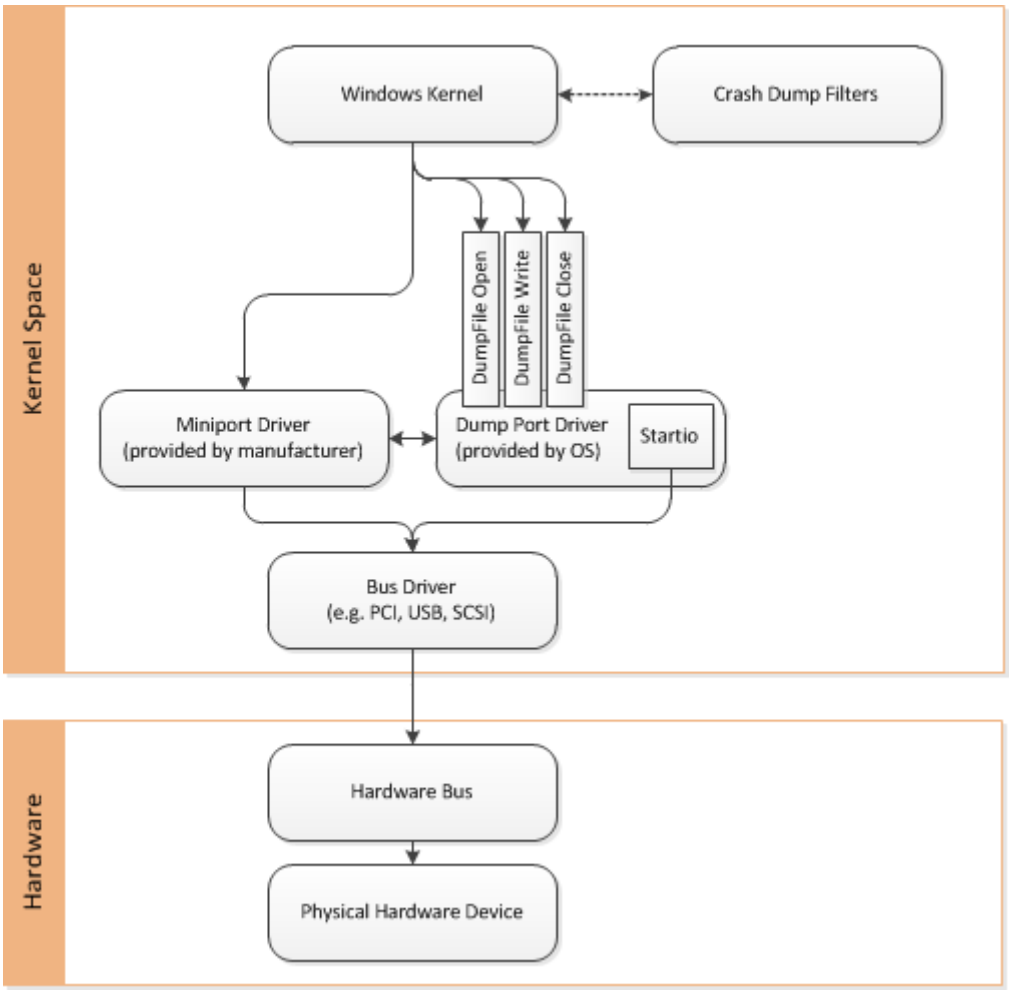


Figure 2: The Crash Dump I/O Path

Aside from missing most of the components in the normal I/O path, the major difference in the crash dump I/O path is that it uses a special copy of the port and miniport drivers. When a system crash occurs, since the operating system cannot guarantee the critical fault did not occur in one of the drivers in the normal I/O path, it uses a separate copy of the port and miniport driver for the boot device to write to the disk.

The port driver is an abstraction interface provided by Microsoft that allows higher-level drivers to communicate with the disk without bothering with protocol-specific details. It communicates with a manufacturer-provided miniport driver beside it, which implements a hardware-specific interface to the Host Bus Adapter of the physical disk.

When the crash dump process is initiated, the normal I/O path is disabled. The operating system activates the special dump port and miniport drivers and writes the crash dump file to disk using special functions provided by the dump port driver. These functions restrict the operating system to only *open*, *write* and *close* a dump file.

Using a special technique, illustrated in Figure 3, it is possible to gain access to the crash dump I/O path and read or write to disk without bothering with the normal I/O path. The approach is similar to the crash dump process briefly described above. However, rather than using the provided dump port functions, which only allow control of the crash dump file in limited ways, the technique involves using the dump port driver's hidden `StartIo` or `DispatchCrb` functions to send read or write requests directly to the hardware, without any restrictions. Since these crash dump drivers are part of a mechanism supported by Microsoft, this technique is for the most part portable to all versions of Windows. It is also applicable to multiple transport protocols such as IDE and RAID.

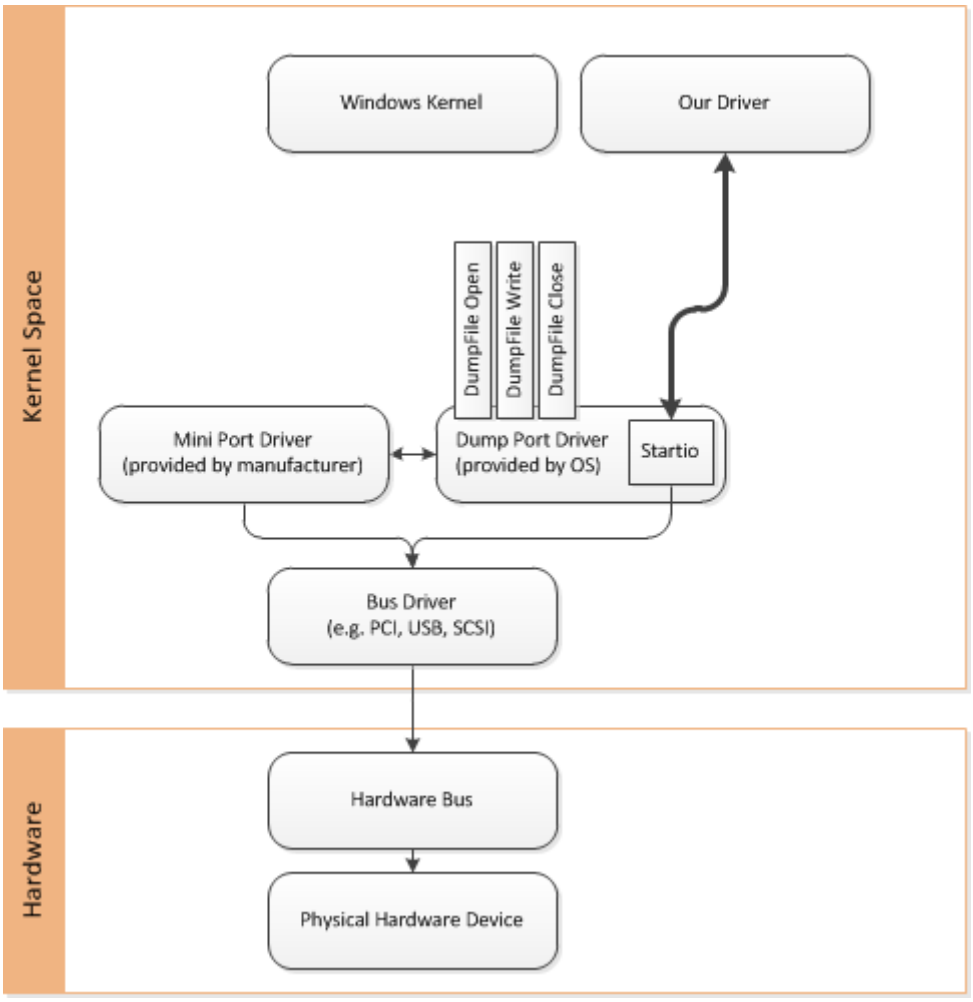


Figure 3: Modified use of the Crash Dump I/O Path

Since these dump drivers are not initialized like normal drivers and are not part of the normal I/O path, it is extremely difficult for rootkits to hook them in classic ways (and doing so might make the system unusable, which rootkit authors would not want). At the time of this writing, there are no known rootkits in the wild that infect the crash dump I/O path. The author speculates this is partially because of the technical challenges and because of the possibility of destabilizing the system if the crash dump I/O path is corrupted. In other words, infecting the crash dump I/O path is a risky proposition for rootkit

authors, so they are likely to avoid contaminating it. This fact makes it a desirable method for collecting forensic evidence.

This paper will illustrate how this technique can be used to subvert one of the lowest-level I/O hooking rootkits in the wild: Master Boot Record (MBR) rootkits (or simply *bootkits*). Bootkits have existed for decades but are more recently gaining widespread attention with the growing deployment of nasty bootkits such as TDL4 and Popureb. The most advanced versions of these rootkits hook the normal I/O path at the lowest possible level: the port and miniport drivers. It infects these drivers, replaces the MBR with an infected one, and fools forensic tools into thinking the system is not infected by returning a clean copy of the MBR. Since this technique leverages the crash dump I/O path, it is possible to reveal the true, infected MBR.

The following section covers in greater detail the crash dump process itself, from Windows 2000 up to Windows 7, and how the crash dump I/O path can be used to control the disk.

The Crash Dump Mechanism In-Depth

On system boot-up, the kernel initializes a number of critical system components, one of which the crash dump mechanism, which is used to save debugging information to disk when the system encounters an unrecoverable condition (a.k.a, crash, blue screen, or bugcheck). The crash dump mechanism consists of the crash dump driver stack and various global kernel data structures that maintain state information.

Overview of the Crash Dump Driver Stack

The crash dump driver stack is used to write a crash dump file to disk when the system crashes. The Windows kernel decides what disk drivers are required given the hardware attributes of the boot device (such as transport protocol) and loads/maps them into memory. The crash dump stack consists of the following drivers:

1. A dump port driver – an abstraction interface provided by the operating system; implements a specific transport protocol (such as SCSI or IDE); the operating system provides three such port drivers: `scsiport.sys` (SCSI), `ataport.sys` (ATA/IDE), and `storport.sys` (optimized for RAID)
2. A dump miniport driver – a hardware-specific driver provided by the vendor which translates generic operating system commands into vendor-specific commands
3. One or more crash dump filter drivers – these are typically third-party drivers that need to perform some pre or post processing of the crash dump file data, such as whole-disk encryption drivers that wish to encrypt the crash dump file itself.

There will always be only one dump port driver and one dump miniport driver. Table 1 below describes some of the drivers that can be found in the crash dump stack.

Driver Name On Disk	Driver Base Name in Memory	Purpose
diskdump.sys	dump_diskdump	SCSI/Storport dump port driver with required exports from

		scsiport.sys and storport.sys. This driver is unloaded.
dumpata.sys	dump_dumpata	IDE/ATA dump port driver with required ataport.sys exports. This driver is unloaded.
scsiport.sys	dump_scsiport	The final SCSI/Storport dump port driver.
ataport.sys	dump_ataport	The final IDE/ATA dump port driver.
atapi.sys	dump_atapi	An older, generic ATAPI miniport driver provided by the OS for IDE/ATA drives
vmscsi.sys	dump_vmscsi	The miniport driver provided by VMWare for SCSI drives.
LSI_SAS.sys	dump_LSI_SAS	The miniport driver provided by LSI Corporation for serial-attached storage drives.
dumpfve.sys	dump_dumpfve	Windows full volume encryption crash dump filter driver

Table 1: Common crash dump stack drivers

The crash dump stack is initialized and configured when a page file is created on any fixed disk, which occurs:

- At system boot up during kernel phase 1 initialization
- When `NtCreatePagingFile()` is called

The crash dump stack is used when:

- A bug check occurs
- The system is about to hibernate

The crash dump stack must be initialized before a crash dump file can be written to a paging file. Storage for a crash dump file is allocated and initialized for the first paging file on a fixed disk drive. The first time this happens is during phase 1 initialization of system boot up, just after the boot device has been initialized. The kernel reads the name of the boot device's page file(s) from the registry key `HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\ExistingPageFiles`. Normally the page file stored at this location is `C:\pagefile.sys`, but it can be any drive and up to 16 page files. The crash dump storage space will be allocated for the first paging file, and the crash dump stack is initialized.

The crash dump stack is initialized and used differently based on operating system version. On operating systems prior to Windows Vista, the kernel performs all of the crash dump stack initialization and dump file creation. For Vista and later operating systems, the majority of this code (along with new features) was moved to a new driver, `crashdmp.sys`. Furthermore, whereas the older kernel exported type information for many of the (partially documented) critical crash dump data structures, the

`crashdmp.sys` driver introduced a host of new data structures that are neither documented nor exported.

There are several principles governing the use of the crash dump stack in all versions of Windows:

1. All processors are disabled except the one the current thread is executing on
2. The active CPU becomes single-threaded (IRQL is raised to HIGH_LEVEL) and uninterruptible
3. I/O sent to the crash dump stack is sent one request at a time
4. If IDE controller, only the channel containing the device with the page file is enabled.

Crash Dump Stack Initialization Prior to Windows Vista

In older operating systems, the crash dump stack and all its critical components were contained in a global variable named `IopDumpControlBlock`. This variable is of an undocumented but exported type `DUMP_CONTROL_BLOCK` and contains information about basic system properties, crash dump settings from the registry, and meta-information about the crash dump file itself.

```
typedef struct _DUMP_CONTROL_BLOCK {
    UCHAR Type;
    CHAR Flags;
    USHORT Size;
    CCHAR NumberProcessors;
    CHAR Reserved;
    USHORT ProcessorArchitecture;
    PDUMP_STACK_CONTEXT DumpStack;
    PPHYSICAL_MEMORY_DESCRIPTOR MemoryDescriptor;
    ULONG MemoryDescriptorLength;
    PLARGE_INTEGER FileDescriptorArray;
    ULONG FileDescriptorSize;
    PULONG HeaderPage;
    PFN_NUMBER HeaderPfn;
    ULONG MajorVersion;
    ULONG MinorVersion;
    ULONG BuildNumber;
    CHAR VersionUser[32];
    ULONG HeaderSize;
    LARGE_INTEGER DumpFileSize;
    ULONG TriageDumpFlags;
    PUCHAR TriageDumpBuffer;
    ULONG TriageDumpBufferSize;
}DUMP_CONTROL_BLOCK, *PDUMP_CONTROL_BLOCK;
```

Figure 4: The DUMP_CONTROL_BLOCK structure

The most important field is `DumpStack`, which is an undocumented (but exported) structure of type `DUMP_STACK_CONTEXT` shown below. This structure is a container for information about the crash dump drivers and associated configuration.

```
typedef struct _DUMP_STACK_CONTEXT {
    DUMP_INITIALIZATION_CONTEXT Init;
    LARGE_INTEGER               PartitionOffset;
    PVOID                       DumpPointers;
    ULONG                       PointersLength;
    PWCHAR                      ModulePrefix;
    LIST_ENTRY                  DriverList;
    ANSI_STRING                 InitMsg;
```

```
    ANSI_STRING                 ProgMsg;
    ANSI_STRING                 DoneMsg;
    PVOID                       FileObject;
    enum _DEVICE_USAGE_NOTIFICATION_TYPE    UsageType;
} DUMP_STACK_CONTEXT, *PDUMP_STACK_CONTEXT;
```

Figure 5: The DUMP_STACK_CONTEXT structure

The `DumpPointers` field contains hardware-specific information about the disk drive (obtained through a call to the disk driver via `IOCTL_SCSI_GET_DUMP_POINTERS`) which is used during write I/O operations to the crash dump file. The `DriverList` field contains a linked list of data structures that describe the driver image of each driver in the crash dump stack (such as crash dump filter drivers). This field will be used at actual crash dump time to initialize each driver. The `Init` field of type `DUMP_INITIALIZATION_CONTEXT` (undocumented but exported), shown below, is only partially filled in during this phase of initialization. The highlighted fields below are filled in later by the crash dump port driver at the time a crash dump is initiated. These fields are pointers to functions exported by the dump port driver which provide the kernel the ability to write the crash dump data to the crash dump file.

```
typedef struct _DUMP_INITIALIZATION_CONTEXT {
    ULONG Length;
    ULONG Reserved;
    PVOID MemoryBlock;
    PVOID CommonBuffer[2];
    PHYSICAL_ADDRESS PhysicalAddress[2];
    PSTALL_ROUTINE StallRoutine;
    PDUMP_DRIVER_OPEN OpenRoutine;
    PDUMP_DRIVER_WRITE WriteRoutine;
    PDUMP_DRIVER_FINISH FinishRoutine;
    struct _ADAPTER_OBJECT *AdapterObject;
    PVOID MappedRegisterBase;
    PVOID PortConfiguration;
    BOOLEAN CrashDump;
    ULONG MaximumTransferSize;
    ULONG CommonBufferSize;
    PVOID TargetAddress;
    PDUMP_DRIVER_WRITE_PENDING WritePendingRoutine;
    ULONG PartitionStyle;
    union  {
              struct {
               ULONG Signature;
               ULONG CheckSum;
              } Mbr;
              struct {
               GUID DiskId;
              } Gpt;
            } DiskInfo;
} DUMP_INITIALIZATION_CONTEXT, *PDUMP_INITIALIZATION_CONTEXT;
```

Figure 6: The DUMP_INITIALIZATION_CONTEXT structure

The call chain outline below summarizes what kernel functions are responsible for initializing the crash dump stack in operating systems prior to Windows Vista.

- `KiInitializeKernel()` -> `IoInitSystem()` OR `NtCreatePagingFile()`:
    - `IoInitializeCrashDump()`

- `IopInitializeDCB()` – read dump settings from registry, allocate memory for global `IopDumpControlBlock`
- `IoGetDumpStack()->IopGetDumpStack()` – Fills `DumpStack` and `DumpInit` substructures of global `IopDumpControlBlock` with boot device type, geometry, and dump retrieval pointers; responsible for locating the port and miniport drivers and mapping them into memory with the "dump_" prefix using `nt!IopLoadDumpDriver()`
- `IopCompleteDumpInitialization()` – sets dump range limits and calculates `IopDumpControlBlock` checksum

Crash Dump Stack Initialization in Windows Vista and Later

On Windows Vista and later operating systems, most of the crash dump stack initialization code was moved from the kernel and integrated with additional features into a driver, `%SYSTEM32%\Drivers\crashdmp.sys`. The kernel's `IoInitializeCrashDump` simply loads this crash dump driver into memory and calls its entry point with two arguments: the name of the "arc" boot device and a pointer to a global crash dump callback table. The crash dump driver's entry point loads the callback table with pointers to callback functions for the kernel to use, as shown in Table 2 below.

Table Offset	Value
0x0	1
0x4	1
0x8	`CrashdmpInitialize`
0xC	`CrashdmpLoadDumpStack`
0x10	`CrashdmpInitDumpStack`
0x14	`CrashdmpFreeDumpStack`
0x18	`CrashdmpDisable`
0x1C	`CrashdmpNotify`
0x20	`CrashdmpWrite`
0x24	`CrashdmpUpdatePhysicalRange`

Table 2: The crashdmp call table

After the entry point of `crashdmp.sys` fills the callback table, `IoInitializeCrashDump` calls the third entry in the table, which is always a pointer to `CrashDmpInitialize`. This function takes two arguments: a handle to the paging file and a pointer to a kernel-global dump block variable (referred to as `DumpBlock`) of an undocumented and unexported type. `DumpBlock` is where crash dump information will be stored when the system crashes. Aside from a single field in the structure, `CrashDmpInitialize` does not modify this dump block variable. Instead, it initializes various internal dump control structures that take the place of the global kernel dump control block `IopDumpControlBlock` used in prior operating system versions (such as the sub-structures `DumpInit` and `DumpStack`).

The call chain outline below summarizes the functions involved in initializing the crash dump stack in Vista and later operating systems.

- `KiInitializeKernel() -> IoInitSystem()` OR `NtCreatePagingFile()`:
  - `IoInitializeCrashDump()`
    - `IopLoadCrashDumpDriver()` – loads `crashdmp.sys` into memory and calls its entry point, passing the arc name of the boot device and a pointer to crash dump call table
      - `Crashdmp!DriverEntry()` – fills crash dump call table
    - `Crashdmp!CrashDmpInitialize()` – reads registry settings, initializes an internal `DumpBlock` and populates the crash dump driver stack (port, miniport and filter drivers) as follows:
      - `Crashdmp!CrashdmpLoadDumpStack()` – queries the boot device and then calls the following functions:
        - `Crashdump!QueryPortDriver()` – sends various IOCTLs to the port driver to collect partition information, disk geometry, and SCSI dump pointers.
        - `Crashdmp!LoadPortDriver()` – responsible for locating the port and miniport drivers and mapping them into memory with the "dump_" prefix. See comments in the next paragraph for details.
        - `Crashdmp!LoadFilterDrivers()` – loads crash dump filter drivers stored in the registry at `HKLM\CurrentControlSet\Control\CrashControl\DumpFilters`, also prefixed with "dump_"
        - `Crashdmp!InitializeFilterDrivers()` – calls the entry point of all crash dump filter drivers

`Crashdmp!LoadPortDriver()` is responsible for an important step in the crash dump stack initialization process – loading the correct port and miniport drivers. It calls `Crashdmp!GetLegacyPortDriverName()` which uses the following strategy to locate the name of the port driver. It calls `GetPortDriverName()`, which calls `GetPortDriverObject()` to get a handle to the port driver object represented by calling `ZwOpenFile()/ObReferenceObjectByHandle()` on the named object `\\Device\ScsiPort<x>` where "x" is the port number of the SCSI boot device. The driver object contains a pointer to the driver path and image name. For non-SCSI devices, a driver object was already obtained, so `GetPortDriverObject()` is not called. It also loads the miniport driver associated with this boot device in the same manner.

Initialization of Drivers in the Crash Dump Driver Stack

In the preceding sections concerning dump stack initialization, `nt!IopGetDumpStack` (prior to Vista) and `crashdmp!CrashDmpInitialize` (Vista and later) were responsible for loading a series of drivers into the crash dump stack. These drivers include a port driver, a miniport driver, and one or more crash dump filter drivers. All of these drivers are re-named with a prefix of "dump_" when they are mapped into memory.

The port driver in the crash dump stack is actually a modified version of the original OS-provided port driver. For ATA transport protocols, the dump port driver is named on disk as `dumpata.sys`. For SCSI transports (as well Storport, Microsoft's optimized version of scsiport), the dump port driver is named on disk as `diskdump.sys`. The miniport driver in the crash dump stack is a copy of the manufacturer-provided miniport driver for the boot device. After locating the appropriate miniport and port drivers (e.g., if the boot device is using SCSI transport protocol, the dump port driver will be `diskdump.sys`), the kernel (or `crashdmp.sys`) maps a copy of these drivers into memory, so that they are permanently resident and available when a system crash occurs. To insure the copies of the port and miniport drivers used for the crash dump stack do not conflict with the normal I/O path, the copies are prefixed with the string "dump_" (e.g., "dump_scsiport", "dump_storport" or "dump_ataport"). This mapping is also done for the miniport driver (eg, "dump_vmscsi" for `vmscsi.sys`, a SCSI miniport driver provided by VMWare) and all of the crash dump filter drivers (e.g., "dump_dumpfve" for `dumpfve.sys`, a Microsoft-provided full volume encryption driver). One exception to this naming convention is on systems where the miniport driver is linked against the storport port driver, the operating system will create a "hybrid" dump port driver named "dump_diskdump.sys". This dump port driver contains both scsiport and storport exports, since the storport protocol is based on SCSI.

It is worth noting that the operating system does not initialize typical driver management structures such as `DRIVER_OBJECT` and `DEVICE_OBJECT` for the crash dump drivers. As a result, these often-hooked management structures are not available for rootkits to tamper with.

There are some important differences between the dump port driver in the Crash Dump I/O path and the port driver in the Normal I/O path that allow the dump port driver to operate without normal operating system components. The dump port driver keeps all of the exported functions that the miniport driver needs but differs from the standard port driver in the following crucial ways:

1. Most internal functions not necessary for dump file creation and writing are stripped
2. Three new exported functions are provided to the kernel for crash dump file generation – `DiskDumpOpen`, `DiskDumpWrite`, `DiskDumpFinish`
3. For SCSI drives, an internal function called `StartIo` is used to transmit a single SCSI request block to the dump miniport driver for completion; for IDE drives, an IDE request block is sent via the internal function `DispatchCrb`

Similarly, the dump miniport driver must be modified by the kernel to operate in the crash dump stack. When a vendor provides a miniport driver to operate its hardware, the driver is linked against the appropriate port driver. The operating system must "rebind" the dump miniport driver to the dump port driver. This is necessary because the original miniport driver is linked against the original port driver. Without this rebinding, the dump miniport driver would attempt to communicate with the original port driver.

With the changes noted above and by forcing the dump miniport driver to use the dump port driver, the operating system has essentially circumvented the normal I/O path. Furthermore, Microsoft requires that manufacturer-supplied miniport drivers operate in "dump mode" [2]:

*A storage miniport driver that manages an adapter for a boot device is subject to special restrictions during a system crash. While dumping the system's memory image to disk, the miniport driver must operate within a different environment. The usual communication between the miniport driver, the port driver, and disk class driver is interrupted. The kernel does disk I/O by direct calls to the disk dump port driver (*diskdump.sys *for SCSI adapters or* dumpata.sys *for ATA controllers), bypassing file systems, and the normal I/O stack. The disk dump driver, in turn, calls the boot device's miniport driver to handle all I/O operations, and the disk dump driver intercepts all of the miniport driver's calls to the port driver.*

In other words, the miniport driver must be able to cope with this special crash dump mode. This is important, because it implies that the crash dump I/O path, something the technique in this paper relies on, is guaranteed to be supported regardless of hard drive type or manufacturer. According to MSDN, the miniport driver is instructed to operate in dump mode when the OS calls its `DriverEntry` with null parameters instead of the normal `DriverEntry` parameters.

Crash Dump Stack Usage Prior to Windows Vista

If the system crashes, and it has been configured to write a crash dump, the kernel uses the already-initialized crash dump stack to write the crash dump file to the paging file on disk.

When the system encounters a crash, the kernel function `KeBugCheck2()` is called, which in turn calls `IoWriteCrashDump()`. This function verifies the checksum of `IopDumpControlBlock` (the global variable that contains information about the crash dump stack) and calls `IoInitializeDumpStack()`, which walks the list of crash dump drivers stored in the `DriverList` field of the `DUMP_STACK_CONTEXT` structure as mentioned in the crash dump driver initialization section. For each driver in the dump stack, its `DriverInit` (`DriverEntry`) routine is called. The first driver in the dump stack (always the dump port driver) gets the `DUMP_INITIALIZATION_CONTEXT` structure (shown below) stored in the `DUMP_STACK_CONTEXT` passed to its `DriverEntry` as the second argument. This driver populates the initialization structure below with functions that will be used by the kernel to do disk I/O, highlighted below.

```
typedef struct _DUMP_INITIALIZATION_CONTEXT {
    ULONG Length;
    ULONG Reserved;
    PVOID MemoryBlock;
    PVOID CommonBuffer[2];
    PHYSICAL_ADDRESS PhysicalAddress[2];
    PSTALL_ROUTINE StallRoutine;
    PDUMP_DRIVER_OPEN OpenRoutine;
    PDUMP_DRIVER_WRITE WriteRoutine;
    PDUMP_DRIVER_FINISH FinishRoutine;
    struct _ADAPTER_OBJECT *AdapterObject;
    PVOID MappedRegisterBase;
```

```
    PVOID PortConfiguration;
    BOOLEAN CrashDump;
    ULONG MaximumTransferSize;
    ULONG CommonBufferSize;
    PVOID TargetAddress;
    PDUMP_DRIVER_WRITE_PENDING WritePendingRoutine;
    ULONG PartitionStyle;
    union  {
              struct {
               ULONG Signature;
               ULONG CheckSum;
              } Mbr;
              struct {
               GUID DiskId;
              } Gpt;
            } DiskInfo;
} DUMP_INITIALIZATION_CONTEXT, *PDUMP_INITIALIZATION_CONTEXT;
```

Figure 7: The dump callback routines in the DUMP_INITIALIZATION_CONTEXT structure

The dump port driver populates the `OpenRoutine`, `WriteRoutine` and `FinishRoutine` fields of this structure with pointers to its own callback functions, which the kernel will call to write the crash dump file to the boot disk drive's paging file.

After the dump port driver is initialized, the second driver in the crash dump stack is the dump miniport driver. The dump port driver relies on the miniport driver to carry out the I/O requests. The miniport `DriverEntry()` routine registers information about its device's hardware with the Port driver by calling the export `ScsiPortInitialize()` for SCSI and `AtaPortInitialize()` for IDE. It passes a structure that contains pointers to callbacks for various operations required by the port driver: `HwInitialize`, `HwResetBus`, `HwStartIo`, `HwInterrupt`, `HwAdapterControl`, and `HwFindAdapter`.

Finally, `IoInitializeDumpStack()` calls the `DiskDumpOpen()` callback of the dump port driver to open the boot partition in preparation for writing to the page file.

Now that post-initialization of the crash dump I/O path is complete, `IoWriteCrashDump()` creates the dump file as follows:

- Displays the dump string "Beginning dump of physical memory", stored in the `DUMP_CONTROL_BLOCK` structure
- Calculates the dump storage space required based on configuration
- Fills a dump header with bug check codes and other debug information
- Invokes all `BugCheckDumpIoCallback` callbacks registered with the kernel via `KeRegisterBugCheckReasonCallback()` [5], passing the dump header
- Writes the appropriate dump file and data as configured in the system registry. If the system is configured to dump only summary crash dumps, `IoWriteCrashDump()` calls `IopWriteSummaryHeader()`/`IopWriteSummaryDump()`. If the system is configured for a triage dump, it calls `IopWriteTriageDump()`.
- Invokes all `BugCheckSecondaryDumpDataCallback` callbacks to allow drivers to append data to the completed crash dump file.

- Invokes all `BugCheckDumpIoCallback` callbacks, informing them crash dump is complete.

Crash Dump Stack Usage in Windows Vista and Later

The usage of the crash dump stack in Vista and later operating systems is similar to prior versions, except the majority of the crash dump stack management code is in the `crashdmp.sys` driver. Additionally, new, undocumented dump structures are used in this driver. As discussed earlier, the kernel uses a `CrashDmpCallTable` variable to call various internal functions within the `crashdmp.sys` driver.

When the system encounters a crash, the kernel function `KeBugCheck2()` calls `IoWriteCrashDump()`, which behaves differently than versions of earlier operating systems. `IoWriteCrashDump()` does not call any entry points for drivers in the crash dump stack. Instead, it just stores the dump information in the global crash dump block configured earlier, as follows:

- Calls the eighth entry in the `CrashDmpCallTable` table, `CrashDmpNotify()` which simply validates the integrity of the internal dump context structure and displays the string "collecting data for crash dump"
- Fills dump block with bug check codes and other debug information
- Appends a triage dump if the system is configured to do so
- Invokes all `BugCheckAddPagesCallback` callbacks and appends any dump pages returned by these callbacks to the dump data
- Calls the ninth entry in the `CrashDmpCallTable` table, `CrashDmpWrite()`, passing the configured dump block

At this point, the `crashdmp.sys` driver takes over writing the crash dump data to disk: `IoWriteCrashDump()` returns after this call and the kernel finishes bringing the system down after the bug check. The process of writing the crash dump data to disk in `CrashDmpWrite()` is summarized in the following call stack outline:

- `CrashdmpInitDumpStack()`
  - `StartFilterDrivers()` – this calls the `DumpStart` callback of each crash dump filter driver registered with the system.
  - `InitializeDumpDriver()` or `InitializeDumpPath()`
    - `InitializeDumpDriver()` – calls the entry point of each driver in the dump stack, passing a null pointer and a pointer to the `DUMP_INITIALIZATION_CONTEXT` structure to the dump port driver. After all dump drivers have been called, it calls the `DiskDumpOpen` callback provided by the dump port driver.
    - `InitializeDumpPath()` – first initializes the dump block structure before simply calling `InitalizeDumpDriver()`

- `DumpWrite()` – carries out the actual writing of crash dump data to disk based on the configured crash dump type:
  - `FillDumpHeader()` – loads the dump header with debugging information
  - `WriteFullDump()` or `WriteKernelDump()` or `WriteMiniDump()` – one of these functions is called to optionally append a full dump, kernel dump or mini dump to the dump header.
- `InvokeSecondaryDumpCallbacks()` - Invokes all `BugCheckSecondaryDumpDataCallback` callbacks to allow drivers to append data to the completed crash dump file.
- `InvokeDumpCallbacks()` - Invokes all `BugCheckDumpIoCallback` callbacks, informing them crash dump is complete.

The dump port and miniport drivers operate the same as in prior versions of Windows.

Leveraging the Crash Dump Stack

The crash dump mechanism represents a pristine path to disk, tucked away in the bowels of the operating system. The problem is that it is specifically limited to writing data to the disk (more accurately, to the paging file). As the prior sections have outlined, crash dump data is written to disk by calling the dump port driver's `DiskDumpWrite` export, which in turn calls the appropriate miniport function with a SCSI Request Block (SRB) or IDE Request Block (IRB). There is an open, write and finish function exported by the dump port driver for use by the crash dump process, but no corresponding read function. If there were a read function supplied, it would be a simple matter of calling that function to read disk. The following section explores how to leverage what has been presented about the internal crash dump structures to coerce the crash dump stack to allow both reading and writing to arbitrary disk locations. Figure 4 below summarizes the steps outlined in this section.

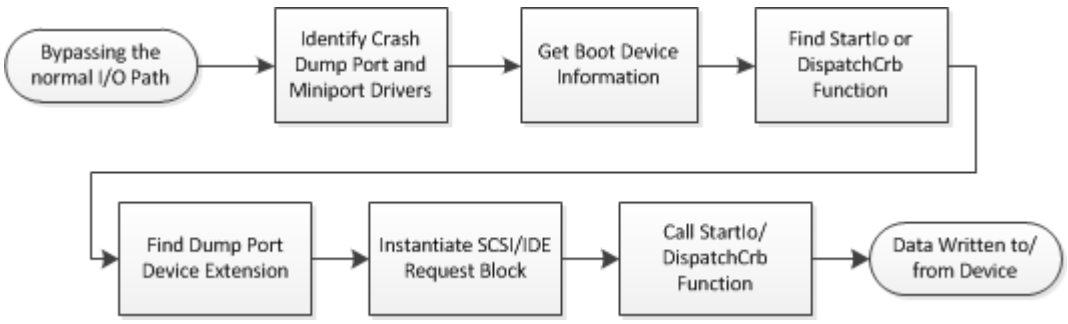


Figure 8: Using the crash dump I/O path

Identify Crash Dump Port and Miniport Drivers

Before the crash dump drivers can be used, they must be located in memory. Since all drivers in the crash dump stack are prefixed with the string "dump_", they can easily be singled out by searching the loaded module list for modules named with that prefix. Since it is known that the dump port driver must be one of "dump_scsiport" (SCSI), "dump_storport" (Storport), or "dump_ataport" (for IDE/ATA), it is trivial to identify the dump port driver. However, the dump miniport driver is provided by the hardware vendor and can be named anything (e.g., "dump_LSI_SAS", "dump_vmscsi", etc.). The name

can be obtained by opening a handle to the boot device's device object. Typically, a handle to the boot device can be obtained through the named device `\\Device\Harddisk0\DR0`, which represents the device object for the boot disk as setup by the class driver (i.e., `disk.sys`). This device object can be "walked" via kernel-provided API's to find the lowest driver object attached to the disk device stack. The name stored in this lowest driver object will be the name of the miniport driver. During crash dump initialization, the operating system obtains the miniport driver name using a global ARC string named `ArcBootDeviceName` which contains the Arc path information to the boot device.

In certain versions of Windows, there is a second way to identify these drivers through the IOCTL_SCSI_GET_DUMP_POINTERS command. If this command is successfully issued to the boot device (via the named device), a structure of documented type DUMP_POINTERS_EX is returned. The last field in this structure is named DriverList and contains a null-separated list of the drivers in the dump stack.

To verify the correct drivers are found, their code sections and/or export tables could be parsed to identify known/required functions.

Now that the drivers have been located, there is an additional initialization step that must be completed. It is useful to recap what is known about the crash dump stack at the point the system has booted up:

1. The crash dump drivers have been mapped into memory
2. The global dump structures have been initialized

All components have been initialized, but an important step has not been completed: calling the dump drivers' entry points. This isn't done by the operating system until a bug check actually occurs and the system is crashing. Thus, in order to use the crash dump stack outside of the operating system, the driver entry points must be called. Since they have been located in memory, it is simply a matter of finding their module information (e.g., by walking the load driver module list using the `PsLoadedModuleList` technique) and calling their entry points with the correct structures. As discussed in prior sections, the port driver takes the dump initialization structure as a second argument and the miniport driver takes two null arguments.

Get Boot Device Information

The following boot device information is required in order to use the crash dump I/O path:

- Dump pointers – of type `DUMP_POINTERS` or `DUMP_POINTERS_EX` obtained via a `IOCTL_SCSI_GET_DUMP_POINTERS` query to the boot device; this query returns register mapping information and hardware port configuration information that is used by the miniport to program the device.
- SCSI address – obtained via `IOCTL_GET_SCSI_ADDRESS`; contains PathId, TargetId, and Lun path information of the underlying device which are used in building the I/O request later in the technique.

Find the Dump Port Driver's StartIo or DispatchCrb Routine

At this point in the process, the crash dump drivers have been located and their entry points have been called. How does one send disk read/write requests to these drivers? To answer this question, it is necessary to quickly discuss how normal I/O is handled in Windows, since the first logical answer might be to try to communicate with the crash dump port driver's device object.

In Windows, I/O is processed by communicating with a logical representation of the physical hardware called a device object which is normally setup by the driver that runs the device. In the case of a miniport driver in the normal I/O stack, the device object is actually setup by the operating system's port driver on behalf of the miniport driver [3]:

In some technology areas, a minidriver that is associated with a class driver or port driver does not have to create its own device objects. Instead, the class or port driver creates the device object, and receives all IRPs for the device. The class or port driver then uses a driver-specific method to pass the I/O request to the minidriver.

For example, the disassembly of the normal I/O path's SCSI port driver's (`scsiport.sys`) `ScsiPortInitialize()` function, which all miniport drivers call in their `DriverEntry()` routine, shows that the port driver takes care of setting up a named device for the physical hardware hosted by the miniport driver:

```
loc_1E2CD:
push    [ebp+var_88]
lea     eax, [ebp+SourceString]
push    offset aDeviceScsiport ; "\\Device\\ScsiPort%d"
push    eax             ; wchar_t *
call    ebx ; __imp__swprintf
add     esp, 0Ch
lea     eax, [ebp+SourceString]
push    eax             ; SourceString
lea     eax, [ebp+SymbolicLinkName]
push    eax             ; DestinationString
call    edi ; RtlInitUnicodeString(x,x) ; RtlInitUnicodeString(x,x)
lea     eax, [ebp+DestinationString]
push    eax             ; DeviceName
lea     eax, [ebp+SymbolicLinkName]
push    eax             ; SymbolicLinkName
call    ds:__imp__IoCreateSymbolicLink@8 ; IoCreateSymbolicLink(x,x
```

Figure 9: The Normal I/O Path Port Driver configures the miniport's device object

However, this code does not exist in the dump port driver, which corroborates the discovery (discussed in the crash dump initialization section) that the dump drivers are loaded without creating any associated `DRIVER_OBJECT` or `DEVICE_OBJECT` structures. Thus, the dump stack does not have a device object associated with it, and therefore traditional means of I/O will not work. This makes perfect sense, because in normal use of the crash dump, the I/O subsystem would be disabled due to the system crash.

For SCSI devices, another possible way to send I/O to the crash dump port driver is through the dump port driver's `StartIo` function. The standard `StartIo` function as described on MSDN [6]:

As its name suggests, a StartIo routine is responsible for starting an I/O operation on the physical device. Most lowest-level drivers provide a StartIo routine and rely on the I/O manager to queue IRPs to a system-supplied device queue. Some lowest-level drivers are designed to set up and manage their own supplemental IRP queues, but even these usually also provide a StartIo routine.

As was already discussed, the I/O manager can't be used to send IRP's to the dump port driver's `StartIo` function. As the description above mentioned, some lowest-level drivers have their own internal queueing mechanism, and since the dump drivers are designed to operate in a special mode completely divorced of the I/O subsystem, it might be possible to call the `StartIo` function directly.

Analysis shows that the `StartIo` function in the dump port driver takes a single argument (a deviation from the standard `StartIo` prototype, which takes two arguments), a SCSI Request Block (SRB). In the Normal I/O path, I/O requests traverse the storage device stack down to a storage class driver, which converts the corresponding IRP to an SRB, which is passed to the storage port driver. The port driver passes the SRB onto the miniport driver, which translates the request into a format specific to its hardware. But in the crash dump scenario case, rather than an IRP being built by the I/O manager, it is possible to build the SRB directly using hardware-specific information needed to complete the request. This structure can then be sent directly to the dump port driver, which passes it to the miniport for completion.

The `StartIo` routine is optional for all drivers, and although it is included in the crash dump port driver, it is not exported. However, the function can be dynamically located by scanning the code (".text") section of the driver's image in memory for some recognizable bytes.

For IDE devices, the overall process is similar, but instead of `StartIo` with a single SRB argument, the `dumpata` port driver uses the internal function `DispatchCrb` which takes a single argument, a channel extension . This function must also be dynamically located.

Find the Dump Port Driver's Device Extension

At this point in the process, the crash dump drivers have been located and initialized, and the dump port driver's `StartIo/DispatchCrb` function has been located, which will be used to send I/O requests to the device. Before sending the request, there is some additional initialization that needs to take place.

One of the critical pieces of this technique is manipulating the device extension structure which is passed between and used by both the dump port and miniport drivers. For SCSI devices, the device extension structure contains similarities between the partially documented type `HW_INITIALIZATION_DATA` and the `SCSI_PORT_DEVICE_EXTENSION` from ReactOs [7]. For IDE devices, the structure is a channel extension structure, which stores information about a particular channel on an IDE controller. For both cases, it is not necessary to know all of the details about what is

contained in these structures, as most of the required fields are populated when the dump port and miniport `DriverEntry` routines are called.

However, it is necessary to manually initialize the field that represents the I/O request being sent. Normally this field is setup by the `DiskDumpWrite` callback when crash dump data is being written, but this function obviously is not going to be called when using the crash dump stack independent of the crash dump process. Because the device extension is an internal (not exported) variable, its address must be located through some other means. Basically an "information leak" is needed – that is, an instance in some function where a pointer to the device extension is stored in a register that survives the function. Fortunately, such a leak exists in dump port drivers for both SCSI and IDE. As an example, for the SCSI dump port driver `diskdump.sys`, the leak exists in the `DiskDumpOpen` function, which is a callback used by the kernel to prepare the crash dump file:

```
; __stdcall DiskDumpOpen(x, x)
_DiskDumpOpen@8 proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
mov     ecx, _DeviceExtension
mov     eax, [ebp+arg_0]
mov     [ecx+10h], eax
mov     eax, [ebp+arg_4]
mov     ecx, _DeviceExtension
mov     [ecx+14h], eax
mov     al, 1
pop     ebp
retn    8
_DiskDumpOpen@8 endp
```

Figure 10: Information leak in DiskDumpOpen callback of diskdump.sys

In the disassembly above, a pointer to the internal device extension variable `_DeviceExtension` is stored in the `ecx` register and is never cleared. Therefore, simply calling the `DiskDumpOpen` callback will provide the pointer to the internal variable. At this point, the field can be manually allocated and the request sent. This information "leak" exists on all versions of Windows in the SCSI dump port driver `diskdump.sys`, but the offset is different for 64-bit architectures. Table 3 below shows where these leaks exist according to transport and architecture.

Transport	Leaking Function	Leaked in Register	Architecture
SCSI/Storoprt (diskdump.sys)	DiskDumpOpen	ecx	x86
SCSI/Storport (diskdump.sys)	DriverEntry	rdx	x64
IDE (dumpata.sys)	IdeDumpOpen	ecx	x86

IDE (dumpata.sys)	IdeDumpOpen	rcx	x64

Table 3: Locations of Device Extension Leaks

Instantiate and Transmit SCSI/IDE Request Block

After locating the dump port driver's device extension, the field that represents the I/O request must be initialized.

For the SCSI dump port driver, this field is an MDL stored in the device extension. The MDL describes the buffer to be read from or written to pertaining to the requested operation (the `DataBuffer` field of the SRB). The MDL is one of the arguments to `DiskDumpWrite`, whose prototype is:

```
typedef
NTSTATUS
(*PDUMP_DRIVER_WRITE) (
    IN PLARGE_INTEGER DiskByteOffset,
    IN PMDL Mdl
    );
```

Figure 11: `DiskDumpWrite` prototype

`DiskDumpWrite` in turn calls `MmMapMemoryDumpMdl()` to map the MDL into a fixed address space, as shown in the disassembly below, and the resulting pointer is stored at offset `0xD0` (`0x118` on x64).

```
unsigned int __stdcall DiskDumpWrite(int byteOffset, int pMDL)
{
  _MDL *_pMdl; // ebx@1
  _SCSI_REQUEST_BLOCK *pSrb; // esi@1
  unsigned __int8 Lun; // al@4
  unsigned __int32 v5; // eax@7
  unsigned int v7; // [sp+Ch] [bp-4h]@1
  int cdb; // [sp+1Ch] [bp+Ch]@7

  v7 = 0;
  _pMdl = (_MDL *)pMDL;
  pSrb = (_SCSI_REQUEST_BLOCK *)(DeviceExtension + 0x48);
  LOWORD(_pMdl->StartVa) &= 0xF000u;
  while ( 1 )
  {
    if ( v7 )
      MmMapMemoryDumpMdl((int)_pMdl);
    memset(pSrb, 0, 0x40u);
    *(_DWORD *)(DeviceExtension + 0xD0) = _pMdl;
```

Figure 12: `DiskDumpWrite` stores the MDL in the device extension

The `DiskDumpWrite` callback is not used by the technique described in this paper, because it carries out only a write operation. Instead, the `StartIo` function, whose disassembly is shown below, is used to execute an arbitrary SRB.

```
int __stdcall StartIo(_SCSI_REQUEST_BLOCK *srb)
{
  char flag; // zf@1

  flag = (srb->SrbFlags & 0xC0) == 0;
  srb->SrbExtension = *(void **)(DeviceExtension + 0x204);
  if ( !flag && *(_BYTE *)(DeviceExtension + 681) )
    AllocateScatterGatherList(DeviceExtension, srb);
  *(_DWORD *)(DeviceExtension + 0xD8) = srb->TimeOutValue;
  return (*(int (__stdcall **)(_DWORD, _SCSI_REQUEST_BLOCK *))(DeviceExtension
                                                             + 0x298))(// <miniport>!HwStartIo
           *(_DWORD *)(DeviceExtension + 0x2AC),
           srb);
}
```

Figure 13: `StartIo` also allocates the MDL via `AllocateScatterGatherList()`

`StartIo()` stores the provided MDL at the same offset as `DiskDumpWrite` via `AllocateScatterGatherList()`. It calls `IoMapTransfer()` to setup map registers for the SCSI disk's adapter object to map a DMA transfer from the disk to system memory via the locked-down buffer in the MDL.

Thus, to use the crash dump stack for SCSI drives, it is a simple matter of crafting an SRB, creating an MDL for it at the appropriate offset, and sending the SRB directly to `StartIo`. The format of an SRB (documented structure) is shown below.

```
typedef struct _SCSI_REQUEST_BLOCK {
  USHORT  Length;
  UCHAR  Function;
  UCHAR  SrbStatus;
  UCHAR  ScsiStatus;
  UCHAR  PathId;
  UCHAR  TargetId;
  UCHAR  Lun;
  UCHAR  QueueTag;
  UCHAR  QueueAction;
  UCHAR  CdbLength;
  UCHAR  SenseInfoBufferLength;
  ULONG  SrbFlags;
  ULONG  DataTransferLength;
  ULONG  TimeOutValue;
  PVOID  DataBuffer;
  PVOID  SenseInfoBuffer;
  struct _SCSI_REQUEST_BLOCK  *NextSrb;
  PVOID  OriginalRequest;
  PVOID  SrbExtension;
  union {
        ULONG  InternalStatus;
        ULONG  QueueSortKey;
  };
  UCHAR  Cdb[16];
} SCSI_REQUEST_BLOCK, *PSCSI_REQUEST_BLOCK;
```

Figure 14: The SCSI_REQUEST_BLOCK structure

The highlighted fields will need to be populated before sending the request:

- `Length` - the size of the SRB structure
- `Function` – the value `SRB_FUNCTION_EXECUTE_SCSI` which directs the device to execute a SCSI-2 compliant command descriptor block

- `PathId`, `TargetId`, `Lun` – the corresponding values stored in a `SCSI_ADDRESS` structure which describe the path to the boot device
- `CdbLength` – the number of bytes in the attached `Cdb` command
- `SrbFlags` – Flags that describe the type of requested operation
- `DataTransferLength` – the number of bytes to be read from or written to disk
- `TimeOutValue` – time in seconds for the operation to timeout
- `DataBuffer` – pointer to system space memory where the device will copy to or read from
- `Cdb` – the SCSI-2 compliant command descriptor block (CDB) which describes the SCSI operation the device will carry out

The CDB field describes the requested SCSI operation, and the values stored there will vary based on the type of operation. For example, to read the boot sector (bytes 0 through 512 in first sector), a 10-byte command could be used in order to support 32-bit logical block address (LBA). In such an example, the `SRB.CdbLength` field would be set to 10 and the CDB structure would be filled out as shown in the illustration below. The complete SCSI-2 specification is available on the internet at various sources including [4].

```
Srb.Cdb[0] = SCSIOP_READ;              //operation
Srb.Cdb[1] = 0;                        //LUN-DPO-FUA-Reserved-RelAddr
Srb.Cdb[2] = 0;                        //LBA (MSB)
Srb.Cdb[3] = 0;                        //..
Srb.Cdb[4] = 0;                        //..
Srb.Cdb[5] = 0;                        //LBA (LSB)
Srb.Cdb[6] = 0;                        //reserved
Srb.Cdb[7] = 0;                        //MSB transfer length (# blocks, 1 block = 512 bytes)
Srb.Cdb[8] = 1;                        //LSB transfer length
Srb.Cdb[9] = 0;                        //control byte
```

Figure 15: SRB to read the MBR

Now that the SRB has been built, it is a simple matter of passing it to `StartIo`. Once the request is completed, the result will be stored in the `DataBuffer` field of the SRB.

For the IDE dump port driver (`dumpata.sys`), the device extension is of a format similar to an IDE channel extension structure. It is allocated in the `IdeDumpPortInitialize` routine, which is called from the dump port driver's `DriverEntry`. The device extension structure (CRB) is stored in the `DUMP_INITIALIZATION_CONTEXT` structure passed to it by the `crashdmp.sys` driver. In the disassembly below, the device extension of size `0x8000` is stored in the `MemoryBlock` field of the `DUMP_INITIALIZATION_CONTEXT` structure.

```
int __stdcall IdeDumpPortInitialize(int a1, DUMP_INITIALIZATION_CONTEXT *dumpInit)
{
  void *portConfig; // edi@1
  void *v3; // eax@1
  char *v4; // edi@1

  portConfig = dumpInit->PortConfiguration;
  dumpInit->OpenRoutine = IdeDumpOpen;
  dumpInit->WriteRoutine = IdeDumpWrite;
  dumpInit->FinishRoutine = IdeDumpFinish;
  dumpInit->WritePendingRoutine = IdeDumpWritePending;
  dumpInit->MaximumTransferSize = *((_DWORD *)portConfig + 2);
  DumpExtension = dumpInit->MemoryBlock;
  memset(DumpExtension, 0, 0x8000u);
```

Figure 16: `IdeDumpPortInitialize()` stores the dump extension in the `MemoryBlock` field

When the `DiskDumpWrite` callback is called, it calls `IdeDumpWritePending()`, whose disassembly is shown below.

```
int __stdcall IdeDumpWritePending(int unknown_zero, int diskOffset, _MDL *pMdl, int unknown2_zero)
{
  int result; // eax@2
  int crb; // esi@5
  unsigned int v6; // edx@6
  int v7; // eax@7

  if ( unknown_zero == 4 )
  {
    result = 0;
  }
  else
  {
    if ( unknown_zero != 1 && unknown_zero )
    {
      crb = (int)((char *)DumpExtension + 288);
    }
    else
    {
      crb = IdeDumpAllocateCrb(DumpExtension);  // just memsets the IRB at offset 0x288
      v6 = *((_DWORD *)DumpExtension + 66);
      *(_DWORD *)(crb + 0x2F8) = 0;
      *(_QWORD *)(crb + 0x2F0) = (*(_QWORD *)diskOffset + *((_QWORD *)DumpExtension + 140)) / (signed __int64)v6;
      pMdl->StartVa = (void *)((unsigned int)pMdl->MappedSystemVa & 0xFFFFF000);
      IdeDumpSetupWriteCrb(
        (int)DumpExtension,
        *(_DWORD *)(crb + 0x2F0),
        *(_DWORD *)(crb + 0x2F4),
        (int)pMdl,                      |       // stuffs the given MDL at offset 0x50 in the CRB
        *(_DWORD *)(crb + 0x2F8),
        crb);
      DispatchCrb(crb);
    }
```

Figure 17: `IdeDumpWritePending()` initializes the controller extension

`IdeDumpWritePending()` sets up a write IRB in the CRB as follows:

- Initializes a CRB at offset `0x120` (`0x1C0` for x64) in the device extension (CRB)
- Calls `IdeDumpSetupWriteCrb()` which initializes a write IDE_REQUEST_BLOCK and calls `IdeDumpAddMdlToCrb()` to save the MDL passed to the `DiskDumpWrite` callback at offset `0x50` (`0x88` for x64)
- Calls `DispatchCrb()` to complete the I/O write request

- Waits for the request to complete by calling `IdeDumpWaitOnRequest()`, which uses hardware polling to query the drive until it has either successfully completed the request or returned an error code

`DispatchCrb()` fills in the remaining fields of the IRB, performs the register mapping and sends the request to the miniport driver.

```
signed int __stdcall DispatchCrb(int crb)
{
  _IDE_REQUEST_BLOCK *irb; // esi@3
  PSINGLE_LIST_ENTRY irbExtension; // eax@3
  BOOLEAN v3; // al@3

  if ( *((_DWORD *)DumpExtension + 276) )
  {
    *((_DWORD *)DumpExtension + 264) = crb;
  }
  else
  {
    irb = (_IDE_REQUEST_BLOCK *)(crb + 0x288);
    *((_DWORD *)DumpExtension + 275) = 1000 * *(_DWORD *)(crb + 668);
    irbExtension = IdeDumpAllocateIrbExtension((int)DumpExtension);
    irb->IrbExtension = irbExtension;
    irb->SenseInfoBufferLength = 24;
    irb->SenseInfoBuffer = (PVOID)IdeDumpAllocateSenseBuffer(DumpExtension, irbExtension);
    v3 = (unsigned __int8)DumpExtension;
    if ( *((_DWORD *)DumpExtension + 45) )
    {
      (*((void (__stdcall **)(_DWORD, _DWORD))DumpExtension + 45))(*(_DWORD *)DumpExtension, irb);// dump_atapi!HwAtapiBuildIo
      v3 = (unsigned __int8)DumpExtension;
    }
    if ( *(_BYTE *)(crb + 0x298) & 0xC0 )
    {
      IdeDumpAllocateScatterGatherList(v3, crb);
      IdeDumpMapBuffers((_MDL *)crb);
    }
    *(_WORD *)(crb + 16) |= 1u;
    if ( *(_BYTE *)(crb + 650) )
      AtaPortCompleteRequest(*(_DWORD *)DumpExtension, irb);
    else
      (*((void (__stdcall **)(_DWORD, _DWORD))DumpExtension + 46))(*(_DWORD *)DumpExtension, irb);// dump_atapi!AtapiHwStartIo
    IdeDumpCompletionDpc(DumpExtension);
  }
  return 259;
}
```

Figure 18: `DispatchCrb()` completes initialization and transmits I/O to the miniport

As shown in the function disassembly above, `IdeDumpAllocateScatterGatherList()` is called. This function performs the same task as the `AllocateScatterGatherList()` function in the SCSI dump port driver. `IdeDumpMapBuffers()` locks the buffers into system space, then the request is sent to the miniport driver via the miniport's `StartIo()` function (in this case, the `dump_atapi` dump miniport's `AtapiHwStartIo()`). The `DispatchCrb()` function finally calls `IdeDumpCompletionDpc()` which calls a completion function stored at offset `0x4` in the CRB.

A simpler example of instantiating an IRB and using `DispatchCrb()` inside the dump port driver can be found in `IdeDumpIssueIdentify()`, shown below. This function is used to issue an ATA identify request to retrieve meta information about an IDE drive, such as channel, drive number, serial number, etc.

```
char __stdcall IdeDumpIssueIdentify(int a1, int a2)
{
  int crb; // esi@1
  _MDL *v3; // eax@1
  char driveNumber; // zf@1

  crb = (int)IdeDumpAllocateCrb((int)DumpExtension);
  v3 = IdeDumpAllocateMdl((int)DumpExtension, 0x200u);
  IdeDumpAddMdlToCrb(crb, v3, 0);
  *(_DWORD *)(crb + 8) = a2;
  *(_DWORD *)(crb + 4) = IdeDumpIdentifyCompletion;
  *(_WORD *)(crb + 0x288) = 0x101u;             // crb+0x288 is the IDE_REQUEST_BLOCK (IRB)
                                                // 0x101 = IRB_FUNCTION_ATA_IDENTIFY, first field in IDE_REQUEST_BLOCK
  *(_BYTE *)(crb + 0x28D) = *((_BYTE *)DumpExtension + 138);// Irb.Channel
  *(_BYTE *)(crb + 0x28E) = *((_BYTE *)DumpExtension + 1117);// Irb.TargetId
  *(_BYTE *)(crb + 0x28F) = *((_BYTE *)DumpExtension + 1118);// Irb.Lun
  driveNumber = *(_BYTE *)(crb + 0x28E) == 0;
  *(_DWORD *)(crb + 0x298) = *(_DWORD *)(crb + 0x298) & 0xFFFFFFFD | 0x40;// Irb.Flags (0x40 = DATA_IN)
  *(_DWORD *)(crb + 0x29C) = 1;                 // Irb.TimeOutValue
  *(_BYTE *)(crb + 0x2BD) = ((!driveNumber - 1) & 0xF0) - 0x50;// Irb.TaskFile.bDriveHeadReg
  *(_BYTE *)(crb + 0x2BE) = 0xECu;              // Irb.TaskFile.bCommandReg (0xEC = IDE_COMMAND_IDENTIFY)
  DispatchCrb(crb);
  return IdeDumpWaitOnRequest(crb, 0) >= 0;
}
```

`IdeDumpIssueIdentify()` provides a good template for using the crash dump stack for IDE drives. The process can be summarized as follows:

- Allocate a CRB in `DUMP_INITIALIZATION_CONTEXT.MemoryBlock` at the correct offset (`0x120` for x86, `0x1C0` for x64)
- Store a pointer to a callback function in the CRB at offset `0x4`, which will be invoked when the dump port driver is notified that the I/O request is complete
- Allocate and fill in an IRB at offset `0x288` (`0x3E8` for x64) in the CRB
- Allocate an MDL at offset `0x50` (`0x88` for x64) in the CRB
- Send the CRB to `DispatchCrb()`
- Call a custom function that implements a polling mechanism to wait for the request to complete, or call the internal `IdeDumpWaitOnRequest()` function

There is a second method for using the crash dump stack on IDE drives without interacting with the port driver at all:

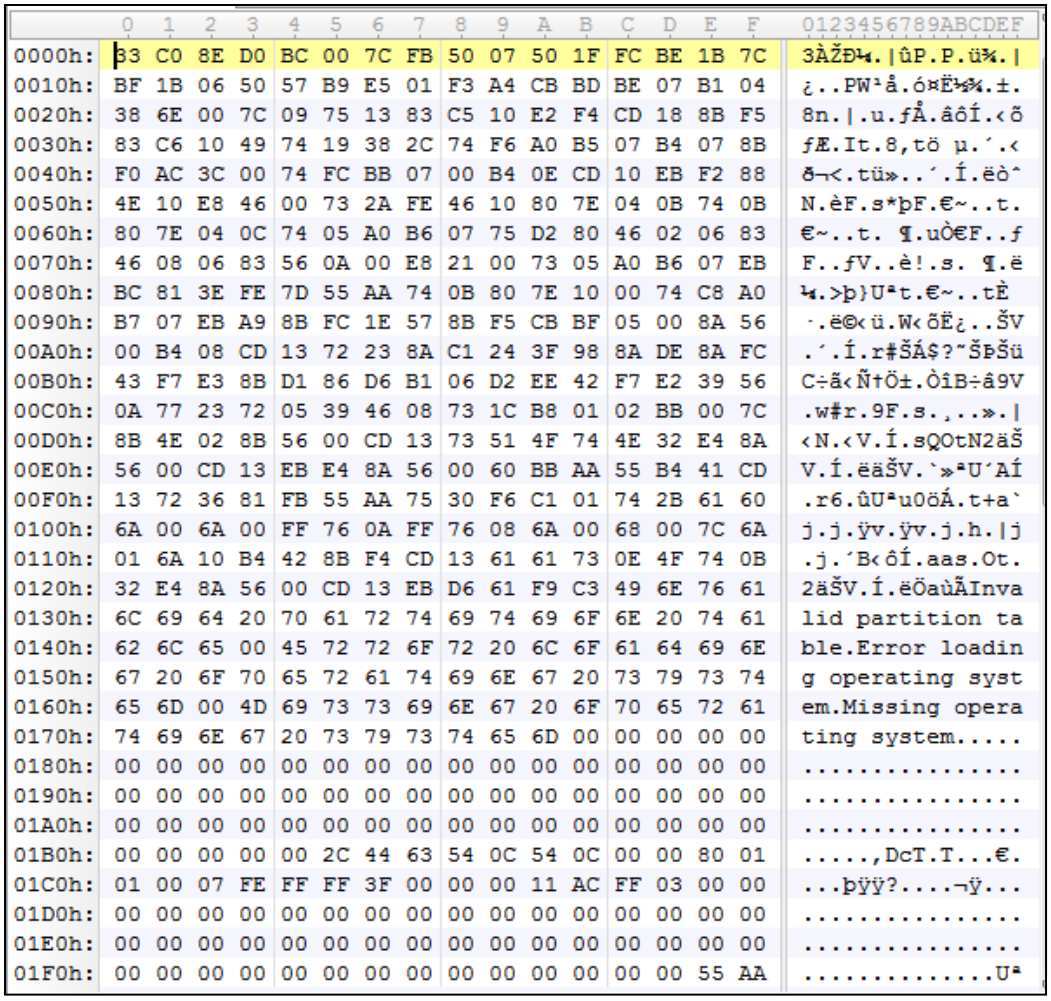
- Allocate a CRB in `DUMP_INITIALIZATION_CONTEXT.MemoryBlock` at the correct offset (`0x120` for x86, `0x1C0` for x64)
- Store a pointer to a callback function in the CRB at offset `0x4`, which will be invoked when the dump port driver is notified that the I/O request is complete
- Allocate and fill in an IRB at offset `0x288` (`0x3E8` for x64) in the CRB
- Call the miniport's `HwStartIo` routine, which is stored at offset 0x2E in the device extension, passing the device extension and the IRB
- Poll the device until the IRB status changes from zero by calling the miniport's `HwInterrupt` routine which is stored at offset 0x2F in the device extension, passing the device extension only

The IDE request block is a documented format available at [8]. Most of the fields are comparable to the format of the SRB with the following key exceptions which are evident in the disassembly above:

- The `Function` field should be set to `IRB_FUNCTION_ATA_COMMAND`
- The `Channel` field should be set to the value stored in the channel extension's `Channel` field which is at offset `0x8A` (`0xEA` for x64) from the start of the `DUMP_INITIALIZATION_CONTEXT.MemoryBlock` structure
- The `TargetId` field should be set to the value stored in the channel extension's `TargetId` field which is at offset `0x45D` (`0x6A9` for x64) from the start of the `DUMP_INITIALIZATION_CONTEXT.MemoryBlock` structure
- The `Lun` field should be set to the value stored in the channel extension's `Lun` field which is at offset `0x45E` (`0x6AA` for x64) from the start of the `DUMP_INITIALIZATION_CONTEXT.MemoryBlock` structure

Subverting the TDL4 Bootkit

The TDL4 bootkit contaminates the normal I/O path by overwriting pointers in the device object of the port and miniport drivers with pointers to its own I/O filtering functions. These functions watch for requests to read the master boot record (located at sector 0) and return a copy of the clean MBR which it keeps in a secret place on disk. The purpose of this misdirection is to fool the investigator into believing the system is not infected, since a normal MBR is returned. The screenshot below shows the clean MBR as read by Sleuth Kit on an uninfected and an infected machine (as noted above, they should match).



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	33	C0	8E	D0	BC	00	7C	FB	50	07	50	1F	FC	BE	1B	7C	3ÀŽÐ¼.|ûP.P.ü¾.|
0010h:	BF	1B	06	50	57	B9	E5	01	F3	A4	CB	BD	BE	07	B1	04	¿..PW¹å.ó¤Ë½¾.±.
0020h:	38	6E	00	7C	09	75	13	83	C5	10	E2	F4	CD	18	8B	F5	8n.|.u.ƒÅ.âôÍ.‹õ
0030h:	83	C6	10	49	74	19	38	2C	74	F6	A0	B5	07	B4	07	8B	ƒÆ.It.8,tö µ.´.‹
0040h:	F0	AC	3C	00	74	FC	BB	07	00	B4	0E	CD	10	EB	F2	88	ð¬<.tü»..´.Í.ëò^
0050h:	4E	10	E8	46	00	73	2A	FE	46	10	80	7E	04	0B	74	0B	N.èF.s*þF.€~..t.
0060h:	80	7E	04	0C	74	05	A0	B6	07	75	D2	80	46	02	06	83	€~..t. ¶.uÒ€F..ƒ
0070h:	46	08	06	83	56	0A	00	E8	21	00	73	05	A0	B6	07	EB	F..ƒV..è!.s. ¶.ë
0080h:	BC	81	3E	FE	7D	55	AA	74	0B	80	7E	10	00	74	C8	A0	¼.>þ}Uªt.€~..tÈ
0090h:	B7	07	EB	A9	8B	FC	1E	57	8B	F5	CB	BF	05	00	8A	56	·.ë©‹ü.W‹õË¿..ŠV
00A0h:	00	B4	08	CD	13	72	23	8A	C1	24	3F	98	8A	DE	8A	FC	.´.Í.r#ŠÁ$?˜ŠÞŠü
00B0h:	43	F7	E3	8B	D1	86	D6	B1	06	D2	EE	42	F7	E2	39	56	C÷ã‹Ñ†Ö±.ÒîB÷â9V
00C0h:	0A	77	23	72	05	39	46	08	73	1C	B8	01	02	BB	00	7C	.w#r.9F.s.¸..».|
00D0h:	8B	4E	02	8B	56	00	CD	13	73	51	4F	74	4E	32	E4	8A	‹N.‹V.Í.sQOtN2äŠ
00E0h:	56	00	CD	13	EB	E4	8A	56	00	60	BB	AA	55	B4	41	CD	V.Í.ëäŠV.`»ªU´AÍ
00F0h:	13	72	36	81	FB	55	AA	75	30	F6	C1	01	74	2B	61	60	.r6.ûUªu0öÁ.t+a`
0100h:	6A	00	6A	00	FF	76	0A	FF	76	08	6A	00	68	00	7C	6A	j.j.ÿv.ÿv.j.h.|j
0110h:	01	6A	10	B4	42	8B	F4	CD	13	61	61	73	0E	4F	74	0B	.j.´B‹ôÍ.aas.Ot.
0120h:	32	E4	8A	56	00	CD	13	EB	D6	61	F9	C3	49	6E	76	61	2äŠV.Í.ëÖaùÃInva
0130h:	6C	69	64	20	70	61	72	74	69	74	69	6F	6E	20	74	61	lid partition ta
0140h:	62	6C	65	00	45	72	72	6F	72	20	6C	6F	61	64	69	6E	ble.Error loadin
0150h:	67	20	6F	70	65	72	61	74	69	6E	67	20	73	79	73	74	g operating syst
0160h:	65	6D	00	4D	69	73	73	69	6E	67	20	6F	70	65	72	61	em.Missing opera
0170h:	74	69	6E	67	20	73	79	73	74	65	6D	00	00	00	00	00	ting system.....
0180h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0190h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01A0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01B0h:	00	00	00	00	00	2C	44	63	54	0C	54	0C	00	00	80	01	.....,DcT.T...€.
01C0h:	01	00	07	FE	FF	FF	3F	00	00	00	11	AC	FF	03	00	00	...þÿÿ?....¬ÿ...
01D0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01E0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01F0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	.....Uª

Figure 19: The clean copy of the MBR as returned by TDL4

The proof-of-concept driver implementing the technique discussed in this paper for SCSI is loaded and the MBR is read. The output of the driver is shown below.

```
kape: IRP_MJ_DEVICE_CONTROL. Control Code 70ff801c
kape:  GetMBR():  Found dump driver entry points:
    dump_scsiport.sys:  f8150b85
    dump_vmscsi.sys:  f814bbbe
kape:  GetMBR():  Boot device:  LUN=0, TargetId=0, PathId=0, Port=2
kape:  GetMBR():  Boot device object at 82309598
kape:  GetMBR():  Contacting disk driver to get dump pointers...
kape:  GetMBR():  Sending device usage notification request...
kape:  GetMBR():  Allocating required buffers...
kape:  GetMBR():  Calling dump port and miniport driver entry points...success!
kape:  FindDriverStartIoAddress():  Text section at f814e300 (size 8102) .
kape:  GetMBR():  DriverStartIo found at address f814e648.
kape:  GetMBR(): Located dump port DeviceExtension pointer at 81d44010!
kape:  GetMBR(): Storing pointer to an allocated MDL at DeviceExtension address 81d440e0!
kape:  GetMBR(): Sending SRB...status:
     Srb.SrbStatus = 00000000
     Srb.ScsiStatus = 00000000
     Srb.InternalStatus = 00000000
kape:  Srb.DataBuffer:
33c08ed0bc007cfb5007501ffcbe1b7cbf1b065057b9e501f3a4cbbdbe07b104386e007c09751383c510e2f4cd188bf58
3c610497419382c74f6a0b507b4078bf0ac3c0074fcbb0700b40ecd10ebf2884e10e84600732afe4610807e040b740b80
```

```
7e040c7405a0b60775d280460206834608068356 0a00e821007305a0b607ebbc813efe7d55aa740b807e100074c8a0b70
7eba98bfc1e578bf5cbbf05008a5600b408cd1372238ac1243f988ade8afc43f7e38bd186d6b106d2ee42f7e239560a77
237205394608731cb80102bb007c8b4e028b5600cd1373514f744e32e48a5600cd13ebe48a560060bbaa55b441cd13723
681fb55aa7530f6c101742b61606a006a00ff760aff76086a0068007c6a016a10b4428bf4cd136161730e4f740b32e48a
5600cd13ebd661f9c3496e76616c6964207061727469746 96f6e207461626c6500457272 6f72206c6f6164696e67206f7
065726174696e672073797374656d004d697373696e67206f7065726174696e672073797374656d000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
02c4463540c540c00008001010007fefffff3f00000011acff0300000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000055aa
kape: OnStubDispatch
kape: OnStubDispatch
```

Figure 20: Output from Proof-of-Concept utility to retrieve hidden MBR

A view of this MBR in a hex editor is shown below. Visual inspection reveals numerous differences in this MBR and the previous one.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	33	C0	8E	D0	BC	00	7C	8E	C0	8E	D8	BE	00	7C	BF	00	3ÀŽÐ¼.|ŽÀŽØ¾.|¿.
0010h:	06	B9	00	02	FC	F3	A4	50	68	1C	06	CB	FB	60	B9	32	.¹..üó¤Ph..Ëû`¹2
0020h:	01	BD	2A	06	D2	4E	00	45	E2	FA	22	2C	5C	83	E0	C5	.½*.ÒN.Eâú",\ƒàÅ
0030h:	31	20	40	43	13	02	70	1C	60	1D	D1	0C	B4	24	AF	ED	1 @C..p.`.Ñ.´$¯í
0040h:	80	3E	18	DE	08	0F	00	B9	31	FD	59	0E	B9	03	00	1D	€>.Þ...¹1ýY.¹...
0050h:	FD	00	99	17	44	0A	FF	C6	47	30	1C	62	FF	74	85	00	ý.™.D.ÿÆG0.bÿt….
0060h:	EB	9C	20	17	0E	C7	02	7E	4A	0D	46	10	85	60	5D	BD	ëœ ..Ç.~J.F.…`]½
0070h:	16	57	00	00	00	00	18	D8	60	FA	20	20	C6	03	58	01	.W.....Ø`ú Æ.X.
0080h:	10	70	3E	0C	65	04	F1	C0	36	40	36	10	66	FF	8D	EF	.p>.e.ñÀ6@6.fÿ.ï
0090h:	80	33	3E	0C	67	04	99	FF	63	1C	20	CC	8F	03	DA	01	€3>.g.™ÿc. Ì..Ú.
00A0h:	66	02	99	52	06	B3	02	CC	38	F0	AD	10	00	5A	90	D7	f.™R.³.Ì8ð ..Z.×
00B0h:	F5	40	2A	2C	5C	83	73	62	AB	20	10	7D	67	04	4C	18	õ@*,\ƒsb« .}g.L.
00C0h:	13	FF	C4	B7	88	CF	57	E0	EF	1E	D5	F1	8A	C7	57	E0	.ÿÄ·ˆÏWàï.ÕñŠÇWà
00D0h:	20	20	00	91	89	E3	A2	B5	D5	38	22	5F	5D	83	22	B1	.'‰ã¢µÕ8"_]ƒ"±
00E0h:	D5	38	19	FD	CE	BA	01	25	6D	44	5B	FD	C3	BA	77	11	Õ8.ýÎº.%mD[ýÃºw.
00F0h:	E1	EA	20	11	1E	2F	02	D7	D8	40	EA	00	02	87	AD	C3	áê ../.×Ø@ê..‡ Ã
0100h:	D5	40	3C	6D	06	2F	02	26	FF	F7	0F	15	8F	AE	C1	00	Õ@<m./.&ÿ÷...®Á.
0110h:	8C	4C	1F	15	AD	AE	C1	11	D8	EA	1C	11	AF	AE	C1	00	ŒL.. ®Á.Øê..¯®Á.
0120h:	9E	81	B7	13	CF	45	63	AB	70	81	30	8C	4A	BA	B6	2C	ž.·.ÏEc«p.0ŒJº¶,
0130h:	3C	33	C4	81	E8	9F	FF	31	DC	DD	40	00	BE	C9	02	7E	<3Ä.èŸÿ1ÜÝ@.¾É.~
0140h:	6A	49	BB	0B	C9	3A	83	20	EC	1C	1B	20	29	F4	40	F9	jI».É:ƒ ì..)ô@ù
0150h:	98	4F	2D	EA	EA	E1	1B	8C	27	89	D8	00	73	79	73	74	˜O-êêá.Œ'‰Ø.syst
0160h:	65	6D	00	4D	69	73	73	69	6E	67	20	6F	70	65	72	61	em.Missing opera
0170h:	74	69	6E	67	20	73	79	73	74	65	6D	00	00	00	00	00	ting system.....
0180h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0190h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01A0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01B0h:	00	00	00	00	00	2C	44	63	54	0C	54	0C	00	00	80	01	.....,DcT.T...€.
01C0h:	01	00	07	FE	FF	FF	3F	00	00	00	11	AC	FF	03	00	00	...þÿÿ?....¬ÿ...
01D0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01E0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
01F0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	.....Uª

Figure 21: Contaminated TDL boot record hidden by rootkit

Disassembling this 16-bit boot sector code reveals a short decryption (`rol/xor`) routine typical of the TDL family. Thus, the concealed, infected MBR has been revealed by using the crash dump I/O path.

Caveats and Further Work

The technique described in this paper has been verified to work for SCSI transports (scsiport.sys and storport.sys) on x86 versions of Windows XP and 7. The strategies for IDE drives (ataport.sys) were

verified in so far as leveraging the crash dump stack to send an IRB to the device. The first technique of using `DispatchCrb` successfully transmitted an IRB to the device, but the result buffer was filled with garbage instead of the requested data. The second technique of using the miniport directly returned all zeroes in the result buffer and an unknown ATA status with IRB data length mismatch error. It is likely that either the IDE_TASK_FILE component of the IRB was improperly constructed or a required data length value is missing from the CRB or dump extension structure. 64-bit operating systems were not tested due to time constraints, but this is a simple exercise in validating the offset adjustments are correct.

Some storage driver professionals have mentioned on various online forums that attempting to call a driver's `StartIo` directly, rather than going through the I/O manager, could violate design principles of the target driver, resulting in system deadlocks [9]. Even though the design constraints of the crash dump environment (synchronous I/O, single-threaded, single processor) allow for such a violation, activating and using the crash dump mechanism during normal system operation could theoretically result in undefined behavior.

References

[1] http://www.sleuthkit.org/
[2] http://msdn.microsoft.com/en-us/library/ff564084(v=VS.85).aspx
[3] http://msdn.microsoft.com/en-us/library/windows/hardware/ff542862(v=vs.85).aspx
[4] http://ldkelley.com/SCSI2/SCSI2/SCSI2-07.html
[5] http://msdn.microsoft.com/en-us/library/windows/hardware/ff553110(v=vs.85).aspx
[6] http://msdn.microsoft.com/en-us/library/windows/hardware/ff566404(v=vs.85).aspx
[7] http://doxygen.reactos.org/d5/d38/scsiport__int_8h_source.html
[8] http://msdn.microsoft.com/en-us/library/windows/hardware/ff559140(v=vs.85).aspx
[9] http://www.osronline.com/showthread.cfm?link=183317

Resources and Further Reading

The Windows Research Kernel (WRK) and various operating system source code repositories on the internet (such as ReactOs) might provide useful information for reversing the crash dump mechanism:

- sysload.c – loader functions such as MmLoadSystemImage
- dumpctl.c – crash dump control functions
- io.h – I/O header defs
- hal.h – useful macros, header defs
- internal.c – driver object creation and dynamic loading
- ioinit.c, iodata.c

The WRK can be downloaded at
https://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=7366&c1=en-us&c2=0&Login=&wa=wsignin1.0

Other sources of potential value:

- ReactOs:
  - scsiport.c, scsiport.h – SCSIPort implementation
- Win DDK:
  - wdm.h
  - ntddscsi.h
  - SPTI example for sending SRBs from user mode (spti.h, spti.c)
  - storport.h
  - ata.h, irb.h
- classpnp example for sending SRBs from a driver

Additionally, Microsoft has published numerous guides for storage manufacturers on best practices for writing ATAPort, SCSIPort and StorPort miniport drivers. These documents contain low-level tips and gotchas straight from the kernel driver developers at Microsoft and provide invaluable insight.