I/O, You own: Regaining control of your disk in the presence of bootkits

Aaron LeMasters
MANDIANT

Agenda and Introduction

- Introduction
  - How this relates to my work at Mandiant
- Terminology
- How the operating system initializes and uses the crash dump stack:
  - Pre-Vista
  - Post-Vista
- How the crash dump stack can be used outside of the operating system
- Demo: defeating TDL4
- Disclaimer: some of this might be wrong…
- Note: Glossing over details – please read the whitepaper!

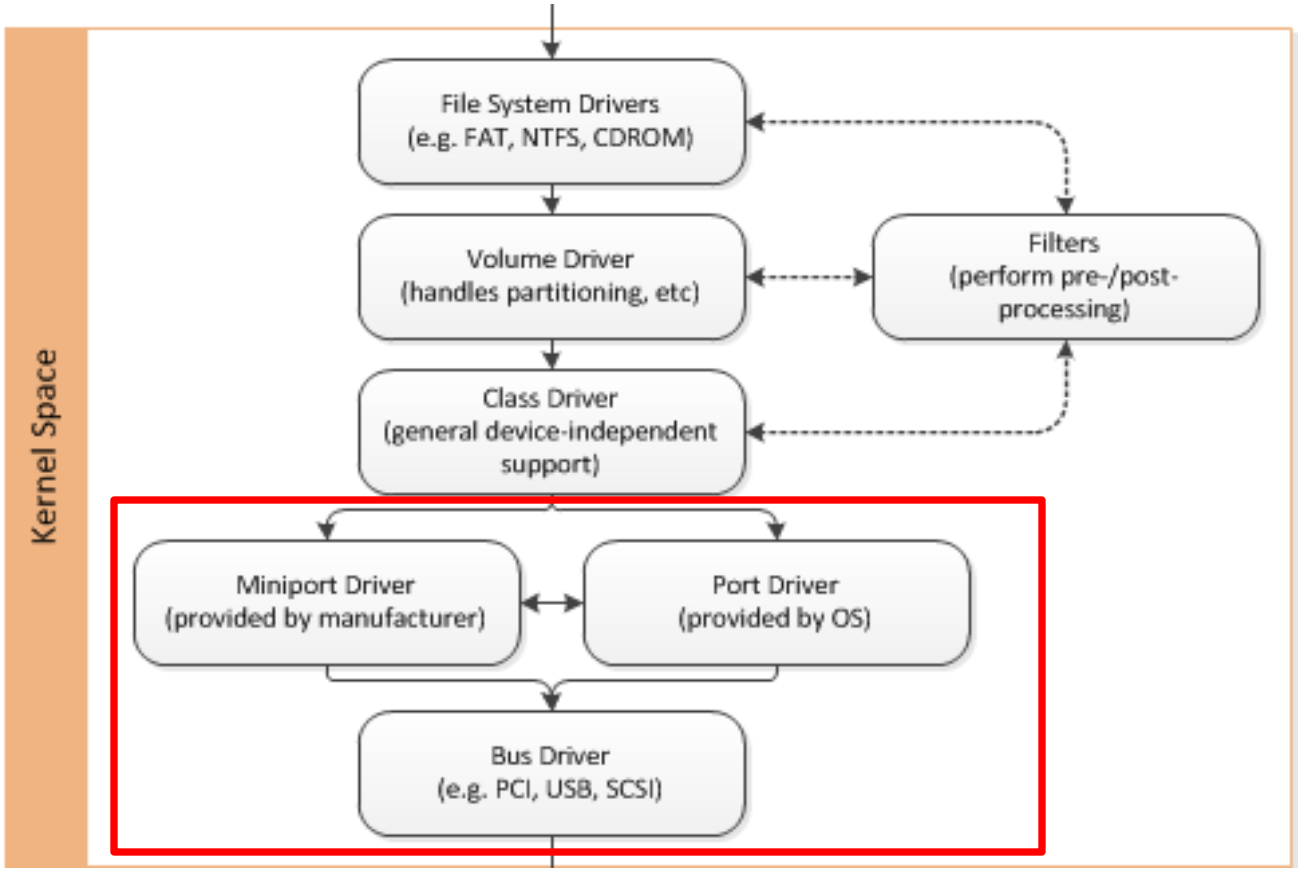
Terminology

Basics

- Port driver – abstraction interface provided by OS, hides underlying protocol details from class driver
- Miniport driver – manufacturer-supplied driver to interface with hardware (Host Bus Adapter/HBA); linked against port driver for specific transport technology
- Class driver – a driver that abstracts the underlying technology of a category of devices that share similar qualities (e.g., cdrom.sys)
- Normal I/O path – the route an I/O request takes during regular system operation
- Crash dump I/O path – the route the kernel uses to write a crash dump file to disk during a crash dump

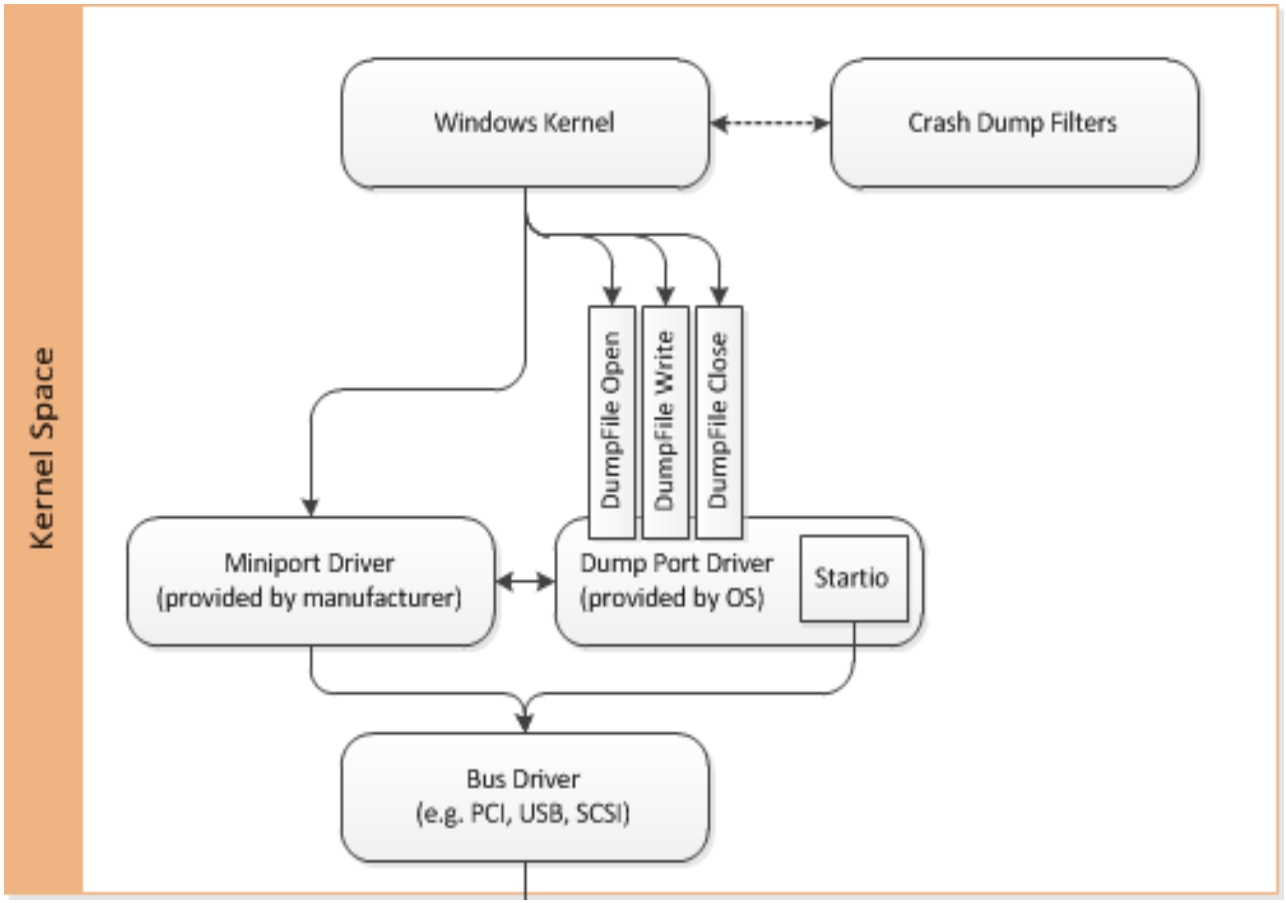
Terminology

Normal I/O Path



Terminology

Crash Dump I/O Path



Terminology

Why two paths?

- When a bugcheck occurs, the OS has no idea where the problem occurred – it could have happened inside a driver in the normal I/O path.
- What differs between the two paths?

	Normal I/O Path	Crash Dump I/O Path
Primary drivers	Many, layered	Modified port and miniport
Filter drivers	Many, layered	Crash dump filters only
Controlled by	I/O manager	Kernel or crashdmp.sys
Documented?	Yes	*cough*

Terminology
The Crash Dump Mechanism

MANDIANT

- Encompasses the entire crash dump process
  - From when it is initialized during system boot up to when it is used after `KeBugCheck2()`
- Primary components:
  - The kernel
  - `Crashdmp.sys` (Vista+)
  - The crash dump driver stack or just “crash dump stack”

Terminology

The Crash Dump Stack

- A “stack” of drivers, consisting of:
  - A dump port driver
  - A dump miniport driver
  - One or more crash dump filter drivers
- Initialized in two phases:
  - System startup/page file creation (pre-initialization)
  - System crash (post-initialization)
- Used when:
  - A bug check occurs
  - The system is about to hibernate

The Crash Dump Driver Stack

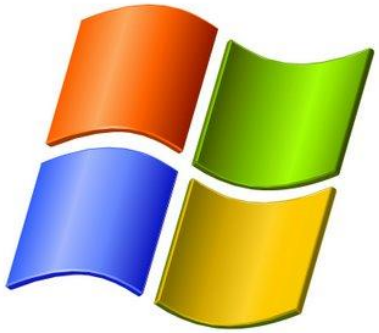
Common Drivers

MANDIANT

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

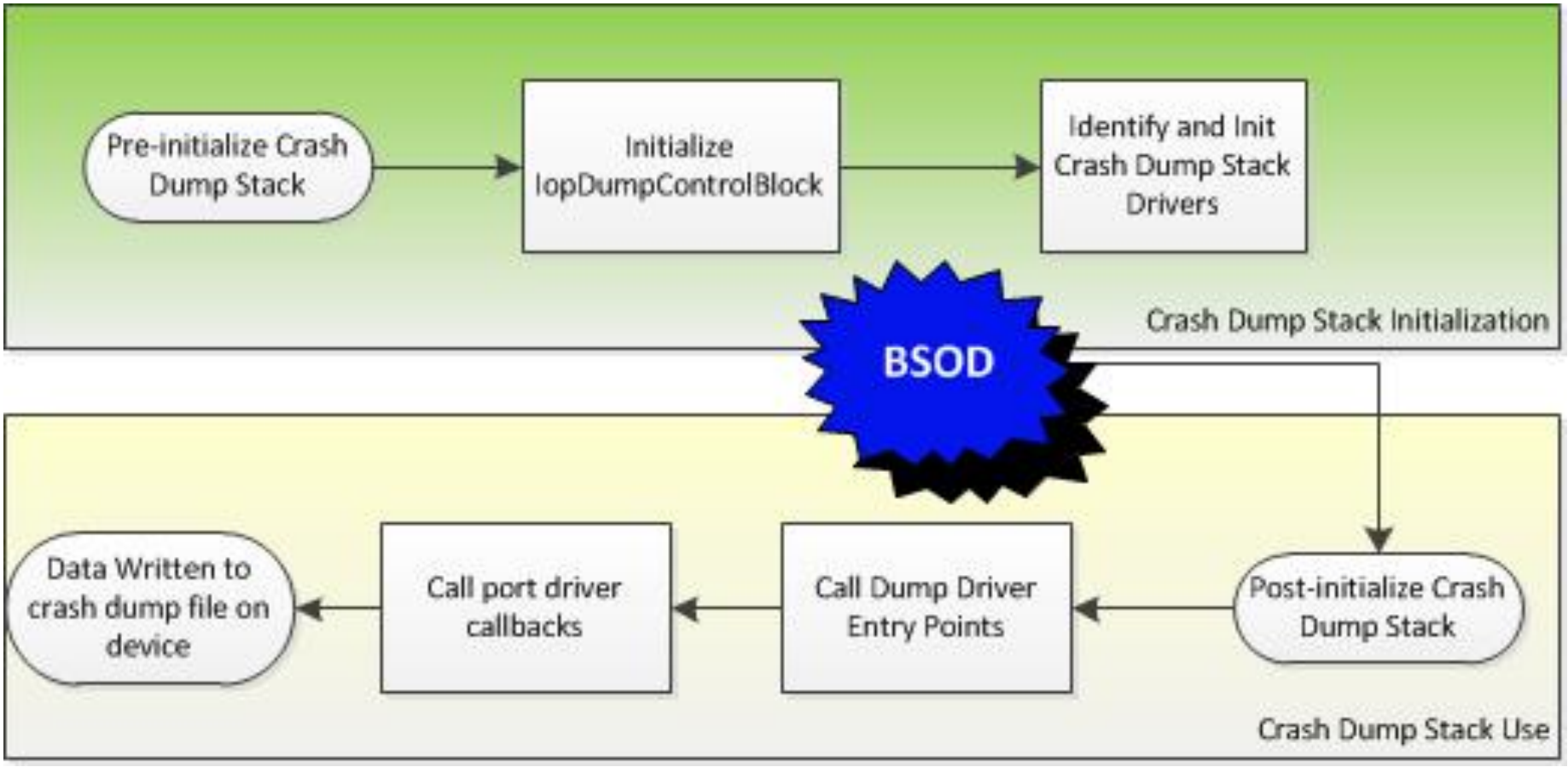
Crash Dump Environment

- Normal I/O path is disabled
- All processors are disabled except the one the current thread is executing on
- Active CPU becomes single-threaded (IRQL is raised to HIGH_LEVEL) and uninterruptible
- I/O sent to the crash dump stack is synchronous
- If IDE controller, only the channel containing the device with the paging file is enabled



Crash Dump Stack Initialization and Usage, Pre-Vista

Initialization and Use



Pre-Initialize Crash Dump Stack

Initialize IopDumpControlBlock

- `KiInitializeKernel()` → `IoInitSystem()` OR `NtCreatePagingFile()`:
  - `IoInitializeCrashDump()`:
    - `IopInitializeDCB()`:
      - Allocate `IopDumpControlBlock` structure
      - Fill in basic debug information - # CPUs, architecture, OS version, etc
      - Read registry settings for crash dump configuration

Pre-Initialize Crash Dump Stack

IopDumpControlBlock

MANDIANT

IopDumpControlBlock – global kernel variable

```
typedef struct _DUMP_CONTROL_BLOCK
{
    UCHAR Type;
    CHAR Flags;
    USHORT Size;
    CHAR NumberProcessors;
    CHAR Reserved;
    USHORT ProcessorArchitecture;
    PDUMP_STACK_CONTEXT DumpStack;
    PPHYSICAL_MEMORY_DESCRIPTOR MemoryDescriptor;
    ULONG MemoryDescriptorLength;
    PLARGE_INTEGER FileDescriptorArray;
    ULONG FileDescriptorSize;
    …
}DUMP_CONTROL_BLOCK, *PDUMP_CONTROL_BLOCK;
```

Pre-Initialize Crash Dump Stack

Identify and Initialize Crash Dump Drivers

- `IoInitializeCrashDump()`:
  - `IoGetDumpStack()` → `IopGetDumpStack()`:
    - Fills `IopDumpControlBlock.DumpInit`:
      - Boot device type, geometry and hardware attributes
    - Locates all crash dump drivers: port, miniport and crash dump filter drivers
      - Copies them into memory with "dump_" prefix
        - No DRIVER_OBJECT or DEVICE_OBJECT!
      - Port driver will be one of `dump_scsiport`, `dump_ataport` or `dump_storport`; on disk either `diskdump.sys` (scsi/storport) or `dumpata.sys` (ataport)
      - Miniport can be named anything
    - Fills `IopDumpControlBlock.DumpStack` with linked list of copied drivers

Pre-Initialize Crash Dump Stack

IopDumpControlBlock.DumpStack


```
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;
```

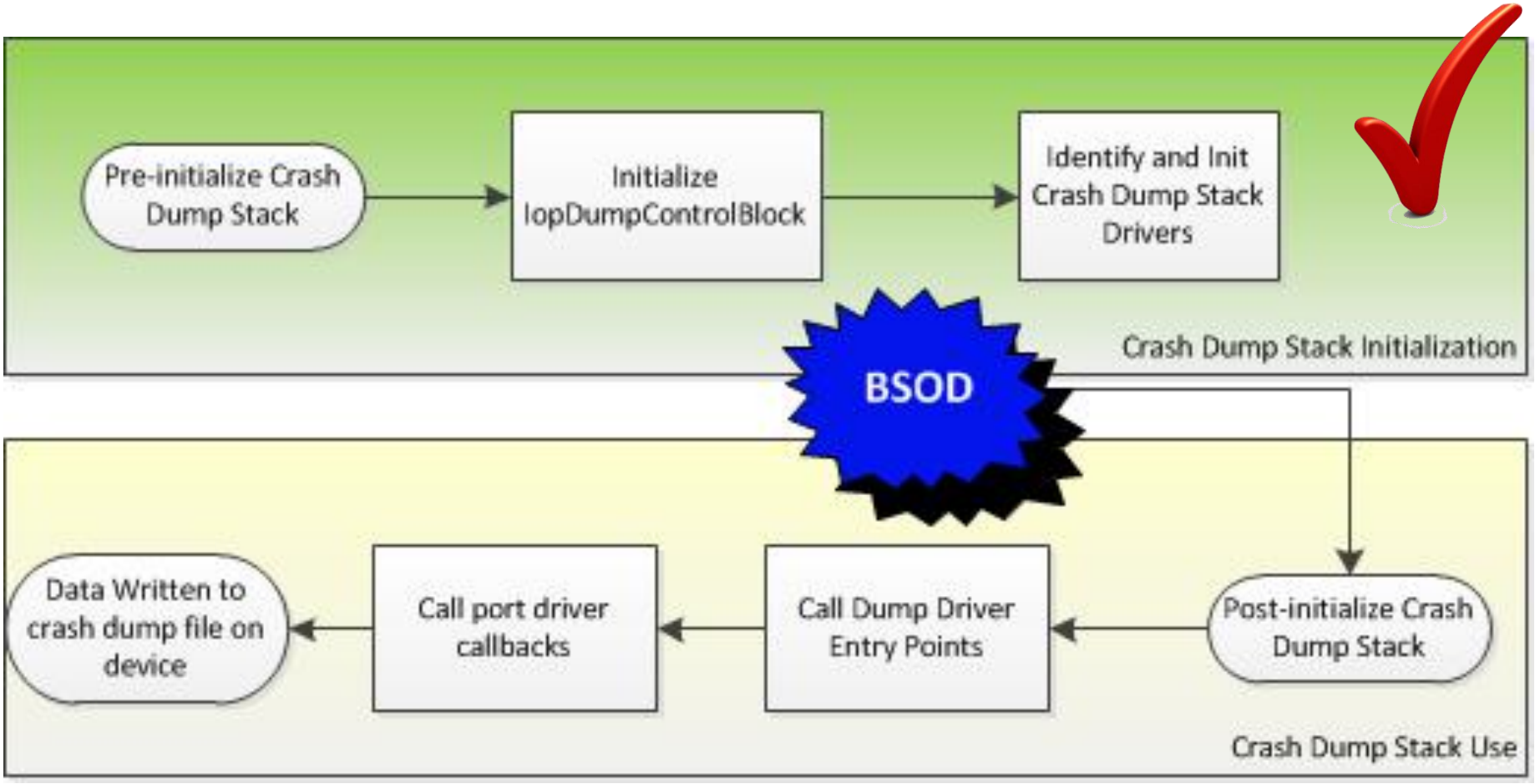
Pre-Initialize Crash Dump Stack

IopDumpControlBlock.DumpStack

- `DumpPointers` - contains hardware-specific information about the disk drive (via `IOCTL_SCSI_GET_DUMP_POINTERS`) which is used during write I/O operations to the crash dump file.
- `DriverList` - contains a linked list of data structures that describe the driver image of each driver in the crash dump stack; used at actual crash dump time to initialize each driver
- `Init` - of type `DUMP_INITIALIZATION_CONTEXT` (undocumented but exported), shown below, is only partially filled in during the 1st phase of initialization.

Source: [3]

Initialization and Use



Post-Initialize Crash Dump Stack

Call Dump Driver Entry Points

- During pre-initialization, drivers were only mapped into memory - no management blocks created, no entry points called
- Each driver in the dump stack will now have its entry point called
- The first driver in the stack is always the dump port and it is always called first
- Two arguments:
  1. `NULL`
  2. `IopDumpControlBlock.DumpInit`

Post-Initialize Crash Dump Stack

Call Dump Driver Entry Points


```
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;
    …
} DUMP_INITIALIZATION_CONTEXT,*PDUMP_INITIALIZATION_CONTEXT;
```

Post-Initialize Crash Dump Stack

Call Dump Driver Entry Points

MANDIANT

- `OpenRoutine`, `WriteRoutine` and `FinishRoutine` fields are populated:
  - 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
- The dump miniport driver's `DriverEntry` is called
  - registers with the dump port driver
- All other dump driver's `DriverEntry` are called with NULL arguments
  - According to MSDN, this notifies them to operate in "crash dump mode"

Post-Initialize Crash Dump Stack

Call Dump Port Driver Callbacks

MANDIANT

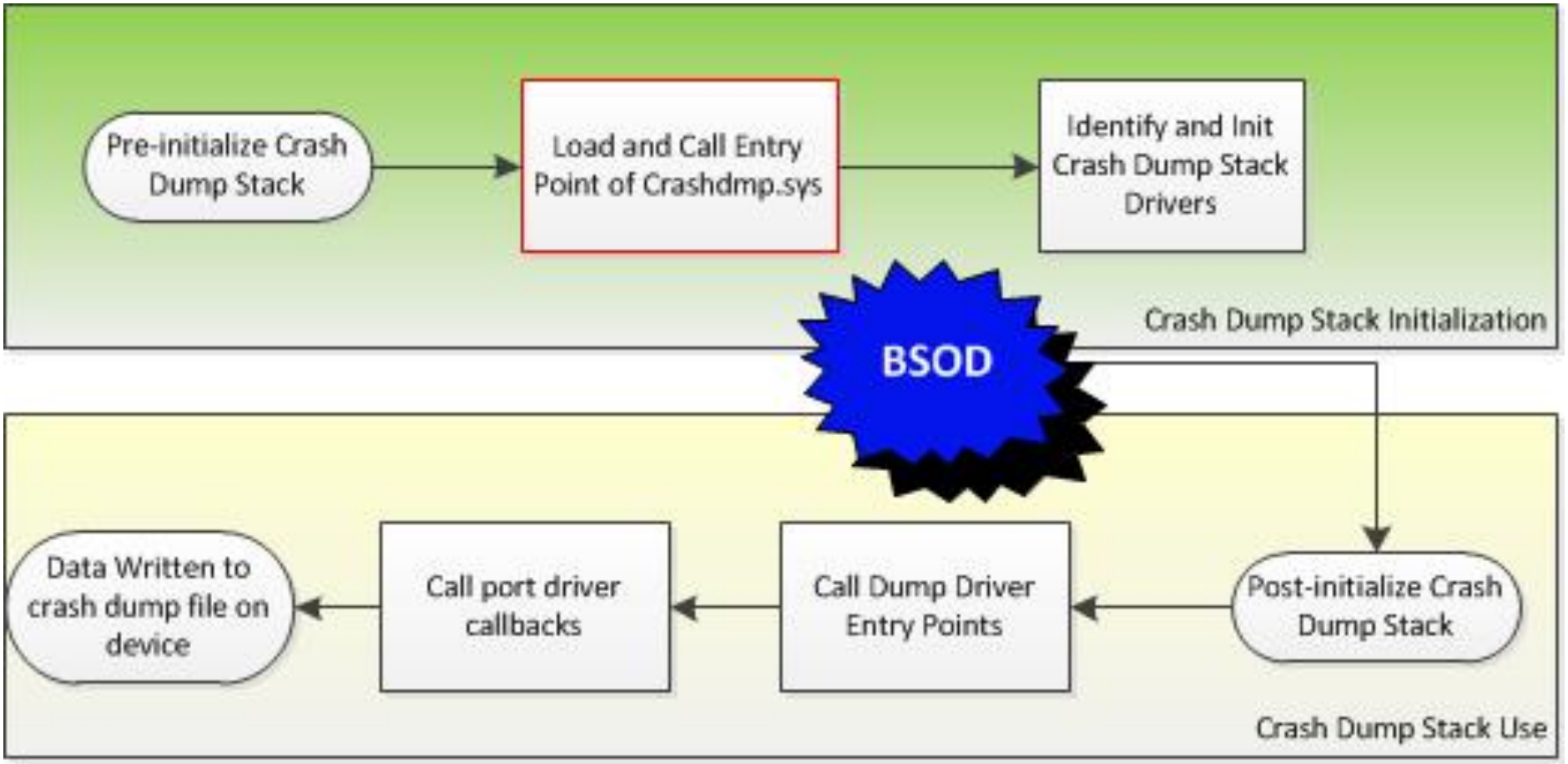
- `KeBugCheck2()` → `IoWriteCrashDump()`:
  - `IoInitializeDumpStack()` – performs post-initialization of crash dump drivers
  - **`DiskDumpOpen()`** – this port driver export is called to prepare the crash dump file
  - Displays the dump string “Beginning dump of physical memory”, stored in the `DUMP_CONTROL_BLOCK` 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()`, passing the dump header

Data Written to Crash Dump File

- `KeBugCheck2()` → `IoWriteCrashDump()`:
  - `IoInitializeDumpStack()` – performs post-initialization of crash dump drivers
  - One of the following functions calls **`DiskDumpWrite()`** until all crash dump data is written:
    - `IopWriteSummaryHeader()`
    - `IopWriteSummaryDump()`
    - `IopWriteTriageDump()`
  - Invokes all `BugCheckSecondaryDumpDataCallback` callbacks to allow drivers to append data to the completed crash dump file
  - Calls **`DiskDumpFinish()`** to close crash dump file
  - Invokes all `BugCheckDumpIoCallback` callbacks, informing them crash dump is complete.

Crash Dump Stack Initialization and Usage, Vista/Windows 7

Initialization and Use



Pre-Initialize Crash Dump Stack

Load and Call Entry Point of Crashdmp.sys

MANDIANT

- Nearly all of crash dump code was removed from kernel and put in `crashdmp.sys`
- `KiInitializeKernel()` → `IoInitSystem()` OR `NtCreatePagingFile()`
  - `IoInitializeCrashDump()`
    - `IopLoadCrashDumpDriver()` – loads `crashdmp.sys`
      - `Crashdmp!DriverEntry()` – fills crash dump call table
- Entry point called with two arguments:
  - Name of the arc boot device
  - Pointer to a global crashdmp callback table

Pre-Initialize Crash Dump Stack

Crashdmp.sys call table

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

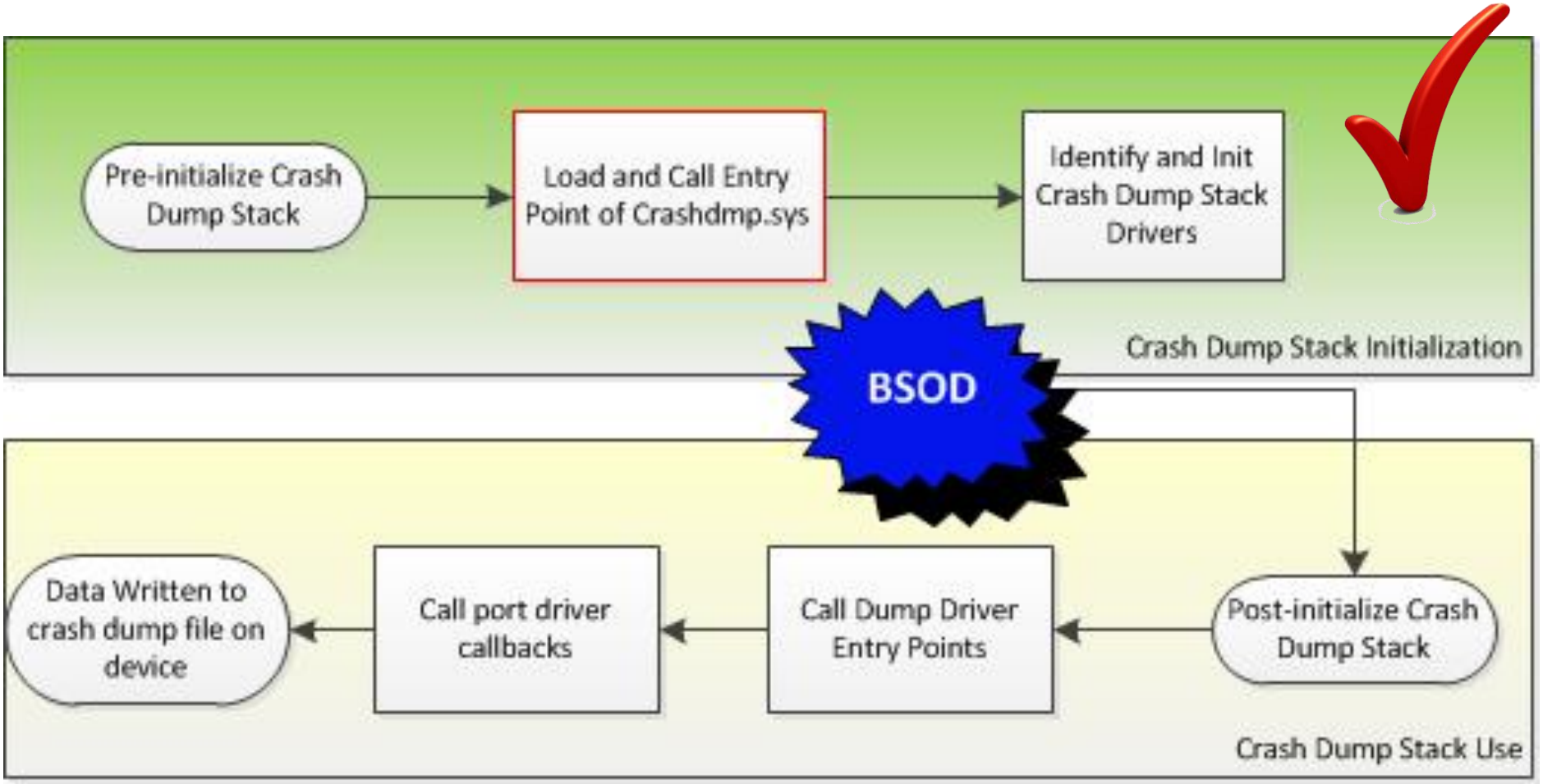
Pre-Initialize Crash Dump Stack

Identify and Initialize Crash Dump Drivers

- KiInitializeKernel() → IoInitSystem()
  OR NtCreatePagingFile()
  - IoInitializeCrashDump()
    - Crashdmp!CrashDmpInitialize():
      - Crashdmp!CrashdmpLoadDumpStack():
        - Crashdump!QueryPortDriver()
        - Crashdmp!LoadPortDriver()
        - Crashdmp!LoadFilterDrivers()
        - Crashdmp!InitializeFilterDrivers()

Source: [3]

Initialization and Use



Post-Initialize Crash Dump Stack

Call Dump Driver Entry Points

- `KeBugCheck2()` → `IoWriteCrashDump()`:
  - Calls the eighth entry in the `CrashDmpCallTable` table, `CrashDmpNotify()` - displays the string “collecting data for crash dump”
  - Fills dump block with bug check codes and other debug information
  - Appends a triage dump if necessary

Post-Initialize Crash Dump Stack

Call Dump Driver Entry Points/Port Driver Callbacks

MANDIANT

- `KeBugCheck2()` → `IoWriteCrashDump()`:
  - Calls the ninth entry in the `CrashDmpCallTable` table, `CrashDmpWrite()`:
    - `CrashdmpInitDumpStack()`:
      - `StartFilterDrivers()` – calls the `DumpStart` callback of each crash dump filter driver
      - `InitializeDumpDriver()` – calls the dump driver entry point; calls the **`DiskDumpOpen`** callback provided by the dump port driver.

Data Written to Crash Dump File

- `KeBugCheck2()` → `IoWriteCrashDump()`:
  - `CrashDmpWrite()`:
    - `CrashdmpInitDumpStack()`:
      - `DumpWrite()` – creates dump file based on configuration:
        - `FillDumpHeader()`
        - Calls one of:
          - `WriteFullDump()`
          - `WriteKernelDump()`
          - `WriteMiniDump()`
      - `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.

How to Use the Crash Dump Stack Outside of the Operating System
or
“How to Bypass the Normal I/O Path”

Inspiration

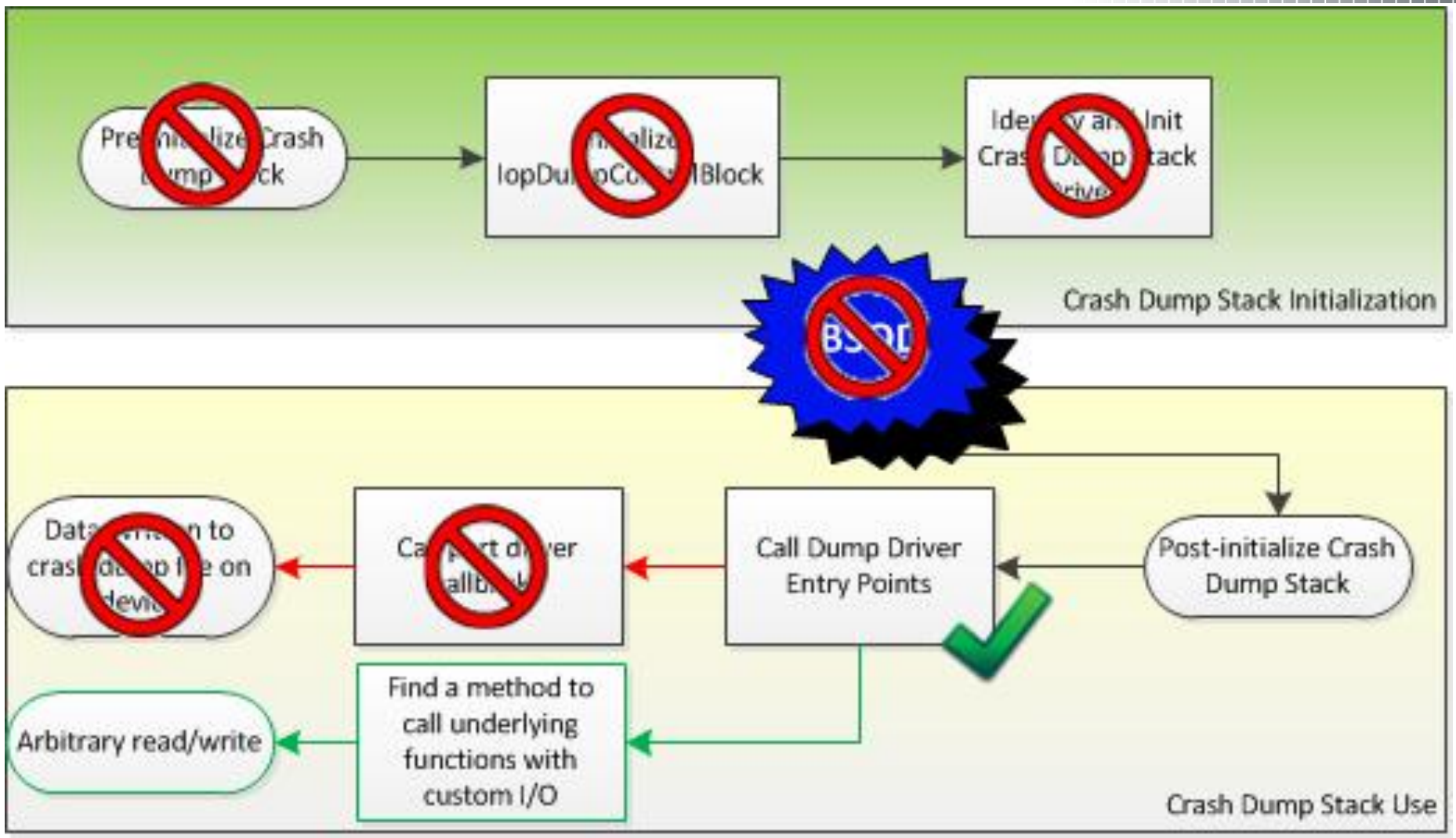
MANDIANT

“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.” [1]

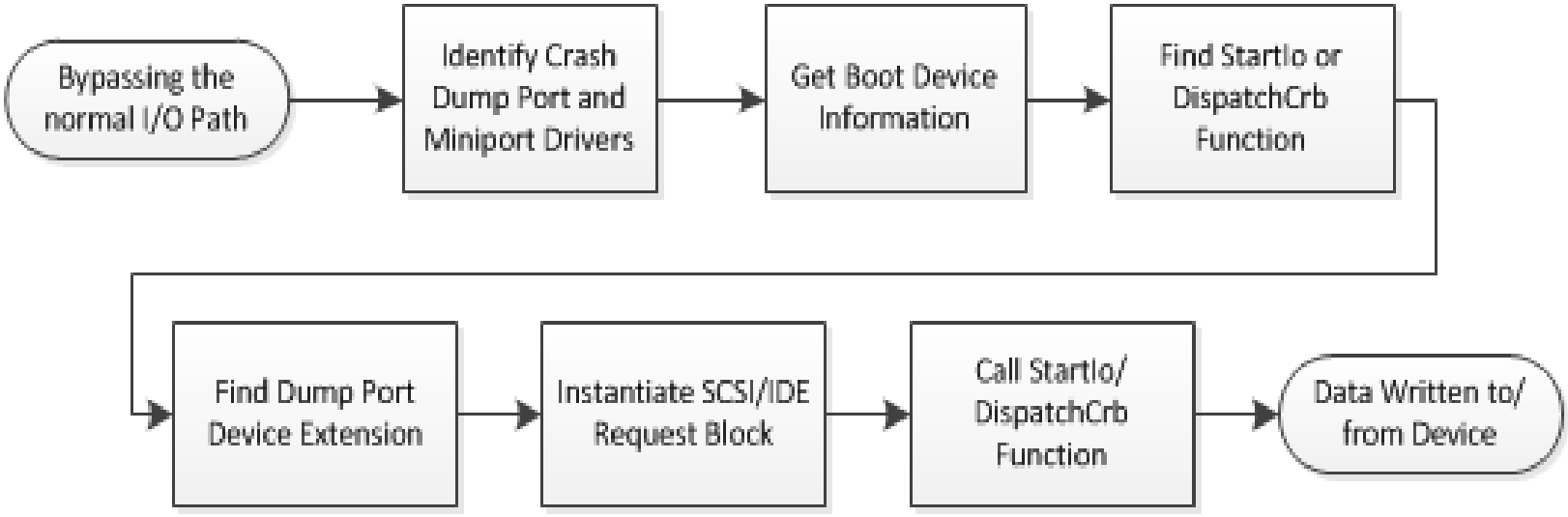
Leveraging the Crash Dump I/O Path

- The crash dump mechanism provides a pristine path to disk
  - But it only provides write capabilities
- Leverage knowledge of the crash dump mechanism and internals of the port/miniport relationship to coerce read/write
- Here's how…

Sort of like the OS does…



…But more like this…



Identify Crash Dump Port and Miniport Drivers

- Walk loaded module list
- Single out dump drivers easily via “dump_” prefix
- Port driver will be one of `dump_scsiport`, `dump_storport` or `dump_ataport`
- Miniport driver name:
  - open a handle to the class driver’s device object, walk attached devices to the lowest one
  - Vista+ `DUMP_POINTERS_EX.DriverList`
- Once drivers have been found, call their entry points with the appropriate arguments

Get Boot Device Information

- IOCTL to get hardware register mapping and port configuration information (`IOCTL_SCSI_GET_DUMP_POINTERS`); data returned in `DUMP_POINTERS` or `DUMP_POINTERS_EX` structure
- IOCTL to get boot device location such as Target id, path Id, and Lun (`IOCTL_GET_SCSI_ADDRESS`); data returned in a `SCSI_ADDRESS` structure
- Resulting information is stored in the `DUMP_INITIALIZATION_CONTEXT` structure before calling the dump port driver's `DriverEntry`

Find StartIo or DispatchCrb

- How do we send I/O requests?
  - There is no device object for dump port driver
- Use internal functions
  - `StartIo` (SCSI) – accepts a single SCSI_REQUEST_BLOCK (SRB) as argument
  - `DispatchCrb` (IDE) – accepts a single argument, a channel extension structure
- Find functions by scanning dump port driver's image's text section for "magic bytes"

Find the Dump Port Driver's Device Extension

- Can't simply call the internal functions, more initialization required
- Port/miniport share a device extension structure that must be properly initialized
  - an internal variable of the port driver
  - Most critical fields are filled in by dump port driver/miniport driver's DriverEntry routines

Find the Dump Port Driver's Device Extension (cont'd)

- Can be found via info leak – pointer to device extension survives function call
  - Example: `Diskdump.sys` leaks in `ecx` register in `DiskDumpOpen()`

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

Send SRB (SCSI)

- Mimic `DiskDumpWrite()`
  - Allocate an MDL at offset `0xD0` (`0x118` x64) into the device extension structure – MDL describes the `SRB.DataBuffer`
  - Call `StartIo()`

```
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;
```

Send SRB (SCSI) (cont'd)

- After MDL is created, send an SRB as follows:
  - `SRB.Function` - `SRB_FUNCTION_EXECUTE_SCSI`
  - `SRB.PathId, SRB.TargetId, SRB.Lun` – set to corresponding fields in `SCSI_ADDRESS`
  - `SRB.CdbLength` - 10 for 10-byte SCSI-2 command
  - `SRB.SrbFlags` - specify flags for a read operation
  - `SRB.DataTransferLength` - 512
  - `SRB.DataBuffer` - allocate 512 bytes NonPagedPool – result stored here
  - `SRB.Cdb` – the SCSI-2 command descriptor block (cdb)

Send IRB (IDE) – The general idea

```
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;
}
```

Send IRB (IDE) – Method 1

- Relies on calling port driver internal functions
- Mimic `IdeDumpWritePending()`
  - Allocate a CRB in `DUMP_INITIALIZATION_CONTEXT.MemoryBlock` at the correct offset (`0x120` for x86, `0x1C0` for x64)
  - Allocate and fill in an IRB at offset `0x288` (`0x3E8` for x64) in the CRB
  - 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 an MDL at offset `0x50` (`0x88` for x64) in the CRB
  - Send the CRB to `DispatchCrb()`
  - Wait via `IdeDumpWaitOnRequest()` function

Send IRB (IDE) – Method 2

- Completely bypass dump port driver
- Mimic `IdeDumpWritePending()`
  - 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
  - Replace `DispatchCrb()` with
    - Call the miniport's `HwStartIo` routine, which is stored at offset `0x2E` in the device extension, passing the device extension and the IRB
  - Replace `IdeDumpWaitOnRequest()` with:
    - 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

Send IRB (IDE) (cont'd)

- `IRB.Function` - `IRB_FUNCTION_ATA_COMMAND`
- `IRB.Channel` - 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 CRB
- `IRB.TargetId` - 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 CRB
- `IRB.Lun` - 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 CRB

Defeating TDL4

Recent MBR Rootkits

- Alureon/TDL4 has gained popularity in the last few years
  - Abuses driver trust chain by hooking the port and miniport drivers, which are at the bottom of the disk driver stack trust chain
  - Modifies I/O requests in various ways to hide its rootkit file system, as well as return a clean MBR
- Similar MBR/VBR rootkits include Popureb, Stoned, Hasta La Vista, Zeroaccess
  - Completely new strains, as well as variants, emerging constantly

TDL4 Infection

- Modifies the miniport's device object and driver object
  - `DRIVER_OBJECT.DriverStartIo` → hooked
  - `DEVICE_OBJECT.DriverObject` → hooked
  - Lowest attached device is unlinked from the miniport device by hooking `DEVICE_OBJECT.NextDevice`
- Monitors for read or write attempts to the boot sector and its hidden file system
  - If boot sector, return original, clean MBR
  - If hidden file system, return zeroes
- For more info, see [2]

Defeating TDL4 (DEMO)

Results – Full Disclosure!

- SCSI – working from Windows XP – 7 (tested)
  - Can cause momentary (10-30s) explorer.exe unresponsiveness (unclear if TDL influence)
    - Pre-mature IRQL lowering? Need to wait on request to complete or risk race/contention with normal I/O path
- IDE – polling is hard ☹
  - Method 1: IRB sent, garbage returned ☹
    - Either the port driver is messing up the IRB somewhere during polling or we are missing a field in dump extension
  - Method 2: IRB sent, nothing returned ☹, Irb status 2 (data length mismatch), ATA status 0x20 (?)
    - IRB.TaskFile might have invalid values
      - IDE_TASK_FILE: No examples anywhere!
    - Dump extension field missing – data length?

Takeaways

- Why is this important?
  - We have a separate path to disk which is not currently hooked by any malware I am aware of
    - Tampering with the crash dump mechanism could destabilize the system
  - We can read AND write to disk this way
    - The crash dump port driver only provides callbacks to write to disk, but the StartIo function lets us issue any arbitrary SRB.
    - The ability to write to disk with this technique provides us unique remediation (file cleaning) opportunities
- Caveats:
  - IDE not fully tested
  - Experimental
- Tons of details left out – read the whitepaper!

References and Sources

[1] http://msdn.microsoft.com/en-us/library/ff564084(v=VS.85).aspx

[2] http://go.eset.com/us/resources/white-papers/The_Evolution_of_TDL.pdf

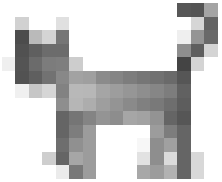
[3] http://bsalert.com/f-store/bsod.jpg

Oops – did we finish early?

MANDIANT

Select your topic:

pdbxtract – pdb type information tool



Arcane secrets of Skype permacatting

THANKS FOR LISTENING!!

Questions???

Aaron.LeMasters@Mandiant.com
@lilhoser