CROWDSTRIKE

ABUSING THE KERNEL SHIM ENGINE

ALEX IONESCU, CHIEF ARCHITECT

@AIONESCU

1. INTRO
2. HOW DOES IT SHIM?
3. WRITING A SHIM DRIVER
4. SHIM CASE STUDIES
5. SHIM ENGINE INTROSPECTION
6. DRIVERMON DEMO
7. CONCLUSION AND Q & A

CROWDSTRIKE

INTRO

BIO

- Chief Architect at CrowdStrike
  - Previously worked at Apple on iOS Core Platform Team
- Co-author of Windows Internals 5th and 6th Editions
- Reverse engineering NT since 2000 – main kernel developer of ReactOS
- Instructor of worldwide Windows Internals classes
- Conference speaking:
  - Infiltrate 2015
  - Blackhat 2016, 2015, 2013, 2008
  - SyScan 2015-2012, NoSuchCon 2014-2013, Breakpoint 2012
  - Recon 2016-2010, 2006
- For more info, see www.alex-ionescu.com

KUDOS

- Robin Keir – CrowdStrike Senior Research Engineer
  - AKA “Tool Extraordinaire”
- Responsible for CrowdInspect, CrowdResponse and the CS Heartbleed Scanner
  - And lots of internal tools..
- Took command-line version of DriverMon and transformed in into GUI Application
  - Filtering, import/export, etc…
  - Will be released as a new community tool (more on this later)
- More info at http://www.keir.net

WHAT THIS TALK IS ABOUT

- Moar Windows Internals!
  - Introducing the **K**ernel **S**him **E**ngine
  - *Kse* functions in the kernel, and related structures
- Somewhat of a kernel-mode follow up to last year's talk on Binary Instrumentation & Hooking Technologies
- Ability to hook any Windows driver using a legitimate (but undocumented) interface
  - Including IAT hooking, IRP Callbacks and Driver Callbacks
  - Easy-to-use and program against, given the right structures
- How to forensically check for registered shim providers
  - And potentially hooked drivers
- Hooking Shimmed Drivers by abusing the KSE

WHAT THIS TALK IS ALSO ABOUT

- Looking at the built-in and inbox shim providers
- Will also dump the Driver Shim Database that ships with Windows
  - Blocked Drivers
  - Device Flags
  - Associated Driver Shims
- Writing a Generic Shim Driver (“Shimmer”) that shims any driver and can send to user-mode notifications for each API/callback
  - Demo of both a command-line tool to dump this data
- Demo and upcoming release of DriverMon tool
  - Ability to filter, parse, and analyze the Shimmer output
  - ProcMon for drivers! ☺

CROWDSTRIKE

HOW DOES IT SHIM?

Kernel Shim Engine Internals

OVERVIEW OF KERNEL SHIMMING

- The KSE provides two primary capabilities to the Windows operating system
  - Device Shims
  - Driver Shims
- Devices (usually at the bus driver layer) can be shimmed, meaning that specific driver-internal flags can be set for various devices, which cause these drivers to apply hardware-specific “workarounds”, or shims
  - Added in Windows XP
  - Functionality similar to the Errata Manager (EM) functionality added in Windows Vista
  - Also similar to “PciHackFlags” and other per-driver registry flags since Windows 2000
- Drivers (identified by a file name) can be shimmed, meaning that specific functions that a driver imports, callbacks that a driver receives, and IRPs that a driver handles, can be hooked by custom routines which might change behavior

KERNEL SHIM ENGINE INITIALIZATION

- Occurs in two phases during pre-driver I/O Manager Initialization
  - Phase 0 done before HAL PnP Driver Initialization
  - Phase 1 done after WMI/ETW Initialization
- Managed by *KseInitialize*
- KSE State is stored in a global KseEngine data structure (see upcoming slide)
- *KseShimDatabaseBootInitialize()* is used to parse the SDB at boot in Phase 0
- KSE initialization is skipped in the following conditions:
  - Safe Boot Mode
  - Driver Verifier Enabled
  - No database present/loaded by loader
  - Win PE Mode goes through initialization but later disables driver shims

SHIM DATABASE LOAD

- KSE uses the exact same Shim DataBase (.SDB) format as in user-mode
  - First appeared in Windows XP as DrvMain.sdb (“Device Flag” implementation only)
- Loaded from disk by OS Loader (Winload.exe) with *OslpLoadMiscModules*
  - Loads driver database from \SystemRoot\AppPatch\drvmain.sdb
  - Loads errata manager .INF file as well (We’ll leave that for another talk…)
- Stored in LOADER_PARAMETER_BLOCK_EXTENSION
  - DrvDBImage
  - DrvDBSize
- After load, *KsepMatchInitMachineInfo()* parses ACPI, BIOS and CPU information used for later matching
  - Used whenever MATCHING_DEVICE is present for a shim entry (vs. MATCHING_FILE)

DEVICE SHIMS

- Kernel provides three exports for drivers:
  - *KseQueryDeviceData()*
  - *KseQueryDeviceDataList()*
  - *KseQueryDeviceFlags()*
- Device Flags are really a subset of Device Data
  - Specifically, Data Type 11
- Device Data can actually be queried from user-mode with *NtQuerySystemInformation()*
  - SystemDeviceDataInformation = 0x88
  - SystemDeviceDataEnumerationInformation = 0x89

REGISTRY DEVICE DATA

- Device data can come from the SDB or from the Registry
  - Data Type **0x20000000** indicates non-registry data
- *KsepDbQueryRegistryDeviceData()* queries \\Registry\\Machine\\System\\CurrentControlSet\\Control\\Compatibility\\Device key
  - One key for each Hardware ID String, replacing “\” with “!” as to not confuse the registry code
- Then reads whatever value is associated with the Data Name
  - Returns raw registry type, ORed with **0x10000000** as the Data Type
- Otherwise, a cache lookup is performed with *KsepDbCacheQueryDeviceData()*
  - If we have a miss, *KsepDbCacheReadDevice* is used to read from the SDB
  - Then *KsepDbCacheInsertDevice* is used to populate it

SDB DEVICE DATA

- *KsepDbCacheReadDevice()* will enumerate the SDB for two types of entries
- **0x7013** known as FLAG
  - Then calls *KSepDbReadKFlag()*
- **0x7028** known as KDATA
  - Then calls *KsepDbReadKData()*
  - Does not appear to be any built-in KDATA in the Windows 10 DrvMain SDB
- Interestingly, **0x7024** known as KFLAG does not appear to be parsed by the code
- Things that read Device Data / Device Flags:
  - **BTHENUM.SYS, BTHPORT.SYS, HDAUDBUS.SYS, HIDBTH.SYS, HIDCLASS.SYS, NDIS.SYS, UFXSYNOPSYS.SYS, URSCX01000.SYS, USBHUB.SYS**

HARDWARE ID CACHE

- Each time a new PnP device is enumerated, *PiProcessNewDeviceNode()* calls *KseAddHardwareId(),* which adds an entry into the Device Hardware ID cache
- This cache can be queried by *KseLookupHardwareId()* which is called by *AhcCacheQueryHwId()*
- *AhcCacheQueryHwId()* is called from *NtApphelpCacheControl* if the service class is **ApphelpCacheServiceHwIdQuery**
  - As of Windows 10, all other queries go to Ahcache.sys
- Since this is callable from user-mode, would indicate that the user-mode app compat service uses this for verifying if certain hardware IDs are present on the machine

BUILT-IN DRIVER SHIM INITIALIZATION

- At the end of Phase 0, the **DriverScope** Shim is registered
  - Hooks all driver callbacks + IOCTL, CREATE/CLOSE, POWER, PNP IRPs
  - Hooks *IoCreateDevice, PoRequestPowerIrp, ExAllocatePoolWithTag, ExFreePoolWithTag, ExAllocatePool, ExFreePool*
  - Prints out over ETW any time any these operations happen, if the shim is active
- During Phase 1, *KseVersionLieInitialize()* initializes the **Win7VersionLie, Win8VersionLie** and **Win81VersionLie** shims
  - All of these hook *RtlGetVersion()* and *PsGetVersion()*
  - Obviously they return fake version data depending on the shim
- Finally, the **SkipDriverUnload** shim is registered
  - Hooks the DriverUnload driver callback
  - Prints out to ETW – and never calls the original routine

SHIMMING LOADED DRIVERS

- *KseDriverLoadImage*() takes care of shimming any new drivers
  - Called by *IopInitializeBuiltinDriver()* for boot drivers
  - Called by *MiCompactServiceTable*() for Win32k.sys
  - Called by *MiDriverLoadedSucceeded*() for all other drivers
- Does not support session loaded drivers
- Two phases:
  - *KsepGetShimsForDriver()*
  - *KsepApplyShimsToDriver()*
- Successful shimming of a driver results in:
  - ETW / Event Log Trace
  - KseEngine.ShimmedDriverHint pointing to base address of driver
  - RegistryFlags OR'ed with 0x800 (ShimsActive)

SHIMMED DRIVER MODULES

- Each shimmed module is added into the shimmed driver list
  - *KsepIsModuleShimmed* will verify this
  - Contains load address of driver, number of shims applied, and array of applied shims
- First, KSE searches the registry for any applicable shims
  - *KsepEngineGetShimsFromRegistry()* and *KsepRegistryQueryDriverShims()*
- Next, the string used as the shim name in the registry, is looked up in the SDB
  - Must match a KSHIM in the database (*KsepDbGetShimInfo()*)
- If no registry match found, *KsepDbGetDriverShims()* is used
  - Tries to find a KSHIM_REF for the KDRIVER entry
- Each applied shim is described by its GUID, Name, Command Line and “Source”
  - 0 = Registry, 1 = SDB

RESOLVING DRIVER SHIMS

- After looking up the shims, it's time for *KsepResolveApplicableShimsForDriver()* to check for registered shims that match its needs
  - *KsepIsShimRegistered()* does a first pass on the registered shim list
- If a registered shim was not found, we check if the Flags in the KSHIM descriptor indicate this is a **delay-load shim provider** (0x02)
  - If so, *KsepLoadShimProvider()* is called, which then calls *ZwLoadDriver()*
  - Then the registered shim list is parsed again – the shim should now exist
- Then, loaded modules are dumped with *KsepGetLoadedModuleList()*
  - Each loaded module described by an "export driver" function hook is parsed, and *RtlFindExportedRoutineByName*() is used to make sure the hooked function actually exists
- The driver is now ready to be shimmed

APPLYING DRIVER SHIMS

- *KsepApplyShimsToDriver()* will now process the targeted module and apply the relevant resolved shims
  - First, *KsepPatchDriverImportsTable()* will patch the IAT by using *KsepPatchImportTableEntry()* which then relies on *MmReplaceImportEntry()*
- Then, a callout notification is made to the shim provider (more on this later)
- However, not all possible shims have been applied – only function hooks
- Another step happens inside of *IopLoadDriver()* and *IopInitializeBuiltinDriver()* by calling *KseShimDriverIoCallbacks()*
  - Right after the DriverInit routine is called
  - This takes care of all the driver callback and IRP handlers using *KsepGetShimCallbacksForDriver()*

CROWDSTRIKE

WRITING A SHIM DRIVER

How Shimmer Came To Be

KSE_SHIM

- Each unique shim (a particular compatibility fix) that is registered is identified by a structure we'll call KSE_SHIM

- ```
  typedef struct _KSE_SHIM
  {
      _In_ ULONG Size;
      _In_ PGUID ShimGuid;
      _In_ PWCHAR ShimName;
      _Out_ PVOID KseCallbackRoutines;
      _Inopt_ PVOID ShimmedDriverTargetedNotification;
      _Inopt_ PVOID ShimmedDriverUntargetedNotification;
      _In_ PVOID HookCollectionsArray;
  } KSE_SHIM, *PKSE_SHIM;
  ```

- The hooks required for the shim to function are then described by the HookCollections array…


  ```

KSE_HOOK_COLLECTION

- Each type of hooked entity needs to be described by a KSE_HOOK_COLLECTION, and multiple collections can exist for a particular shim – last one must be Invalid
- Basically two types of things can be hooked:
  - Exported Functions
  - Driver Callbacks
- ```
  typedef struct _KSE_HOOK_COLLECTION
  {
      ULONG Type; // 0: NT Export, 1: HAL Export, 2: Driver Export, 3: Callback
      PWCHAR ExportDriverName; // If Type == 2
      PVOID HookArray;
  } KSE_HOOK_COLLECTION, *PKSE_HOOK_COLLECTION;
  ```
- The actual hooks for each shim collection are then described by the Hooks array…


  ```

KSE_HOOK

- Each individual hook is then finally described by a KSE_HOOK structure
- Multiple hooks can exist for a given collection – last entry must be Invalid
- ```
  typedef struct _KSE_HOOK
  {
      _In_ ULONG Type;                    // 1: Function, 2: IRP Callback
      union
      {
          _In_ PCHAR FunctionName;        // If Type == 1
          _In_ ULONG CallbackId;          // If Type == 2
      };
      _In_ PVOID HookFunction;
      _Outopt_ PVOID OriginalFunction; // If Type == 1
  } KSE_HOOK, *PKSE_HOOK;
  ```



  ```

HOOKING FUNCTIONS

- Hooking functions is extremely easy, but different collections must be used:
  - For all exports from “Ntoskrnl.exe”, use a collection of type 0
  - For all exports from “Hal.dll”, use a collection of type 1
- For all custom driver exports, use a collection of type 2 – for each such driver
  - Write down the driver name in each collection
- The hooks then simply look like this:
  - {0, “ExAllocatePoolWithTag”, (PVOID)(ULONG_PTR)ShimExAllocatePoolWithTag, NULL}
- In each hook function, the original function should eventually be called:
  - Return ((PCAST_TO_FUNCTION)(ULONG_PTR)Hooks[0].OriginalFunction(args);
- Use _ReturnAddress() intrinsic to figure out the shimmed driver, if needed

HOOKING CALLBACKS

- The I/O manager implements a variety of callbacks that driver developers can choose to implement
  - DRIVER_OBJECT's DriverInit, DriverUnload, DriverStartIo
  - DRIVER_EXTENSION's AddDevice
- Then, each I/O operation is associated with an I/O Request Packet, or IRP
  - DRIVER_OBJECT has a DriverDispatch array of up to IRP_MJ_MAXIMUM_FUNCTION pointers for each defined IRP type
- The shim engine allows hooking any and all of these routines
  - Doesn't actually work for DriverInit – the shim engine is not called yet ☺

```
Status = (DriverObject->DriverInit)(DriverObject, RegistryPath);
if ( Status >= 0 )
{
    VfXdvDriverCaptureIoCallbacks(DriverObject, RegistryPath, BaseName);
    KseShimDriverIoCallbacks(DriverObject, RegistryPath, &Destination);
}
```


RETURNING FROM CALLBACKS

- When hooking a callback, KSE defines a callback ID for each type of callback:
  - 1 for DriverInit
  - 2 for DriverStartIo
  - 3 for DriverUnload
  - 4 for DriverAddDevice
- For IRPs, add 100 to the IRP_MJ_XXX definition
  - For example, IRP_MJ_CLOSE becomes callback 102
- The KSE_HOOK does not contain the OriginalFunction pointer filled out
  - Instead, we must call *KseGetIoCallbacks* to recover the original handler
  - Our KSE_SHIM structure contains a pointer to the two KSE callbacks in KSE_ENGINE

OBTAINING IRP COMPLETION INFORMATION

- IRPs very often require output information (some are unidirectional/input only)
  - For example, for an IRP_MJ_WRITE: how many bytes were actually written?
  - For IRP_MJ_READ, what was actually read
  - …etc…
- We cannot call the original IRP handler, and then read the result from the IRP field
  - Because the original IRP handler may have already “completed” the IRP – which could cause it to be freed/reused/stale
  - In other cases, the original IRP handler may have forwarded the IRP – which means the result is not yet known
- KSE allows us to register a “completion callback” for each IRP
  - Using *KseSetCompletionHook*
  - Again, our KSE_SHIM will receive a pointer to this callback, just like on the earlier slide

SHIM TARGETING CALLBACKS

- When a driver is loaded which matches the Shim Database & Registry rules, a notification is sent to the shim driver's (if one was registered)
  - The driver's name, load address, size, timestamp, and checksum are sent
  - This can be useful to then maintain state around the memory address range and associated driver
- Similarly, when a driver is later unloaded, its base address is sent to the shim driver's un-targeting notification (if one was registered)
- Because hooked functions do not return information on which hooked driver sent it, keeping driver state is necessary if per-driver shimming is needed
  - For function hooks use _ReturnAddress() and scan within Start->Start+Size per driver
  - For IRP callback hooks use DeviceObject->DriverObject->DriverStart and scan

REGISTERING A SHIM PROVIDER

- To register our shim provider, we have to call *KseRegisterShimEx* for each shim we want to provide
  - Pass in the KSE_SHIM structure that identifies the shim, its collections, and hooks
  - Pass in the DRIVER_OBJECT for the shim driver – this will make KSE take a reference on it
- Older *KseRegisterShim* is not safe because no DRIVER_OBJECT is passed – therefore, the driver might unload with active shims
  - Used by kernel and non-unloadable drivers
- If shim driver unloads/unregisters shims, but shimmed drivers are still active, the driver will remain resident in memory (but not handle new I/O)
- Similarly, if shim provider registers after a driver has loaded, it won't be shimmed

CASE STUDIES

Inbox (but not built-in) Shim Drivers

ANALYZING THE BUILT-IN DATABASE

- Can use the sdb2xml tool or sdb-explorer or shims
  - https://github.com/evil-e/sdb-explorer
  - https://tzworks.net/prototype_page.php?proto_id=33
  - https://blogs.msdn.microsoft.com/heaths/2007/11/03/shim-database-to-xml
- Contains four types of relevant entries:
  - APPHELP entries which will block loading the driver entirely, from user-mode, and display an error message (not checked by kernel)
  - KDEVICE entries which are used by the “Device Compatibility” part of KSE, and have been there since Windows XP
    - Contain FLAG entries which specify a FLAG_MASK_KERNEL and associated NAME
  - KDRIVER entries which are used by the “Driver Compatibility” part of KSE, which are new to Windows 8
  - KSHIM entries which are referenced (using KSHIM_REF) by KDRIVER entries above, and describe the list of valid shims in the database

SOME EXAMPLES…

```
- <KDEVICE>
    <NAME type="xs:string">USB:ROOT_HUB\VID_1B21&PID_1040</NAME>
    <APP_NAME type="xs:string">XHCI Host Controller Root Hub</APP_NAME>
    <VENDOR type="xs:string">ASMedia</VENDOR>
    <EXE_ID type="xs:string" baseType="xs:base64Binary">{74dbcda1-025d-4fa8-a3af-08f26af1219b}</EXE_ID>
  - <FLAG>
      <NAME type="xs:string">USB</NAME>
      <FLAG_MASK_KERNEL type="xs:long">4096</FLAG_MASK_KERNEL>
    </FLAG>
  </KDEVICE>
```

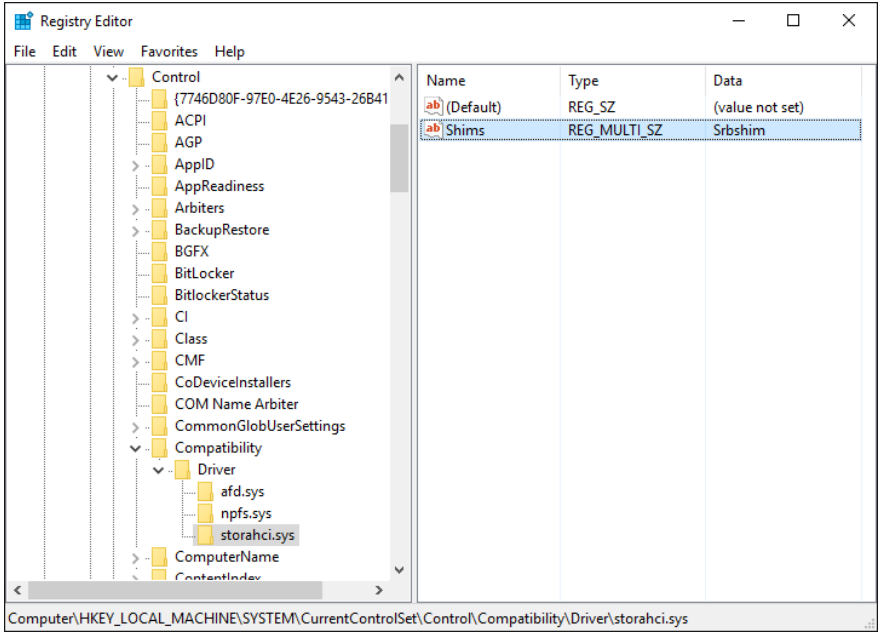
```
<EXE tid="0xbe78" typ="LIST">
 <NAME tid="0xbe7e" typ="STRINGREF">tessafe.sys</NAME>
 <APP_NAME tid="0xbe84" typ="STRINGREF">QQ Game</APP_NAME>
 <VENDOR tid="0xbe8a" typ="STRINGREF">Tencent</VENDOR>
 <EXE_ID tid="0xbe90" typ="BINARY" len="0x10" guid="2DECD936-9AFA-C14D-ABB3BFC5AA43F387" />
 <APP_ID tid="0xbea6" typ="BINARY" len="0x10" guid="B9F675A7-7FA3-484B-8ED34A2D8DFDEE52" />
 <APPHELP tid="0xbebc" typ="LIST">
  <HTMLHELPID tid="0xbec2" typ="DWORD">0x1addc</HTMLHELPID>
  <APP_NAME_RC_ID tid="0xbec8" typ="DWORD">0x0</APP_NAME_RC_ID>
  <VENDOR_NAME_RC_ID tid="0xbece" typ="DWORD">0x0</VENDOR_NAME_RC_ID>
  <SUMMARY_MSG_RC_ID tid="0xbed4" typ="DWORD" resId="10017">A driver is installed that causes stability problems with your system.
 </APPHELP>
 <MATCHING_FILE tid="0xbeda" typ="LIST">
  <NAME tid="0xbee0" typ="STRINGREF">*</NAME>
  <UPTO_BIN_PRODUCT_VERSION tid="0xbee6" typ="QWORD">0x100000002</UPTO_BIN_PRODUCT_VERSION>
 </MATCHING_FILE>
</EXE>
```


DEFINED WINDOWS SHIMS & PROVIDERS

- DriverScope, SkipDriverUnload, KmWin8VersionLie, KmWin81VersionLie, KmWin7VersionLie
  - Built-in
- kmautofail/autofail
  - Registered by kmautofail.sys/autofail.sys (non-existent)
- StorPort, DeviceIdShim, Srbshim
  - Registered by Storport.sys
- Usbshim
  - Registered by Usbd.sys
- NdisGetVersion640Shim
  - Registered by Ndis.sys

DEFINED REGISTRY SHIM TARGETS

- HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Compatibility contains KSE settings
  - Base name of the driver used as subkey
  - A REG_MULTI_SZ value called “Shims” can exist to list all the applicable shims
- By default, Storahci.sys is shimmed with Srbshim



DEFINED SDB SHIM TARGETS

- Usbshim
  - Wsr_rci.sys, Edgeser64.sys, Acfdcp64.sys
- NdisGetVersion640Shim
  - A6100.sys, Rtwlane.sys, Rtwlans.sys, Rtwlanu.sys
- KmWin81VersionLie
  - DefragFS.sys
- Others
  - Lots of drivers…
  - Simply dump the SDB.XML or SDB.TXT and look for KSHIM_REF's matching the driver
- Can be easily updated through Windows Update

INBOX SHIM PROVIDER CASE STUDY

- Scsiport.sys calls *StorpRegisterShim* on load
  - Registers 2 shims on Windows 10 Threshold 2: **SrbShim** and **DeviceIdShim**
- Each one has a single shim cook collection:
  - Type = 3 (Callback Collection)
- Each collection has a single hook:
  - Type = 1 (Callback)
  - CallbackId = IRP_MJ_DEVICE_CONTROL
- Hence, all storage IOCTLs are hooked through StorPort
  - *DeviceIdShimHookDeviceControl* for the **DeviceIdShim**
  - *SrbShimHookDeviceControl* for the **SrbShim**

SRB SHIM BEHAVIOR

- Hooks any IOCTLs of type IOCTL_STORAGE_QUERY_PROPERTY
  - With QueryType == PropertyStandardQuery and PropertyId == StorageAdapterProperty
  - With a Length of >= 32 bytes
- Registers an I/O completion callback for any IRP matching the above
  - *SrbShimStorageAdapterPropertyCompletionHook*
- The completion hook checks if the result of the IRP was STATUS_SUCCESS and the return length was >= 32 bytes as well
  - If so, it clears out the SrbType and AddressType fields of the resulting output structure (STORAGE_ADAPTER_DESCRIPTOR)

```
__int64 __fastcall SrbShimHookDeviceControl(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
  __int64 v2; // r9@0
  _IO_STACK_LOCATION *ioStack; // r8@1
  __int64 majorFunction; // rsi@1
  STORAGE_PROPERTY_QUERY *Query; // rax@2
  __int64 v8; // rax@10
  __int64 v9; // rax@11
  __int64 v10; // rax@11
  __int64 v11; // r8@11
  __int64 v12; // r9@11

  ioStack = Irp->Tail.Overlay.CurrentStackLocation;
  majorFunction = ioStack->MajorFunction;
  if ( ioStack->Parameters.Read.ByteOffset.LowPart == 0x2D1400 )
  {
    Query = Irp->AssociatedIrp.SystemBuffer;
    if ( Query )
    {
      if ( Query->QueryType == PropertyStandardQuery
        && Query->PropertyId == StorageAdapterProperty
        && ioStack->Parameters.Read.Length >= 0x20 )
      {
        if ( WPP_GLOBAL_Control != &WPP_GLOBAL_Control
          && *(WPP_GLOBAL_Control + 11) & 0x10
          && *(WPP_GLOBAL_Control + 41) >= 4u )
        {
          WPP_SF_(
            *(WPP_GLOBAL_Control + 3),
            14i64,
            &WPP_0e2033ffec88805dc5dfbfc3b9546116_Traceguids);
        }
        v8 = SrbShim.Callbacks->KseSetIoCompletionHookRoutine;
        _guard_dispatch_icall_fptr(
          DeviceObject,
          Irp,
          SrbShimStorageAdapterPropertyCompletionHook,
          0i64);
      }
    }
  }
  v9 = SrbShim.Callbacks->KseGetIoCallbacksRoutine;
  v10 = *(_guard_dispatch_icall_fptr(
            DeviceObject->DriverObject,
            Irp,
            ioStack,
            v2)
        + 8 * majorFunction
        + 32);
  return _guard_dispatch_icall_fptr(DeviceObject, Irp, v10, v11);
}
```

SRBSHIM COMPLETION HOOK

```
void __fastcall SrbShimStorageAdapterPropertyCompletionHook(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
    _STORAGE_ADAPTER_DESCRIPTOR_8 *storageAdapterDescriptor; // rax@3

    if ( Irp->IoStatus.Information >= 0x20 && Irp->IoStatus.Status >= 0 )
    {
        storageAdapterDescriptor = Irp->AssociatedIrp.SystemBuffer;
        if ( storageAdapterDescriptor )
        {
            storageAdapterDescriptor->SrbType = 0;
            storageAdapterDescriptor->AddressType = 0;
            if ( WPP_GLOBAL_Control != &WPP_GLOBAL_Control )
            {
                if ( *(WPP_GLOBAL_Control + 11) & 0x10 )
                {
                    if ( *(WPP_GLOBAL_Control + 41) >= 4u )
                        WPP_SF_(
                            *(WPP_GLOBAL_Control + 3),
                            15i64,
                            &WPP_0e2033ffec88805dc5dfbfc3b9546116_Traceguids);
                }
            }
        }
    }
}
```


MSDN TO THE RESCUE

- Interestingly, when looking in IDA (or doing the math), one can see that due to alignments, the size of the structure is actually 32 bytes even on < Windows 8
- The shim is likely attempting to fix app compat issues where older drivers were still sending 32 bytes (counting for alignment), and were not expecting the bottom two fields to be non-zero
  - Perhaps incorrectly reading the BusMinorVersion as a ULONG?

SrbType

Specifies the SCSI request block (SRB) type used by the HBA.

Value	Meaning
SRB_TYPE_SCSI_REQUEST_BLOCK	The HBA uses SCSI request blocks.
SRB_TYPE_STORAGE_REQUEST_BLOCK	The HBA uses extended SCSI request blocks.

This member is valid starting with Windows 8.

AddressType

Specifies the address type of the HBA.

Value	Meaning
STORAGE_ADDRESS_TYPE_BTL8	The HBA uses 8-bit bus, target, and LUN addressing.

This member is valid starting with Windows 8.

CROWDSTRIKE

SHIM ENGINE INTROSPECTION

Debugger and Event Log Analysis

INTROSPECTING THE SHIM ENGINE

- Because there are no symbols, it's incredibly hard to analyze (at runtime), what shims are active/registered, which modules are shimmed, etc…
- With the kernel debugger and the right scripts/knowledge, one can easily get a good view of everything that the Kernel Shim Engine knows about
- In this section, we'll take a look at some of these data structures and show a few WinDBG scripts
  - Shim.wds – Dump current Kernel Shim Engine state
  - Shimmod.wds – Dump an active Shim Module and its active shims
  - Shimreg.wds – Dump an active Registered Shim and its applicable shims
  - Shimcache.wds – Dump information on the Hardware ID and/or Device Data Cache
    - Can also dump the Hardware ID Cache Entries
- Will cleanup & release scripts on GitHub later next week

KSE_ENGINE

- All KSE-related relevant state is stored in a structure we'll call KSE_ENGINE.
  - Allows forensic analysis of current KSE state, and possible activation/deactivation reasons

```
typedef struct _KSE_ENGINE
{
    ULONG DisableFlags; // 0x01 : DisableDriverShims, 0x02: DisableDeviceShims
    ULONG State; // 0 : Not Ready, 1: In Progress, 2: Ready
    ULONG Flags; // 0x02: GroupPolicyOK, 0x800: DrvShimsActive, 0x1000: DevShimsActive
    LIST_ENTRY ProvidersListHead;
    LIST_ENTRY ShimmedDriversListHead;
    PKSE_GET_IO_CALLBACKS KseGetIoCallbacksRoutine;
    PKSE_SET_COMPLETION_HOOK KseSetCompletionHookRoutine;
    PVOID DeviceInfoCache;
    PVOID HardwareIdCache;
    PVOID ShimmedDriverHint;
} KSE_ENGINE, *PKSE_ENGINE;
```

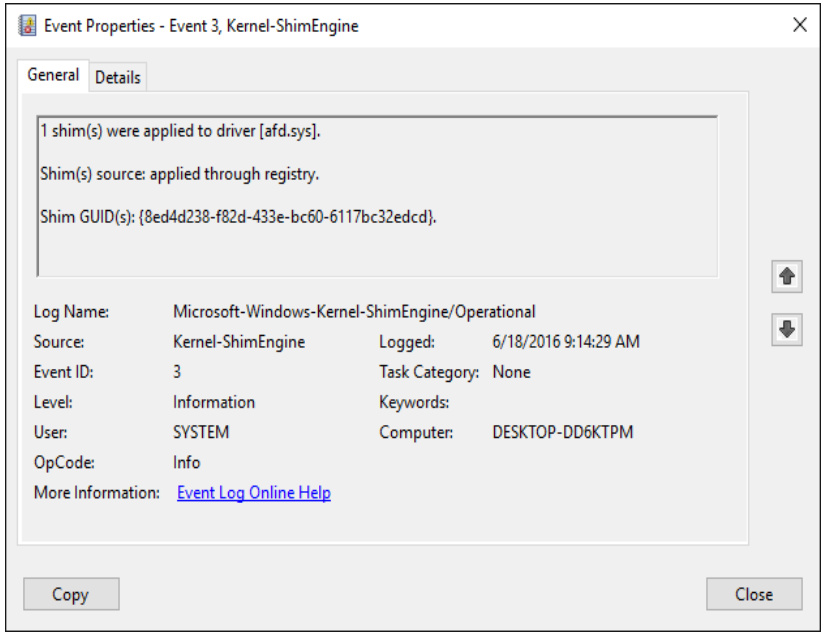

CONFIGURING THE SHIM ENGINE

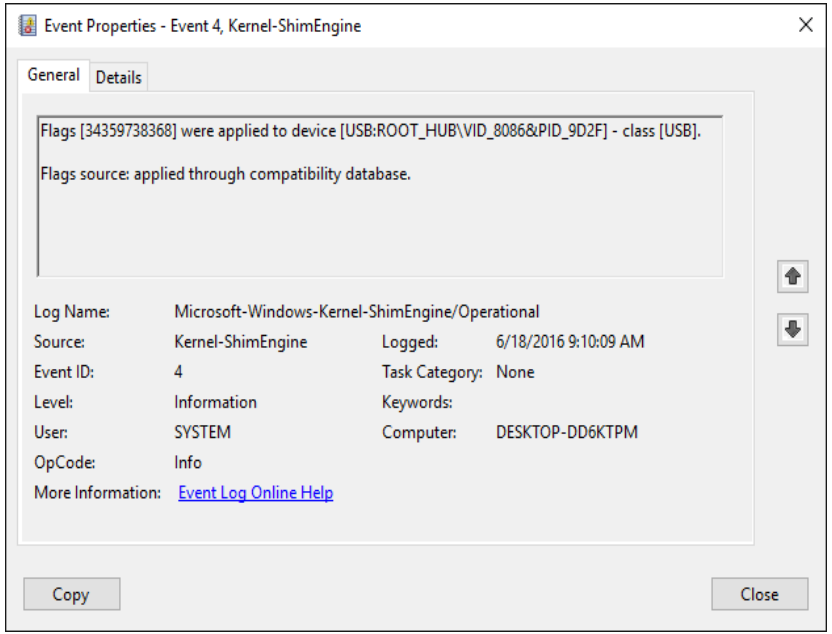
- Configurable through \\Registry\\Machine\\System\\CurrentControlSet\\Policies\\Microsoft\\Compatibility
  - DisableDeviceFlags REG_DWORD (0x1)
  - DisableDriverShims REG_DWORD (0x1)
- Exposed through Group Policy Editor
  - Windows Components->Device and Driver Compatibility->Device Compatibility Settings
  - Windows Components->Device and Driver Compatibility->Driver Compatibility Settings
- However, also configurable through \\Registry\\Machine\\System\\CurrentControlSet\\Control\\Compatibility
  - DisableFlags REG_DWORD (0x1, 0x02, 0x03)
  - Directly maps to KseEngine.DisableFlags

DEBUGGER ANALYSIS

- At anytime during kernel execution, the **KsepDebugFlag** variable can be set to a bitmask that determines what types of KSE errors should be seen (errors, warnings, etc…)
  - Then use WinDBG with a remote kernel connection, or DbgView from SysInternals
- Can also use KsepHistoryErrors array, which contains an array of “errors”
  - Each filename has its own identifier
  - Followed by the line number
  - Followed by the NTSTATUS code
- On top of that, if “Debug & Analytic” logs are enabled in Event Viewer, all errors are logged as text too
- Operational logs provide good forensic analysis, and are always turned on

EVENT LOG EXAMPLE



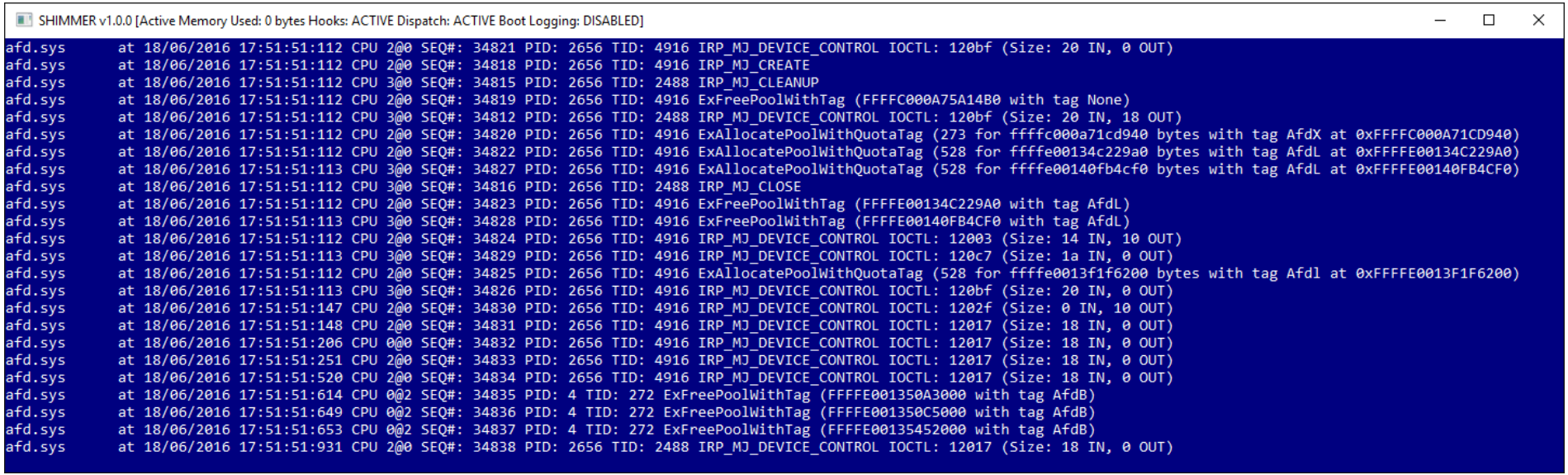


CROWDSTRIKE

DRIVERMON

TOOL FOR MONITORING SHIMMED DRIVERS

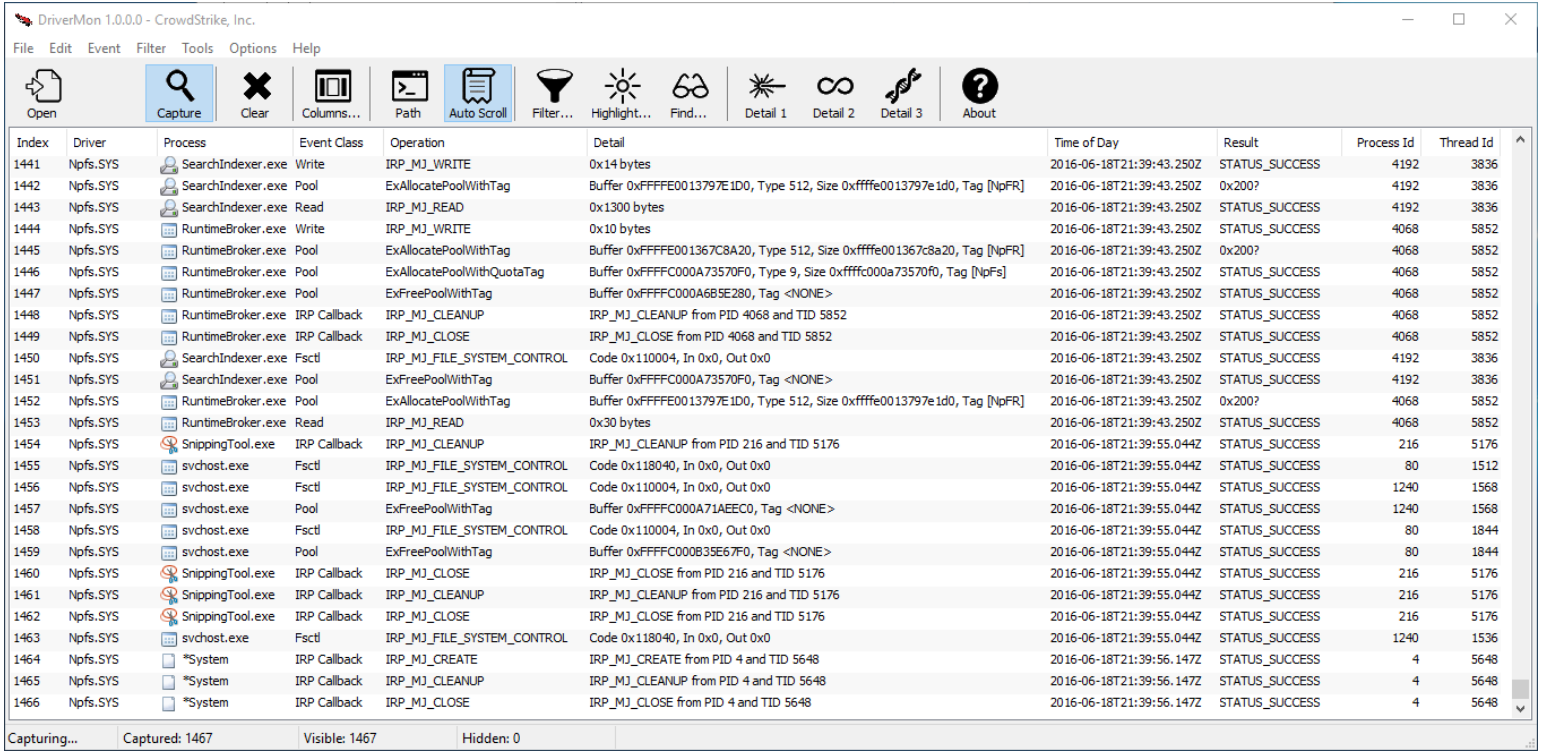
- Initially started out with a command-line tool very similar to “Poolmon”



GUI TOOL – DRIVERMON

- Asked Robin Keir, which has extensive tool building experience if a “Procmon-like” UI would be possible
  - Introducing the ability to search, filter, and highlight operations
  - Non-destructive filtering and boot logging
  - Export to .CSV and import/analysis later on a different machine
  - Custom decoding of IOCTLs, and raw buffer analysis
- Started working on the tool on spare time – initial 1.0 release will be ready shortly
  - Currently working on wrapping up UI, EULA, Build & Signing…
- Will be available from https://www.crowdstrike.com/resources/community-tools/
  - Will announce release on Twitter in a few weeks
- Let us know your suggestions/bugs/etc. at drivermon@crowdstrike.com
  - Or reach out on Twitter

DRIVERMON



BONUS

Hooking the Shim Engine

DRIVER_EXTENSION HIDDEN FIELDS

- DRIVER_EXTENSION is documented in wdm.h… but doesn't match symbols!
  - A number of fields, including KseCallbacks, are undocumented

```
typedef struct _DRIVER_EXTENSION {
    struct _DRIVER_OBJECT *DriverObject;
    PDRIVER_ADD_DEVICE AddDevice;
    ULONG Count;
    UNICODE_STRING ServiceKeyName;

    //
    // Note: any new shared fields get added here.
    //

} DRIVER_EXTENSION, *PDRIVER_EXTENSION;
```

```
nt!_DRIVER_EXTENSION
   +0x000 DriverObject     : Ptr64 _DRIVER_OBJECT
   +0x008 AddDevice        : Ptr64     long
   +0x010 Count            : Uint4B
   +0x018 ServiceKeyName   : _UNICODE_STRING
   +0x028 ClientDriverExtension : Ptr64 _IO_CLIENT_EXTENSION
   +0x030 FsFilterCallbacks : Ptr64 _FS_FILTER_CALLBACKS
   +0x038 KseCallbacks     : Ptr64 Void
   +0x040 DvCallbacks      : Ptr64 Void
   +0x048 VerifierContext  : Ptr64 Void
```


KSEGETIOCALLBACKS

- Turns out that *KseGetIoCallbacks()* merely returns DriverObject->DriverExtension->KseCallbacks.
- These are setup in *KseShimDriverIoCallbacks()* as soon as the driver loads

```
status = KsepGetShimCallbacksForDriver(
             DriverObject->DriverStart,
             &IoCallbacks);
if ( status >= 0 )
{
    kseCallbacks = KsepPoolAllocateNonPaged(0x100ui64);
    if ( kseCallbacks )
    {
        driverInit = DriverObject->DriverInit;
        if ( driverInit )
        {
            hookedDriverInit = IoCallbacks.DriverInit;
            if ( IoCallbacks.DriverInit )
            {
                kseCallbacks->DriverInit = driverInit;
                DriverObject->DriverInit = hookedDriverInit;
            }
        }
```

```
v5->KseCallbacks = kseCallbacks;
status = 0;
```


HOOKING THE HOOKS

- This means that any driver which is being shimmed by a provider (such as storport.sys) can potentially be hooked without having to touch the DRIVER_OBJECT structure
- Instead, one can modify the KseCallback pointers that exist in the DRIVER_EXTENSION
- These pointers are not monitored by PatchGuard
- Obviously easy to detect by forensic software if made aware of these fields and the possibility
  - By the way, Driver Verifier works in a similar way, and may also be vulnerable to this technique

CROWDSTRIKE

CONCLUSION

THOUGHTS ON THE KSE

- How is Shimmer able to shim arbitrary drivers?
  - Normally even if a shim is applied through registry, it must match a KSHIM in the SDB
  - However, the SDB pre-defines some “non-existent” shims, such as “kmautofail”
  - This allows us to spoof one of those shims (which nobody registers for), and then shim arbitrary drivers
- Additionally, certain pre-defined shims are marked as “delay-load”
  - Allows for sneaky way of ensuring persistence, by having a driver called kmautofail.sys for example, and then setting kmautofail as a shim for an active driver
- Security software / IR forensics should probably
  - Validate/monitor the Compatibility registry key for new “Driver” keys being created
  - Probably a good idea to monitor “Device” keys as well
- Remember that **SDBs are not signed** – watch out for malicious Drvmain.sdb!

CROWDSTRIKE

Q & A

Ask away