THE SHADOW NETWORK STACK IN WINDOWS 8

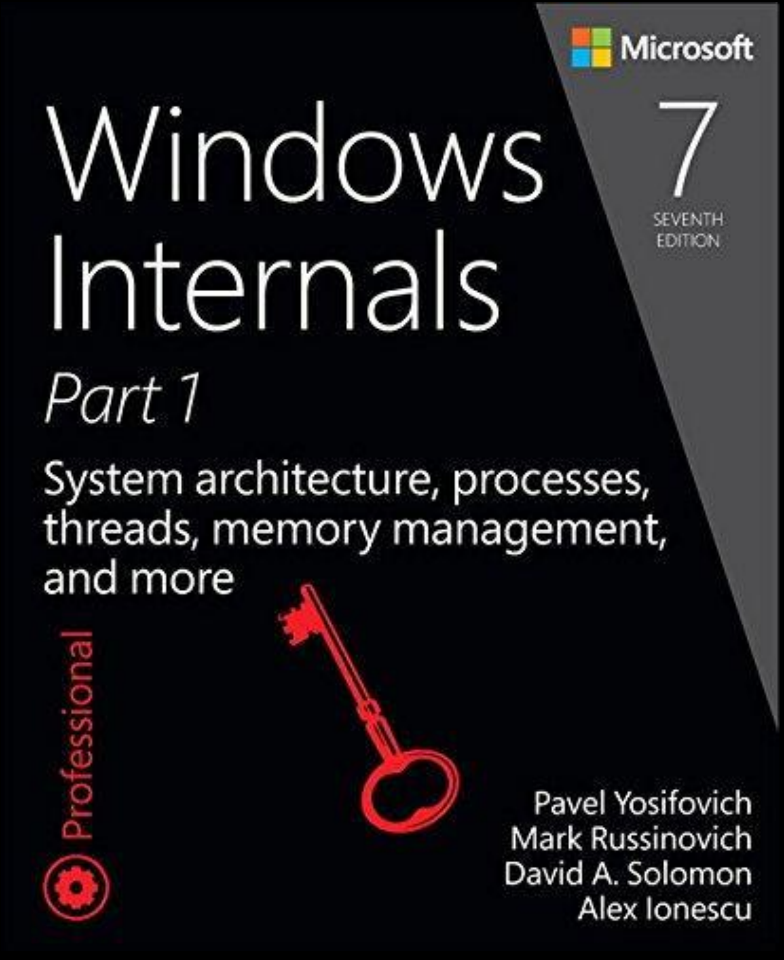
Forget NDIS, TDI or NIC Drivers

Alex Ionescu [@aionescu]

CanSec West 2018, Vancouver

BIO

- VP of EDR Strategy and Founding Architect at CrowdStrike
  - Previously worked at Apple on iOS Core Platform Team
- Co-author of *Windows Internals 5th-7th Editions*
- Reverse engineering NT since 2000
  - Lead kernel developer of ReactOS (now UEFI Boot Loader)
- Instructor of worldwide Windows internals classes
- Author of various tools, utilities and articles
- Conference speaking:
  - SyScan 2012-2015, Infiltrate 2015, OffensiveCon 2018
  - NoSuchCon 2013-2014, Breakpoint 2012, EkoParty 2017
  - Recon 2010-2018, EuskalHack 2017, CanSecWest 2018
  - Blackhat 2008, 2013-2016, 18?
- For more info, see www.alex-ionescu.com or @aionescu



AGENDA

- Introduction / Bio / Motivation
- What is KDNET?
- Initializing KDNET Outside of Its Comfort Zone
- Using KDNET to Communicate
- HAL KD Callbacks and PCI Access
- (BONUS) BugCheck I/O Callbacks – if time permits
- Concluding Thoughts / Q & A

MOTIVATION

- Windows and 3rd party PSPs provide good visibility into network I/O and file-based I/O at a variety of levels
  - IFS Filter Drivers / MiniFilters
  - WFP Callouts (Winsock LSPs, TDI Filters in older days)

- Attackers ultimately (usually) need to
  - Perform lateral movement
  - Exfiltrate
  - Beacon / C2 Communications

- All these actions require highly visible network comms I/O
  - We want to hide this I/O

HOW IT'S DONE TODAY

- Rootkit-level hooks are applied into NDIS and/or TCPIP drivers to modify "Net Buffer Lists" and other similar structures
  - Typically packet data is added-on, or incoming packets are re-directed
  - Other times, completely co-existent uIP stacks are built SxS with Windows stack
- Ultimately these approaches go through NDIS
  - LWF/IM or other PSP hooks can still see packets
  - PatchGuard and other anti-rootkit/forensic tools will typically discover hooks
  - Of course, visible to firewalls/routers (unless further compromise on infrastructure)
- A few "non-NDIS" approaches show up in academia and highly targeted attacks
  - These require intimate knowledge of the hardware and a custom driver to talk to it

HOW IT COULD BE DONE INSTEAD…

- It turns out Windows also needs to send packets without disturbing the main OS stack
  - And even before the main OS stack is initialized
  - That means no NDIS, no TCPIP, not even NIC drivers are present

- Windows can even do this from UEFI
  - Even from VTL1
  - Even from Hyper-V itself

- How does Windows achieve this?
  - And can we re-purpose this technology?

USE CASES

- With this ability, one can think of a number of both “Blue” and “Red” use cases

- Blue Team (until Red Team finds out about these techniques and blocks you)
  - Debugging/tracing over a secure, hidden channel not subject to tampering
  - Beaconing/checking-In to some server/console to indicate PSP health/comms loss
  - Detecting anomalies by comparing a packet sent through OS stack vs. this stack
  - Streaming bugchecks (such as when fuzzing) without OS dump stack I/O working

- Red Team (until Blue Team finds ways to break your techniques)
  - Stealth network communications (w.r.t endpoint software)
  - Potential to hook regular network communications as well (TBD)
  - Ability to communicate out even when the network card driver is disabled ☺

CAVEATS / CONS

- Once the 'shadow stack' will be activated, if the user is using an existing network card driver, it will be cut-off
  - Usually will sense the 'network cable as being disconnected'
  - There may be a way around this (TBD)

- If the device driver is physically _removed_ and PnP has no driver for the NIC, Windows will not enable DMA or PCI BARs in general for the device
  - Could technically re-enable this manually, but not done in current PoC
  - Works fine if the NIC is disabled/suspended/unbound, however

- No WLAN NIC supported for obvious reasons (baseband FW) – nor USB NICs (yet)

- Obviously, perimeter visibility of packet TX/RX still exists
  - However, there may be a way to do packet injection/modification on the existing stack (TBD)

WHAT IS KDNET?

Windows Network Debugging

WINDOWS KERNEL DEBUGGING

- Traditionally done over UART (NS16550 basically) – all but gone from modern machines
  - And very slow piping for virtual machines
  - Also supported IEEE1394 (FireWire) – all but non-existent on most PCs (not for VMs)
  - And eventually USB 2.0 – with special controllers, firmware, and cables (not for VMs)

- Windows 8 changed all that by adding
  - Network Debugging – from UEFI till the OS Afterlife on 30+ branded NICs
    - Including the Intel E1000 which is endlessly virtualizable on all VM products
  - USB 3 XHCI Debugging – part of the standard and with regular cables and firmware
  - Windows 8.1 and later kept expanding the NICs supported, even adding 10G support

- Today a DDK exists for vendors to write their own KDNET extensibility modules

KDNET EXTENSIBILITY MODULES

- Plugins to the main Kdnet.dll library with a single import – KdInitializeLibrary
  - Import provides access to a table of exports and shared data that plugin can use
  - Plugin provides internal routines that KDNET library will call into
  - Similar to NDIS port/miniport model
- Exports provide two interfaces for interacting with library
  - Packet-based (NIC, WLAN, USB) vs byte-based (serial)
  - Yes, "NET" is *meant* to potentially encompass a greater series of hardware
  - This new architecture is replacing all existing KD libraries one day
    - kdnet_uart16550.dll now replaces kdcom.dll, for example
- Modules run single-threaded with interrupts disabled and APs busy-waiting on IPI

LOADING EXTENSIBILTY MODULES

- Winload.efi first scans for PCI Hardware identified by BUSPARAMS or internal guess
  - Reads PCI Configuration Space for PCI Vendor Id and PCI Class
  - Loads KD_CLASS_VENDORID.DLL on disk
    - i.e.: kd_02_8086.dll (Intel NIC) or kd_07_1415.dll (OXSEMI Serial) or kd_0C_8086 (Intel USB)

- Otherwise, DBG2 ACPI Table is read (See Table 3 In DBG2 ACPI Table Specification)
  - PortType field is read to determine transport (0x8000 == Serial, 0x8003 == Network)
  - PortSubtype is read to determine vendor (for Network, this is the Vendor ID)
    - i.e.: kd_8003_5143.dll (Qualcomm USB NIC)

- Once loaded, the KdInitializeLibrary routine will be called twice
  - And this process repeats for each debug-configured component (BootMgr, WinLoad, OS)
  - KDNET.DLL has an import from KDSTUB.DLL which is overridden by loaded module name

REQUIRED ‘EXPORT’ FUNCTIONS

- An extensibility module needs to implement KdInitializeController and KdUninitializeController to kick off the hardware engine and eventually shut it down
- KdGetHardwareContextSize is setup to determine all MMIO and physical memory that will be needed to map the hardware registers as well as RX/TX buffers
  - Used to set PDEBUG_DEVICE_DESCRIPTOR Memory Length field
- KdGetRxPacket/KdGetTxPacket/KdReleaseRxPacket are used to get packet buffers
- KdSendTxPacket to send a packet
- KdGetPacketAddress/KdGetPacketLength to get packet virtual address and size

INITIALIZING KDNET OUTSIDE OF ITS COMFORT ZONE

Messing with the Loader Block – again!

INITIALIZING KDNET

- When Winload.efi needs to initialize KDNET, it calls KdInitialize in a number of phases
  - Phase0 sets up the entire stack, Phase1 and later are used for ETW, Registry Status, etc…

- Nothing prevents us from importing this function from Kdnet.dll and calling it again
  - However, unless we edit Kdnet.dll's IAT (or mess with kernel structures), it will call Kdstub.dll's KdInitializeLibrary function
  - Can also move the required DLLs to a local path (\System32\Drivers, for example) and rename kd_xx_xxxx.dll to Kdstub.dll

- We will need to pass in two parameters
  - The Loader Block
  - And a KD Context

WHAT IS THE LOADER BLOCK?

- The boot loader does a lot of work to get the kernel loaded
  - Including loading the registry
  - And all the drivers
  - And the hypervisor
  - And the shim database
  - And the API set mappings
  - And the INF errata
  - And the ELAM hive
  - And the page tables
  - And the kernel imports
  - And gather boot entropy
  - And hash everything / TPM-all-the-things
  - And setup TCP/UDP for netboot if needed
  - And gather boot-time configuration parameters from firmware and BCD options

WHAT'S IN THE LOADER BLOCK?

- So the boot loader needs to pass along all that data to the kernel
- This is done by sending a parameter to its entrypoint called the loader parameter block
- This structure leaked in NT4 sources, and Win2K source, and 2003 source… and eventually made it into the Windows 7 symbols (yay)
- Contains data that KDNET will need, such as the kernel command-line options (now typically provided as BCD elements, but still ultimately a string internally)

WINDOWS 8 LOADER BLOCK

- Unfortunately, between NT4/2K/2003/7, most of the loader block stayed the same
  - That being said, they added a header in Windows 7, which adds nice forward-compatibility
  - Even better since real UEFI support is only in Windows 7+
- But post Windows 7, they were smart enough to remove the symbol
  - And breaking changes were made to the structure, such as supporting ELAM
  - Symbol hasn't come back since ☹
- But that's OK, they leaked the entire structure in the Windows 10 SDKs for TH2
  - And they leaked it again in RS1…
  - And actually leaked in early RS2 Preview SDKs too – gone now but RS2+ has what we need

LOADER BLOCK EXTENSION

- For compat reasons, the loader block doesn't have *all* the information the kernel uses
  - The rest is in the "Extension" structure
    - Which has lots of sub-extensions (HyperV extension, NetBoot extension, headless extension, etc…)
  - Again, all in Windows 7 symbols as well as in 2015-2016 Windows 10 SDKs

- Ultimately, *our* loader block needs to have
  - OsMajorVersion == 10, OsMinorVersion == 0
  - LoadOptions pointing to a proper load option string (we'll see the rules next)
  - Extension pointing a Loader Block Extension (can be zeroed out, but must be present)

- These offsets haven't changed, and only top-level offsets are read from Extension
  - Partly used to generate MAC through SMBIOS UUID data

MINIMAL LOAD OPTIONS

- The minimal number of options we have to set are
  - "ENCRYPTION_KEY=1.2.3.4" ➔ Sets up a simple encryption key – can be any valid value
  - "HOST_IP" ➔ Sets up the IP address of the machine we'll be talking to – this is static
    - Newer versions now support "HOSTIPV6" for an IPV6 address instead
  - "HOST_PORT" ➔ Sets up the port address of the machine we'll be talking to – also static

- We can also setup some additional options
  - "NO_DHCP TARGET_IP" ➔ Indicates the IP address of our *own* machine and disables DHCP
  - "NO_KDNIC" ➔ Disables KdNic.sys NDIS intermediate miniport after the OS has booted up

- Other possible options
  - "KD_TRANSPORT_LOGGING" ➔ Enables KdPrint during debugging/development time
  - "VERIFY_HOST_MAC" ➔ Checks that received packets are coming from MAC of HOST_IP

WHAT ABOUT KDCONTEXT?

- Older version leaked online in WRK and other places
  - KdpControlCPending, KdpDefaultRetries – filled out by KDNET
- A flags field was later added
  - Followed by a pointer back to the PDEBUG_DEVICE_DESCRIPTOR for the device
  - And a pointer to private transport data
- This data usually isn't useful/relevant to us, but needs to be allocated at initialization stage in persistent memory
  - And then passed around to the send/receive functions, among others
- Really just allocate a blank page (or some large global) and you'll be fine

KDNET TROUBLESHOOTING

- KdLogIndex
- KdNetAlwaysDisconnect
- KdNetArpPacketReplyFailures
- KdNetArpPacketsHandedOff
- KdNetArpPacketsHandled
- KdNetBailCreateFile
- KdNetBailLoadSymbols
- KdNetBailPrintString
- KdNetBailTraceIo
- KdNetControlChannelPacketsDropped
- KdNetControlChannelPacketsHandled
- KdNetD0TransitionCount
- KdNetD3TransitionCount
- KdNetData
- KdNetDataChannelInitialized
- KdNetDebuggerInitialize0Count
- KdNetDecryptKdPacketBadDirection
- KdNetDecryptKdPacketBadFlags
- KdNetDecryptKdPacketBadLength
- KdNetDecryptKdPacketBadPadding
- KdNetDecryptKdPacketBadSequenceNumber
- KdNetDecryptKdPacketBadSignature
- KdNetDecryptKdPacketBadVersion
- KdNetDecryptKdPacketFailedAuthentication
- KdNetDecryptKdPacketLostPackets
- KdNetDecryptKdPacketNoDataChannel
- KdNetDecryptKdPacketSucceeded
- KdNetDecryptKdPacketTooShort
- KdNetDhcpDiscoverResponseTime
- KdNetDhcpInvalidNetworkResponse
- KdNetDhcpLeaseExpired
- KdNetDhcpLeaseValid
- KdNetDhcpPacketsHandled
- KdNetDhcpPacketsProcessed
- KdNetDhcpRequestResponseTime
- KdNetDisconnectDebuggerHost

KdNetErrorStatus	00000001C001CC0C
KdNetErrorStatusLog	00000001C0025760
KdNetErrorString	00000001C001A878
KdNetErrorStringLog	00000001C0025A00
KdNetExports	00000001C001AAA0
KdNetExtensibilityInitCount	00000001C001AA10
KdNetFirstBoot	00000001C00254C0
KdNetGratuitousArpFailures	00000001C001D078
KdNetGratuitousArpsSent	00000001C002336C
KdNetHardwareContextSize	00000001C00259E0
KdNetHardwareID	00000001C001CC08
KdNetInitialConnectAttempts	00000001C001D0B8
KdNetInitialConnectTime	00000001C001D088

KdNicReceiveEntered	00000001C001CC10
KdNicReceivePacket	00000001C0007658
KdNicReceivePacketsDropped	00000001C001D048
KdNicReceivePacketsDroppedInDebugger	00000001C001D070
KdNicReceivePacketsHandled	00000001C001D04C
KdNicReceivePacketsIgnored	00000001C001CC20
KdNicReceivePacketsTruncated	00000001C001CC28
KdNicReceiveReentered	00000001C001CC24
KdNicSendEntered	00000001C001D074
KdNicSendPackets	00000001C0007A28
KdNicSendPacketsFailed	00000001C001D050
KdNicSendPacketsFlushed	00000001C001D058
KdNicSendPacketsSent	00000001C001D040
KdNicSendPacketsUnavailable	00000001C001CC14
KdNicSendQueuedPackets	00000001C0007854
KdNicSendReentered	00000001C001D044
KdNicSendUnmappedBufferCount	00000001C001CC18
KdNicSendUnmappedPartialBufferCount	00000001C001CC1C
KdNicTruncatedPacketLengths	00000001C001CC40
KdPerformanceFrequency	00000001C00254B8

KdNetReceivedPackets	00000001C0025508
KdNetReconnectRunningTimeout	00000001C0023368
KdNetReconnectTimestamp	00000001C001D0D0
KdNetResendRequestsReceived	00000001C00254D0
KdNetRestartController	00000001C001D134
KdNetRetryCount	00000001C001A264
KdNetRxEthernetPacketsHandedOff	00000001C001D080
KdNetRxIpPacketsHandedOff	00000001C001F244
KdNetRxIpPacketsMalformed	00000001C001D0F0
KdNetRxIpPacketsMatched	00000001C001D08C
KdNetRxIpv6PacketsHandedOff	00000001C001D0C0
KdNetRxIpv6PacketsMalformed	00000001C001D0E8
KdNetRxIpv6PacketsMatched	00000001C001D0B4
KdNetRxKdPacketsHandedOff	00000001C0023394
KdNetRxPacketId	00000001C001A268
KdNetRxPacketTooSmallForIp	00000001C00254A8
KdNetRxPacketTooSmallForIpv6	00000001C001D10C
KdNetRxPacketTooSmallForUdp	00000001C001D0DC
KdNetRxPacketsDiscarded	00000001C001D0C8
KdNetRxPacketsFailed	00000001C001D100
KdNetRxPacketsMatched	00000001C002338C
KdNetRxPacketsReceived	00000001C001D0EC
KdNetRxPacketsReleased	00000001C00254A0
KdNetRxUdpPacketsHandedOff	00000001C001F25C
KdNetRxUdpPacketsMalformed	00000001C001F258
KdNetRxUdpPacketsMatched	00000001C00254A4
KdNetSendKdPacketNoDataChannel	00000001C00254B0
KdNetSendPingPacketCalled	00000001C001D0D8
KdNetSentPackets	00000001C00254F0
KdNetSentPingPacket	00000001C001D130
KdNetSerialExtensibility	00000001C001ABA0
KdNetShutdownController	00000001C001D0B0
KdNetTxError	00000001C00254AC
KdNetTxOk	00000001C001D098
KdNetTxPacketId	00000001C001A294
KdNetTxTimeout	00000001C001D0F8
KdNetUpdateTargetRandom	00000001C0023364
KdNetWaitForRxPacketCalled	00000001C001D128
KdNetWaitForRxPacketStalls	00000001C001D0E4
KdNetWaitForRxPacketTimeouts	00000001C0023370
KdNicData	00000001C001A600
KdNicEnabled	00000001C001A260
KdNicHasConnected	00000001C001D054

WHAT HAPPENS NEXT?

- At this point, KDNET will check the status of KdNetExtensibilityInitCount before binding with the extensibility module
  - Which is why, if the machine *has already enabled KDNET for debugging* this technique is not immediately usable as is – unless the IP settings and encryption key are OK
  - Or maybe there's a way to modify them… (coming up soon)

- Calls are eventually made to
  - KdEnumerateDebuggingDevices, KdSetup/ReleasePciDeviceForDebugging
  - These functions are called through the HAL Private Dispatch Table

- Controller initialization is performed, then network stack is setup when a potential initial DHCP offer, if not at least an initial gratuitous ARP and potential ping
  - Reply is expected to confirm things are working as expected

USING KDNET TO COMMUNICATE

Droppin’ Dimes

ONCE IT'S ALL DONE…

- Once the KDNET network stack is working, we have access to a simple set of exports
- KdReceivePacket – receives a KD packet from the Host IP
- KdSendPacket – sends a KD packet to the Host IP
- These packets (based on flags) have to obey certain rules and structure (see next)
- KdSetHiberRange – will call back into extensibility module to save its data for S4

KD PACKET RULES (CLIENT-SIDE)

- First level of abstraction – which you need to understand in both your client and server – is that you will be sending KD packets
  - Same definitions and structures as the original KDCOM library all the way back to NT4
    - See Windbgkd.h

- PACKET_TYPE_KD_CONTROL_REQUEST (10) is likely the best choice during RX
  - Expects a STRING structure which contains the Buffer and Length of the packet header
  - Call RtlInitEmptyAnsiString with your input buffer and length to configure it
  - Pass in NULL to KdReceivePacket's MessageData and DataLength parameters

- For TX, you can actually just use PACKET_TYPE_UNUSED
  - Bypasses any special checks/code paths, and expects your data into yet another STRING
  - NOTE: Don't send more than MTU – 1408 bytes is the maximum size KD allows

KD PACKET RULES (SERVER-SIDE)

- KD Packet-Based communication has certain rules that the server side needs to implement (transparent to actual client code – but implemented in KDNET stack)

- Client will send CONTROL_PACKET_LEADER packets indicating either
  - PACKET_TYPE_KD_ACKNOWLEDGE – which you can use to detect packet loss/sync issues
  - PACKET_TYPE_KD_RESEND – which you must use to retransmit your last packet ID

- Client will also send PACKET_LEADER packets
  - These are the ones actually coming from the machine's calls to KdSendPacket
  - You must acknowledge these packets back with PACKET_TYPE_KD_ACKNOWLEDGE
    - Unless client has set KD_CONTEXT->Flags to 1 (KD_CONTEXT_FLAGS_NO_ACK)

- Client can also set KD_CONTEXT->Flags to 4 (KD_CONTEXT_FLAGS_ASYNC)
  - KDNET calls extensibility module with TRANSMIT_ASYNC (does not wait on hardware)

WHAT A KD PACKET LOOKS LIKE

```
typedef struct _KD_PACKET
{
    ULONG PacketLeader;
    USHORT PacketType;
    USHORT ByteCount;
    ULONG PacketId;
    ULONG Checksum;
} KD_PACKET, *PKD_PACKET;
```

- PacketLeader will be CONTROL_PACKET_LEADER (iiii) or PACKET_LEADER (0000)
- PacketType is one of the PACKET_TYPE enumeration values
- Checksum is computed as 32-bit rolling sum
- PacketId can be set to zero on every send, will be KD’s internal monotonic ID on receive

IT'S NOT THAT SIMPLE

- Recall that we provided an ENCRYPTION_KEY to KDNET
  - That is because all protocol communication is encrypted with an AES-256 session key
  - This is generated based on the ASCII key as well as other internal details (not relevant)
- Therefore, server needs to implement key negotiation algorithm
- And correctly handle a KDNET-specific header that is added on top of the KD packet
- This adds a layer of complexity that would be nice to ignore (we'll see soon)
  - But server side will still receive KD packet *after* the KDNET-specific header

WHAT A KDNET HEADER LOOKS LIKE

- ```
  typedef struct _KDNET_PACKET_HEADER
  {
      ULONG Signature;
      USHORT Version;
      USHORT Flags;
      ULONGLONG SequenceKey;
  } KDNET_PACKET_HEADER, *PKDNET_PACKET_HEADER;
  ```
- Signature will be ‘MDBG’ (Modern DeBuG?)
- Version is 4 on Windows 10 (2 and 3 on older Win8/8.1 systems – 1 on beta Win7)
- Flags is only filled on the initial ‘offer packet’ (see next)
- SequenceKey is a monotonic sequence number encoded with the packet size, swapped
  ```

OFFER PACKET

- On first connection (as well as if host reconnection is requested/supported), this packet is sent to initialize the session key and state

- Additional Flags field is now used
  - 0x1 – This is an offer packet (all other packets have zero)
  - 0x2 – “SEND_KD_STATUS” was requested, and KdEnteredDebugger (in NTOS) is TRUE
  - 0x4 – KdEventLoggingEnabled (in NTOS) is TRUE, additional trace data in offer packet

- Server-side should consume this data and use it to initialize the session key to allow communications to function
  - Server should know what key to use to be able to read this packet
  - But let’s just skip all of this…

KDNET DATA

- KDNET uses a large global variable called KdNetData to encode its entire state
  - Contains the DEBUG_DEVICE_DESCRIPTOR setup through WinLoad and HAL (coming up)
  - Full network stack state (target/host IP and port, MAC, DHCP lease and state)
  - Contains all encryption settings (user key, session key, nonce)
  - GUIDs to identify the host and VM NIC (if synthetic)
  - Timestamps for bring up and power down

- Interesting configuration parameters are present as well
  - VerifyHostMac (configurable through load options)
  - DebuggerState (is there someone on the other side, and shared user data wants debug)
  - ConnectionState (is there someone on the other side, at all)
  - EncryptionState (should encryption be used)

MODIFYING KDNET DATA

- Obviously it would be great if we could read this data (for example, to confirm our DHCP lease, host MAC, etc..)
  - And maybe change it as well – allowing dynamic port/IP changes outside of load options

- Even better if we can modify it so that we can set “DebuggerState” to DBG_STATE_ACTIVE and trick the KDNET engine
  - Turns out that this is **<u>not needed nor necessarily desireable</u>**

- And so that we can set “EncryptionState” to ENCRYPTION_STATE_DISABLED and stop the KDNET packet encryption code from being active
  - This will leave the entire KDNET_PACKET_HEADER empty (must still account for it)
  - The offer packet still comes in encrypted at all times – but you can ignore it (no ACK)

- TBD next

HAL KD CALLBACKS AND PCI ACCESS

Runtime Hooks and Backdoor PCI Routines

HAL KD ROUTINES

- As part of talking to hardware, KDNET and the extensibility module obviously need access to PCI configuration space to and memory map the registers and/or IO ports
  - There's no Plug-and-Pray manager support, so device must be 'enumerated' and 'configured' by 'shadow PnP stack' – this is where the HAL KD routines come into play

- The state we operate in means we can't be sending IRPs to PCI.SYS
  - So there must also a 'shadow PCI driver' – (once again, it's the HAL itself)

- First, KDNET will call KdEnumerateDebuggingDevices, passing in the loader block (which should be our fake one), a PDEBUG_DEVICE_DESCRIPTOR for us to fill out, and a PDEBUG_DEVICE_FOUND_FUNCTION callback – which is unused
  - Our job is to return the descriptor back with Initialized == FALSE, Configured == TRUE and then fill out all the required fields

FILLING OUT A DEVICE DESCRIPTOR

- NameSpace ➔ KdNameSpacePCI or ACPI (won't cover ACPI scenario here)
- PortType ➔ 0x8003 (recall the spec) for Ethernet
- Bus, Segment, Slot ➔ For a PCI device, its B:D:F
- BaseClass, SubClass, ProgIf, VendorID, DeviceID ➔ All from PCI Config Header
- BaseAddress[N] ➔ Based on BARs, using CmResourceTypeXxx
  - TranslatedAddress must be MMIO mapped with MmMapIoSpace(Ex) – not at HIGH_LEVEL
- Memory ➔ Length, MaxEnd, Start and VirtualAddress must be filled out
  - VirtualAddress must be the entire size of the hardware state/buffers needed (up to 16MB)
    - Something like MmAllocateContiguous(Node)Memory is good here – not at HIGH_LEVEL
  - Start is result of MmGetPhysicalAddress on the VirtualAddress
- Set various flags such as DBG_DEVICE_FLAG_BARS_MAPPED/SCRATCH_ALLOCATED

INITIALIZING THE DEBUG DEVICE

- Now that KdEnumerateDebuggingDevices has returned a configured, uninitialized debug device descriptor, KdSetupPciDeviceForDebugging will be called next

- This is where you would normally want to do any PCI-specific initialization (such as potentially only enabling memory decoding/bus mastering at *this* stage, or doing the memory mappings)
  - In our case, we can be lazy and all we *really* need to do is set Configured ➔ TRUE
    - That's because the NIC driver is present and so the NT PnP/PCI stack has done the leg work

- But there's *one* more thing…
  - Remember how KdNetData <u>*contains*</u> the DEBUG_DEVICE_DESCRIPTOR?
  - I used that word on purpose – it's not a pointer to the descriptor, it <u>is</u> the descriptor
    - The same one we are being passed in to the HAL functions – CONTAINING_RECORD FTW

IMPLEMENTING THE HAL ROUTINES

- These KDNET->HAL calls are made through the HalPrivateDispatchTable, and the HAL provides its own functions there – so why are *we* implementing them?
  - When you call enumerate, the HAL will enumerate <u>the debug devices it was told about at boot</u>
    - i.e.: none if the user isn't doing remote debugging
  - If the user *has* enabled debugging already, again, that's a different use case/approach we would need to take here, since none of this is needed – but we must now find KdNetData

- To provide our own functions, we must overwrite HalPrivateDispatchTable with pointers to our own functions
  - PatchGuard does not protect this structure as it changes dynamically after boot
  - However other Microsoft technologies will monitor and may 'Sense' (hehehe) changes

ONE LAST NOTE ABOUT SETUP

- Setup does require an MMIO mapping and a physically contiguous memory allocation
  - Therefore, this part cannot be done in the HIGH_LEVEL context. User of this library should have these buffers/registers prepared ahead of time at up-to DISPATCH_LEVEL
  - Or, you can actually use KdMapPhysicalMemory64 which uses the HAL Heap and works ☺
    - This is located in the HalPrivateDispatchTable

- Also, how does one read PCI configuration space/registers from the KD routines?
  - KdSetPciDataByOffset, KdGetPciDataByOffset are in the HalPrivateDispatchTable as well
  - These can be used at any time and provide full, synchronized access to the bus
  - They support VTL1, IOMMU, MMIO, Hyper-V behaviors, not just CF8/CFC
    - This is why drivers and companies who just IN/OUT CF8/CFC make me cry/die a little inside
    - I mean, do *you* handle "PciAmdK8SpecialLocationHack"?

(BONUS) BUGCHECK I/O CALLBACKS

Unravelling Some Magic

BUGCHECK CALLBACKS

- A somewhat little-known feature of Windows is that you can register callbacks on every Sad Face Of Sorrow (aka BSOD)
  - These are called “bugcheck callbacks”
- The bugcheck callbacks that allow you *add* data to a crash dump file are well documented and have examples
- But there is also a callback type that *gives* you access to the data in the crash dump
  - Literally as it’s about to be written back to disk
- The API is KeRegisterBugCheckReasonCallback, with the KbCallbackDumpIo type

KBCALLBACKDUMPIO ISSUES

- Windows calls this in a few places
  - As it's writing the crash dump header
  - As it's writing additional crash dump-type specific data (bitmap block, minidump block)
  - And for pages that contain actual physical data (if that's the case), the sector blocks

- Windows knows what it's writing, but only provides back:
  - KbDumpIoHeader – crash dump header (always virtual)
  - KbDumpIoBody – additional crash dump header data, or dump pages (virtual or physical)
  - KbDumpIoSecondaryData – 3rd party crash dump data (always virtual)
  - KbDumpIoComplete – signal that the dump is done (no memory)

- Therefore must build and maintain internal state (no way to know virt vs. physical)
  - There is one public Xen driver out there using this – broken assumptions

CORRECT (PROPOSED) LOGIC LOOP

- KbDumpIoHeader – assert this is the first call (bail out if not)
  - Write/send virtual data into <x> and add running tally of data received so far
  - Once sizeof(DUMP_HEADER64) received, switch to 'state 2'
    - May want to assert that 2 calls of PAGE_SIZE have happened (since that's the header size)
- KbDumpIoBody
  - If in state 2, check signature (FDMP for Full Bitmap Dump, for example)
    - If additional data needed (such as BITMAP_DUMP), write virtual data, get HeaderSize, enter 'state 3'
    - If no additional header data needed (dump pages have started), enter 'state 4'
  - If in state 3, write virtual data, and add running tally of data received so far.
    - Once >= HeaderSize, enter 'state 4'
  - If in state 4, write physical data (running tally may be useful for troubleshooting)
- KbDumpIoSecondaryData – write virtual data (running tally may help), enter 'state 5'
- KbDumpIoComplete – assert state is 5, assert running tally is indicated dump size, end.

ACCESSING PHYSICAL PAGES

- These callbacks get executed at HIGH_LEVEL IRQL (all interrupts disabled)
  - No IPI ->No TLB flush -> Not even non-pageable memory allocations work

- Accessing physical memory usually involves mapping pages for it
  - In Windows terms this usually involves building an MDL and allocating PTEs/PFNs for it
    - This isn't possible at HIGH_LEVEL

- Undocumented API exists which uses a 'static MDL' and pre-allocated VA from boot
  - Build MDL on the stack: `UCHAR mdlBuffer[sizeof(MDL)+(17*sizeof(PFN_NUMBER))];`
  - Fill out the PFN array (`MmGetMdlPfnArray`) with up to 16 pages from the callback's input
  - Call `MmMapMemoryDumpMdl` to map the pages in the reserved crash dump VA
  - Call `MmGetSystemAddressForMdlSafe` to get the virtual address where they were mapped

PARTING THOUGHTS

Future PoC Improvements & References

LET'S REVISIT WHAT WE CAN DO

- As long as the user hasn't activated remote kernel debugging (or if we're OK reusing that hardware), we can send and receive KD packets (with or without encryption)
  - Through some memory tricks, we can even change the host/target IP and port
  - Packets are UDP, but protocol has built-in retransmit/acknowledgement logic
  - We can do sync or async sends (and disable acknowledgment logic)

- We are limited by the 4-5 major vendors Microsoft supports 'inbox', for a total of 30-50 network cards (and no USB/WLAN cards for now)
  - But we could actually write our own extensibility module – much simpler than a full blown network stack and full Windows-compliant Ethernet driver

- We can use this library in any context – even in the middle of a bugcheck
  - Using this library will cut off the user's network connection as currently described (boo!)

TAKING IT TO THE NEXT LEVEL

- When kernel debugging is *legitimately* used, your NIC still works for WAN/LAN use
- This is because of KdNic.sys
  - This is an NDIS Miniport driver that now loads instead of your real NIC driver
  - It captures all Ethernet traffic directed to the NIC, and **sends it through KDNET instead**
  - There are some pretty interesting data structures/queues that are used to achieve this
- This means that we could technically bring the network back up from the user's perspective with some NDIS/registry trickery (hot-swap the driver with KdNic.sys)
  - As well as offers some interesting packet 'injection'/'redirection' techniques as a completely different use case
- Also, the fact that a driver needs to be present for Windows to enable the BARs is merely because our PCI configuration code in our HAL KD Callbacks is minimalistic
  - We could implement more robust code to manually enable the BARs and memory decoding that's needed for the extensibility module to work

ANYTHING FOR MICROSOFT TO FIX ?

- I suppose one could argue that Microsoft should make it ‘impossible’ to use this library unless the user has opted into kernel debugging
  - In fact, it kind of does – it expects a boot-time loader block, configured for debugging, and it calls HAL functions that are only setup based on boot-time debugging settings, and even its import table needs to be ‘bound’ correctly by the boot-time loader

- But ultimately – the consumer of the library has Ring 0 privileges and can fake all state
  - In fact, this is exactly what the PoC is doing
  - Layers of obfuscation could be added to make it harder – but this is just playing games
    - Or PatchGuard could detect use – same caveat though

- That being said, there may be opportunities here with TCB Secure Launch and VTL1
  - But ultimately even without this library, attacker can do all this. **KDNET is merely *convenient***

REFERENCES

- Check out your WDK\Debuggers\ddk folder for the complete Extensibility Module Development Kit
  - Contains entire API documentation, and relevant data structures
  - Contains source code (!) for Intel 10G and 1G network cards, as well as RealTek
  - Contains source code for UNDI network cards through UEFI runtime firmware (!!)
  - Also contains source code for 16550 and SIIG serial ports, and Qualcomm, Synopsys and ChipIdea USB Controllers

- Also take a look at windbgkd.h and other past talks (including mine at Recon) on the KD protocol and its internals

- Reversing Kdnet.dll and Winload.efi/Ntoskrnl.exe will complete the picture for you

THANK YOU!

Q & A