The Adventures of a KeyStroke
An in-depth look into Keyloggers on Windows
Emre TINAZTEPE
What you will learn?

- Completing this training, you will be able to:
  - Use a kernel debugger for malware analysis,
  - Understand the threats posed by keyloggers,
  - Detect / Remove all kinds of keyloggers,
  - Understand how a keylogger works in greatest detail,
  - Be prepared to Advanced Persistent Threats!

- We will cover a lot of OS Internal structures.

- Without dealing with OS Internals, you can’t be sure that your system is clean.
Who am I?

- Emre TINAZTEPE

- Ex military:
  - Maltepe Military High School (21 / 421)
  - Turkish War Academy (8 / 838)
  - Passed half of his life in the army (First Lieutenant)
  - Resigned 3 years ago.

- Low level guy who likes to deal with OS Internals

- Currently leading a Malware Analysis Team

- Responsible of malware analysis and mobile av dev.
Methodology

- Hard to easy because it all starts at hardware 😊
- If you have question, just interrupt me.
- Hands on labs combined with theory.
  - Labs are made in a Win 7 x32 machine.
Why keyloggers?

- Because keyboard is the device you command your computers.

- Logging keys from a PC provides the malware authors with great power.

- Best way to gather intelligence.
  - Russia is said to be switching to “typing machines” in critical institutions.

- Best way to get rich 😊
Before we begin

- Please download these files:
  
- Please turn your AV/Windows Defender OFF!
VirtualBox
Window Detective

Window picking tool
# Rootkit Unhooker

![Rootkit Unhooker LE v3.8.388.590 Service Release 2](image)

<table>
<thead>
<tr>
<th>Id</th>
<th>Service Name</th>
<th>Hooked</th>
<th>Address</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NtAcceptConnectPort</td>
<td></td>
<td>0x82A86F97</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>1</td>
<td>NtAccessCheck</td>
<td></td>
<td>0x828C8B55</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>2</td>
<td>NtAccessCheckAndAuditAlarm</td>
<td></td>
<td>0x82A16D35</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>3</td>
<td>NtAccessCheckByType</td>
<td></td>
<td>0x82832897</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>4</td>
<td>NtAccessCheckByTypeAndAuditAlarm</td>
<td></td>
<td>0x82A88585</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>5</td>
<td>NtAccessCheckByTypeResultList</td>
<td></td>
<td>0x82A9F112</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>6</td>
<td>NtAccessCheckByTypeResultListAndAuditAlarm</td>
<td></td>
<td>0x82AF9127</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>7</td>
<td>NtAccessCheckByTypeResultListAndAuditAlarm</td>
<td></td>
<td>0x82AF9170</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>8</td>
<td>NtAddAtom</td>
<td></td>
<td>0x82A08551</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>9</td>
<td>NtAddBootEntry</td>
<td></td>
<td>0x82B13992</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>10</td>
<td>NtAddDriverEntry</td>
<td></td>
<td>0x82B138E7</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>11</td>
<td>NtAdjustGroupsToken</td>
<td></td>
<td>0x82A01D29</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>12</td>
<td>NtAdjustPrivilegesToken</td>
<td></td>
<td>0x82A93E8A</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>13</td>
<td>NtAlertResumeThread</td>
<td></td>
<td>0x82AEBDF3</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>14</td>
<td>NtAlertThread</td>
<td></td>
<td>0x82A36CB7</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>15</td>
<td>NtAllocateLocallyUniqueIds</td>
<td></td>
<td>0x82A0E999</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>16</td>
<td>NtAllocateReserveObject</td>
<td></td>
<td>0x829A49E3</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>17</td>
<td>NtAllocateUserPhysicalPages</td>
<td></td>
<td>0x82ADDCCF</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>18</td>
<td>NtAllocateLuids</td>
<td></td>
<td>0x829F527A</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>19</td>
<td>NtAllocateVirtualMemory</td>
<td></td>
<td>0x82A37C5C</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>20</td>
<td>NtApcAcceptConnectPort</td>
<td></td>
<td>0x82A31459</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>21</td>
<td>NtApcCancelMessage</td>
<td></td>
<td>0x829E52E6</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>22</td>
<td>NtApcConnectPort</td>
<td></td>
<td>0x82A3576</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
<tr>
<td>23</td>
<td>NtApcCreatePort</td>
<td></td>
<td>0x82A0CDA9</td>
<td>C:\Windows\system32\ntkrnlpa.exe</td>
</tr>
</tbody>
</table>
GMER / Tuluka
Process Explorer

![Process Explorer Screenshot]

- System Idle Process: CPU 97.72%, Private Bytes 0K, Working Set 24K
- System: CPU 0.07%, Private Bytes 44K, Working Set 808K
- Intermits: CPU 0.38%, Private Bytes 0K, Working Set 220K
- csrss.exe: CPU <0.01%, Private Bytes 1.264K, Working Set 3.380K
- wininit.exe: CPU 0.07%, Private Bytes 0K, Working Set 220K
- services.exe: CPU 0.07%, Private Bytes 0K, Working Set 220K
- WmiPrvSE.exe: CPU 0.20%, Private Bytes 1.396K, Working Set 3.668K
- VBoxService.exe: CPU 0.20%, Private Bytes 1.396K, Working Set 3.668K
- svchost.exe: CPU 0.20%, Private Bytes 1.396K, Working Set 3.668K
- svchost.exe: CPU 0.20%, Private Bytes 1.396K, Working Set 3.668K

- ACPI.sys: Description: ACPI Driver for NT, Company: Microsoft Corporation, Path: C:\Windows\system32\drivers\ACPI.sys
- d.sys: Description: Ancillary Function Driver for WinSo... Company: Microsoft Corporation, Path: C:\Windows\system32\drivers\afsd.sys
- RAS Agil... Company: Microsoft Corporation, Path: C:\Windows\system32\DRIVERS\AgileVpn.sys
- ndxata.sys: Description: Storage Filter Driver, Company: Advanced Micro Devices, Path: C:\Windows\system32\drivers\vmdxata.sys
- syncmac.sys: Description: MS Remote Access serial network... Company: Microsoft Corporation, Path: C:\Windows\system32\DRIVERS\asyncmac.sys
- atapi.sys: Description: ATAPI IDE Miniport Driver, Company: Microsoft Corporation, Path: C:\Windows\system32\drivers\atapi.sys
Windbg
Windbg Cheat Sheet

- `lm` : Lists loaded modules (drivers, dlls)
- `!process -1 0` : Displays current process
- `!process 0 0 winlogon.exe` : Displays info for the process
- `.process EPROCESS` : Switches to the process (implicit)
- `bp ADDRESS` : Puts a breakpoint at the address
- `g,p,t` : Go, Step, Trace
- `bl` : Lists the breakpoints
- `bc INDEX` : Clears the BP indicated by the index
- `bd INDEX` : Disables BP temporarily
- `.echo` : Outputs a string
Windbg Cheat Sheet

- .cls : Clears the screen
- u ADDRESS / SYMBOL : Unassembles the address
- uf ADDRESS OF FUNCTION : Unassembles the whole func.
- db ADDRESS : Dumps the address.
- ? poi(ADDRESS) : Displays the address pointed by.
- !pic / !ioapic : Displays information about interrupt controllers.
- !drvobj \Driver\kbdclass 0x7: Display the specified driver.
- !devobj OBJECT : Display information about device obj.
Let’s infect ourselves

- Restart RED VM, make sure it is not in “KERNEL DEBUG” mode.
- Go to Materials/Keyloggers directory
- Double click “Elite Keylogger.exe”
- Install with default settings (Click NEXT multiple times)
- Choose “Allow” in case Windows Defender consents.
- Restart the VM in non debug mode.
- Write “unhide” on start menu and provide a password at least 3 chars long.
- Fire up a “Notepad” and write your name in it.
- Please also provide your Credit Card number 😊😊😊
- Do not save it please, it is safer 😊😊😊
You are infected now 😞

- We will see how to detect keyloggers in the following ours.
- For the moment, please restore your VM to snapshot “Informatics” and start your VM in “Kernel Debugging” Mode.
Ready to dive?
An overview of a mother board

1. CPU
2. Front Side Bus (FSB)
3. North Bridge
4. Internal Bus (IB)
5. South Bridge
6. Peripherals
An overview of a mother board

- Bus is a communication system that transfers data between components inside a computer,

- FSB is the CPU's connection to the North Bridge and through it to rest of the system,

- North Bridge is a high-speed hub that in most systems connects the CPU to the graphics card and to RAM,

- South Bridge is a slower-speed hub that connects the CPU to the rest of the system.
South Bridge (SB)

- It is also named as “Input/Output Controller Hub”.
- Responsible from the peripheral device connections such as USB, PCI, PS/2, Sound and etc.
- Why two bridges?
  - Same as the idea of having RAM, Cache, Register
  - Simpler design which is easy to modify in terms of IO capabilities.
- It is what you actually connect your keyboard to.
PS/2 Keyboard Controller

- A component of a mainboard which handles the connection between a motherboard and a PS/2 keyboard.
PS/2 Keyboard

- Just a limited computer system which scans a wireframe continuously for finding a closed/opened circuit.
The PS/2 Keyboard is a device that talks to a PS/2 controller using serial communication.

The PS/2 Keyboard accepts commands and sends responses to those commands, and also sends **scan codes** indicating when a key was pressed or released.

The keyboards processor includes its own timer, 33 instruction set, and can even access 128K of external memory.
Talking to a Keyboard?

- A PS/2 Keyboard accepts many types of commands,
- Each command is one byte,
- Some commands have data byte/s which must be sent after the command byte,
- The keyboard typically responds to a command by sending either an "ACK" (to acknowledge the command) or a "Resend" (to say something was wrong with the previous command) back.
Talking to a Keyboard?

- Commands must be sent one at a time (IN/OUT),
- Some commands have data byte/s which must be sent after the command byte,
- 0xFE (resend) expects a command to be sent again, while 0xFA (ACK) means command is successfully processed.
### PS/2 Keyboard Controller/Encoder Ports

<table>
<thead>
<tr>
<th>IO Port</th>
<th>Access Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Keyboard Encoder</td>
</tr>
<tr>
<td>0x60</td>
<td>Read</td>
<td>Read Input Buffer</td>
</tr>
<tr>
<td>0x60</td>
<td>Write</td>
<td>Send Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Keyboard Controller</td>
</tr>
<tr>
<td>0x64</td>
<td>Read</td>
<td>Status Register</td>
</tr>
<tr>
<td>0x64</td>
<td>Write</td>
<td>Send Command</td>
</tr>
</tbody>
</table>

- Port 0x60 is what we use for reading and writing data to/from the keyboard device.
- The Status Register contains various flags that indicate the state of the PS/2 controller such as the state of input/output buffers.
- The Command Port (0x64) is used for sending commands to the PS/2 Controller (not to PS/2 Devices).
### Some of the PS/2 Keyboard Encoder Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Data</th>
</tr>
</thead>
</table>
| 0xED    | Set LEDs   | Bit0: ScrollLock  
|         |             | Bit1: NumberLock  
|         |             | Bit2: CapsLock    |
| 0xEE    | Echo        | For diagnostic purposes. |
| 0xF0    | Get/set current scan code set | 0: Get current scan code set  
|         |             | 1: Set scan code set 1  
|         |             | 2: Set scan code set 2  
|         |             | 3: Set scan code set 3  |
| 0xF4    | Enable scanning | - |
| 0xF5    | Disable scanning | Discard key presses or mouse movements. Used especially while identifying the attached PS/2 device in order to prevent messing up the identification process. |
A scan code set is a set of codes that determine when a key is pressed or repeated, or released.
Scancodes and Code Sets

- There are 3 scan code sets, normally on PC compatible systems the keyboard itself uses scan code set 2 and the keyboard controller translates this into scan code set 1 for compatibility.
How to read scan codes?

- Poll the Bit 0 of status register and then read the data from port 0x60
  - Too much CPU time!
  - Multiple PS/2 devices lead to problems for differentiating the data.

- Wait for an interrupt to occur
  - Much better!
  - Wait for an IRQ 1 / IRQ 12 (wait for the next slide😊)
What is an interrupt?

- Interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.

Do this and let me know when it’s done!
I am a little bit busy 😊

It’s done!

Let’s see what you have.

CPU

Device (Hard disk, Keyboard)
Why called as “IRQ”? 

- Each peripheral device requests to “Interrupt the CPU” this is why it is a “Request” which may or may not be handled by the CPU.

- Question: What happens when multiple devices send an IRQ at the same time?

- Answer: The one with a higher IRQL gets processed while the others keep waiting.
Interrupt Handling

- One of the best advantages of an interrupt driven device is the ability to overlap device’s processing time with the CPU’s activity.
Where do I connect my device?

- Question: If we have 2 or more devices attached to our mainboard, how will we differentiate one device’s interrupt from the other?

- Answer: Each motherboard has an at least one Programmable Interrupt Controller (PIC / APIC) into which your external devices get connected. You do not have to do anything, all is done seamlessly by this electronic circuit.
Programmable Interrupt Controller

- OMG! What is an interrupt controller?
- One of the most important chips making up the x86 architecture,
- Without it, the x86 architecture would not be an interrupt driven architecture,
- The function of the PIC is to manage hardware interrupts and send them to the appropriate system interrupt.
- This way, no polling needed 😊
APIC

- More sophisticated interrupt handling and the ability to send interrupts between processors.
- In an APIC-based system, each CPU is made of a "core" and a "local APIC".
The external I/O APIC is part of Intel’s system chip set. Its primary function is to receive external interrupt events from the system and its associated I/O devices and relay them to the local APIC as interrupt messages.

It is programmed by the OS before enabling interrupt handling mechanism.
What magic CPU does to handleIRQs?

- There is no magic, we tell it what to do.
- We create a table of function pointers and tell the CPU where it resides.
- This table is called as “Interrupt Descriptor Table” and the address for this table is hold by a register called IDTR (IDT register).
Intel x86 CPU Modes

- 3 + 1 Modes of operation is supported by CPU.
  - Real Mode
  - Virtual 8086 Mode
  - Protected Mode
  - System Management Mode
Real Mode

- Also called **real address mode**.
- Real mode is characterized by a 20-bit segmented memory address space and unlimited direct software access to all memory, I/O addresses and peripheral hardware.
- Real mode provides no support for memory protection, multitasking, or code privilege levels.
- Before the release of the 80286, which introduced Protected mode, real mode was the only available mode for x86 CPUs.
- In the interests of backwards compatibility, **all x86 CPUs start in real mode** when reset.
Protected Mode

- Also called protected virtual address mode.

- It allows system software to use features such as virtual memory, paging and safe multi-tasking designed to increase an operating system's control over application software.

*Create some tables for Virtual Memory and set PE bit in CR0 register*
Virtual 8086 Mode

- Also called **virtual real mode**.

- Allows the execution of real mode applications that are incapable of running directly in protected mode while the processor is running a protected mode operating system.
System Management Mode

- Is an operating mode in which all normal execution (including the operating system) is suspended, and special separate software (usually firmware or a hardware-assisted debugger) is executed in high-privilege mode.

- SMM is a special-purpose operating mode provided for handling system-wide functions like:
  - Handle system events like memory or chipset errors,
  - Manage system safety functions, such as shutdown on high CPU temperature and turning the fans on and off,
  - Emulate motherboard hardware that is unimplemented or buggy.
More on SMM

- A powerful mode of CPU which can even preempt the whole OS!!!

- SMM is entered via the SMI (system management interrupt)

- SMM is a really good place to execute malicious software *without modifying the structures created by OS.*

Here comes the karate kick!
#1 SMM Rootkits
Did you know that you can see the keystrokes even before they are handled by “Interrupt Handler”? 

![Diagram of SMM Rootkits]

**Normal Path**

- Key Press
- I/O APIC
  - IRQ: 0, 1, 2, 23
  - Vector: -, 0x93, -
- Local APIC
- IDT
  - Int: 0x00, 0x93
  - Handler: -, 0xA806304
- OS Handler
  - push esp
  - push ebp
  - push ebb
  - push esi
  - push edi
  - sub esp, 54
  - mov ebp, esp

**Infected Path**

- I/O APIC
  - IRQ: 0, 1, 2, 23
  - Vector: -, SMI #
- SMM Handler
  - Log/Transmit the keycode and send a message to the local APIC to invoke the normal keyboard handler.
- Description
  - The normal operation (top-half) is subverted allowing the new SMM handler to log/transmit the keycodes and then forward the interrupt.
The implementation

1. Use SMRAM Control Register (SMRAMC)
   - Check bit D_OPEN (is SMRAM visible to outside code)
   - Check bit D_LCK (is SMRAMC is read-only, if yes a reset is needed)

2. If D_LCK bit is clear:
   1. Set D_OPEN bit to make SMRAM visible to protected mode code,
   2. Copy the SMM Handler code to the handler portion of SMRAM defined by Intel Docs,
   3. Clear D_OPEN bit and set D_LCK bit to protect our evil code 😊

3. We are invisible!
Routing IRQ 1 to Malicious SMM Handler

1. Modify the I/O APIC in such a way that when ever a user presses a key, our SMM code is executed,

2. SMM Handler reads the scan code, logs it and sends a special command to keyboard for overcoming the problem of a popped up scancode.

3. This in turn makes the next data written into the keyboard buffer available for OS Keyboard Interrupt handler,

4. Send an IPI to ourself for handling an emulated IRQ 1!

5. Let the OS think it is a real scancode generated by the keyboard encoder 😊
Pros & Cons

1. Pros
   1. Totally invisible to the OS!
   2. No need to change any OS created structures.
   3. Very hard to detect.

2. Cons
   1. Works only with PS/2
   2. Limited to single processor system
   3. D_LCK bit is already set on modern systems 😐
#2 IDT Hooking
Structure of an Interrupt Descriptor Table

1. Protected Mode counterpart of Real Mode Interrupt Vector Table (IVT),

2. Contains at most 256 entries.

3. Each entry is 8 bytes long and they are structured as defined below:

```c
nt!_KIDTENTRY
+0x000 Offset : Uint2B
+0x002 Selector : Uint2B
+0x004 Access : Uint2B
+0x006 ExtendedOffset : Uint2B
```
Keyboard Interrupt is not mapped to IDT#1???

1. Where is IRQ 1 mapped? Which IDT Entry???
   - “IOAPIC makes IRQ and remaps IRQ to IDT.”

```
kd> !ioapic
IoApic @ FEC00000 ID:1 (11) Arb:0
Inti00.: 00000000`000100ff  Vec:FF FixedDel  Ph:00000000   edg high  m
Inti01.: 01000000`00000991  Vec:91 LowestDi Lg:01000000  edg high
Inti02.: 00000000`000100ff  Vec:FF FixedDel  Ph:00000000   edg high  m
Inti03.: 00000000`000100ff  Vec:FF FixedDel  Ph:00000000   edg high  m

kd> !idt -a
31: 84866058 i8042prt!i8042KeyboardInterruptService (KINTERRUPT 84866000) NO I/O APIC
91: 84864058 i8042prt!i8042KeyboardInterruptService (KINTERRUPT 84864000)
```

2. Methods for retrieving the vector address:
   - Use APIC
   - Scan kernel memory
   - Use the kernel API function (HalGetInterruptVector)
How to read scancode?

1. It’s as easy as executing an “in al,60h” instruction 😊
   - IN instruction empties the data, we need to put it back into its place for system’s use.

2. Here is an excerpt from the Keyboard Controller command set:

   Command 0xd2: Write keyboard output buffer

   Write the keyboard controllers output buffer with the byte next written to port 0x60, and act as if this is a keyboard generated data.
Here is the method

Record the keystroke into a buffer and execute the special keyboard controller command for putting it back into place.
Structure of a KINTERRUPT

```
kd> !idt 51
Dumping IDT: 80b95400
91: 84864058 i8042prt!i8042KeyboardInterruptService (KINTERRUPT 84864000)
kd> dt nt!KINTERRUPT 84864000
+0x000 Type : 0x22
+0x002 Size : 0x632
+0x004 InterruptListEntry : _LIST_ENTRY [ 0x84064004 - 0x84064004 ]
+0x00c ServiceRoutine : 0x8a71d43a unsigned char i8042prt!i8042KeyboardInterruptService+0
+0x010 MessageServiceRoutine : (null)
+0x014 MessageIndex : 0
+0x018 ServiceContext : 0x88020208 Void
+0x01c SpinLock : 0
+0x020 TickCount : 0xfffffff
+0x024 ActualLock : 0x85025368 - 0
+0x028 DispatchAddress : 0x8294e8b0 void nt!KiInterruptDispatch+0
+0x02c Vector : 0x91
+0x030 Irql : 0x8
+0x031 SynchronizeIrql : 0x8
+0x032 FloatingSave : 0
+0x033 Connected : 0x1
+0x034 Number : 0
+0x038 ShareVector : 0
+0x03f Pad : [3] ""
+0x04c Mode : 1 ( Latched )
+0x050 Polarity : 0 ( InterruptPolarityUnknown )
+0x054 ServiceCount : 0
+0x058 DispatchCount : 0xfffffff
+0x060 RawId : 0
+0x068 DispatchCode : 0x85653554
```
Where does this code come from?

1. **KINTERRUPT->DispatchCode** is actually a modified version of **KiInterruptTemplate**.

2. Can be easily modified for different kinds of interrupts such as **KiChainedDispatch**, **KiFloatingDispatch**.
What does a DispatchCode do?

- Acquire the SpinLock of ServiceRoutine
- Raise the IRQL to DEVICE_IRQL
- Call the ServiceRoutine
- Lower IRQL
- Release the SpinLock of ServiceRoutine

This is the point where “Interrupt Servicing” takes place!

i8042KeyboardInterruptService
How to intercept

1. Put an inline hook into DispatchCode’s prolog,
2. Create a new KINTERRUPT object and make EDI point to it,
3. Replace the ServiceRoutine field of KINTERRUPT,
4. Inline hook the ServiceRoutine.
Windows Driver Model

- A layered design with support for adding drivers into the stack dynamically.
- Great design for management.
- Allows another driver to filter some other driver’s packets.
Keyboard Device Stack
What is an IRP?

- A structure which is used by the I/O manager for defining a request targeted to a device.
- Reading a file, writing to a file and much more operation is handled with IRPs.
- Each IRP has a Major code which makes it possible to call appropriate handler for that IRP.
i8042prt.sys

1. Port driver for 8042 compatible keyboard and mouse devices.

2. Handles the interrupt for a keyboard device and delivers it to the system.

3. Contains good candidates for a keylogger.
i8042prt.sys

GLOBALS Globals:

i8xGetByteAsynchronous

i8042KeyboardIsrDpc

i8042KeyboardInterruptService

i8xWriteDataToKeyboardQueue
i8042prt.sys Overview

- **I8xGetByteAsynchronous**
  - Uses `Globals.Read` method internally

- **Call IsrHookCallback if one is registered**
  - May also terminate the ISR by modifying `ContinueProcessing` param

- **I8xQueueCurrentKeyboardInput**
  - Queues a DPC for giving a chance to class driver for processing the input data at DISPATCH_LEVEL (`I8042KeyboardIsrDpc`)

- **I8xWriteDataToKeyboardQueue**
  - Adds the INPUT data into keyboard input data queue
#4 i8042prt!Globals Hack
i8042prt.sys GLOBALS structure

Globals for what?
A look into i8042prt!Globals

kd> dps i8042prt!Globals
8d9540c0  85799cd8
8d9540c4  8594cab8
8d9540c8  85a52c88
8d9540cc  8281a094 hal!READ_PORT_UCHAR
8d9540d0  8281a0fc hal!WRITE_PORT_UCHAR
8d9540d4  00720070
8d9540d8  859b3c80

Replace it with your own 😊
Here we have the keystrokes, also little noisy but can be parsed with a simple script.
#5 I8xGetByteAsynchronous
I8xGetByteAsynchronous

- Defined as
  
  I8xGetByteAsynchronous(CHAR KeyboardType,CHAR*ScanCode)

- Pretty good place to hook.

- Internally uses Global.Read
#6 Hacking IsrHookCallback
**IsrHookCallback**

- Used by upper level drivers to modify the scan code in the ISR routine.
- Gets called right after scan code is retrieved from the keyboard controller.

---

**PI8042_KEYBOARD_ISR function pointer**

This topic has not yet been rated – [Rate this topic]

A PI8042_KEYBOARD_ISR-typed callback routine customizes the operation of the I8042prt keyboard ISR.

### Syntax

```c
typedef BOOLEAN ( *PI8042_KEYBOARD_ISR)(
    _In_    PVOID IsrContext,
    _In_    PKKEYBOARD_INPUT_DATA CurrentInput,
    _In_    POUTPUT_PACKET CurrentOutput,
    _In_    UCHAR StatusByte,
    _In_    UCHAR Byte,
    _Out_   PBOOLEAN ContinueProcessing,
    _In_    PKKEYBOARD_SCAN_STATE ScanState
);```

---
Hack IsrHookCallback

- As easy as modifying DEVICE_EXTENSION of port device:
  - DeviceObject->DeviceExtension->IsrHookCallback
- Right after that, keys will start flowing into our callback!
- Callback can even stop the ISR’s processing.
#7 Hacking ClassService
What does I8xQueueCurrentKeyboardInput do?

- Queues a DPC for further processing.
- DPC calls DeviceExtension->ConnectData.ClassService function for delivering the scan code information to the class driver.

Question: Can’t we hook that?

Answer: Definitely yes!

How: Replace the ClassService function with your own 😊
**I8xQueueCurrentKeyboardInput**

- Queues a DPC object for further processing the input data.
- This gives class drivers or any upper level drivers a chance to process the input data structure, even modify it!
- As soon as IRQL drops to DISPATCH_LEVEL, DPC gets executed and calls the callback supplied by Class Driver.
DPC – Deferred Procedure Call

- Time is a precious thing!
- Do what ever you can to make hardware feel better and queue a procedure to be called when everything is OK.
- This prevents keeping a CPU at a high IRQL level for a long time.
#8 I8xWriteDataToKeyboardQueue
I8xWriteDataToKeyboardQueue

- A great candidate for hooking!
- Gets the INPUT data as it’s second parameter and writes that into it’s internal data queue.
- Flags describe whether the key is down or up.

### KEYBOARD_INPUT_DATA structure

0 out of 1 rated this helpful - Rate this topic

KEYBOARD_INPUT_DATA contains one packet of keyboard input data.

**Syntax**

```cpp
typedef struct __KEYBOARD_INPUT_DATA {
    USHORT UnitId;
    ULONG MakeCode;  // This is the scan code!
    USHORT Flags;
    USHORT Reserved;
    ULONG ExtraInformation;
} KEYBOARD_INPUT_DATA, *PKKEYBOARD_INPUT_DATA;
```
#9 Filter Drivers
How to filter?

- Meaning of layer in malware authors slang:
  - “A point for injecting evil”

- Two methods:
  - IoAttachDevice API: The **IoAttachDevice** routine attaches the caller's device object to a named target device object, so that I/O requests bound for the target device are routed first to the caller.

  ```c
  NTSTATUS IoAttachDevice(
      _In_  PDEVICE_OBJECT SourceDevice,
      _In_  PUNICODE_STRING TargetDevice,
      _Out_ PDEVICE_OBJECT *AttachedDevice
  );
  ```

  - Registry hacks for devices. Set UpperFilter and LowerFilters. Upper filter drivers go between the operating system and the main driver, and lower filter drivers go between the main driver and the hardware.
Let’s check for Keyboard Filters

1. Go to Materials/Applications copy RegShot directory to your Desktop.

2. Execute “regshot.exe”

3. Set output path to “Desktop”

4. Click on “1st Shot” -> “Shot”

5. Install “Zemana AntiLogger Free.exe”

6. Go to regshot again and click “2nd Shot” -> “Shot”

7. Click “compare”

8. Search for “UpperFilters” (Upper filters for keyboard device)

9. Copy the GUID and google it. Guess what does it define?

10. Restart the machine in DEBUG MODE and execute:
   1. !drvobj \Device\kbdclass
   2. !devstack SECOND OBJECT ADDRESS
#10 IRP Handler Hooking
Keyboard Class Driver

- \\Driver\\kbdclass

- Represents a Keyboard Device either USB or PS/2.

- Used **exclusively** by the Raw Input Thread (RIT) (coming next).

```
kd> !drvobj 0x85768f08 7
Driver object (85768f08) is for:
\\Driver\\kbdclass
Driver Extension List: (id , addr)

Device Object list:
85861030 857687d8

DriverEntry: 8e1419f2 kbdclass!GsDriverEntry
DriverStartIo: 00000000
DriverUnload: 00000000
AddDevice: 8e13fdee kbdclass!KeyboardAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                 8e13a000 kbdclass!KeyboardClassCreate
[01] IRP_MJ_CREATE_NAMED_PIPE      828d20e5 nt!IoInvalidDeviceRequest
[02] IRP_MJ_CLOSE                  8e13a294 kbdclass!KeyboardClassClose
[03] IRP_MJ_READ                   8e13b0ba kbdclass!KeyboardClassRead
[04] IRP_MJ_WRITE                  828d20e5 nt!IoInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION      828d20e5 nt!IoInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION        828d20e5 nt!IoInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA               828d20e5 nt!IoInvalidDeviceRequest
[08] IRP_MJ_SET_EA                 828d20e5 nt!IoInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS          8e139f78 kbdclass!KeyboardClassFlush
```
Look at the difference

- KbdClass has a READ routine while the Port Driver doesn’t! Why?

```
Command
kd> ldevobj 0x957689d0
Device object (957689d0) is for:
   \Driver\18042prt DriverObject 857a5a28
   Current Irp 00000000 RefCount 0 Type 00000027 Flags 00002004
   DevExt 85766a83 DevObjExt 85766d18
   ExtensionFlags (0x00000000) DDB_DEFAULT_SD_PRESENT
   Characteristics (00000000)
   AttachedDevice (Upper) 857667d6 \Driver\kbdclass
   AttachedDc (Lower) 94070030 \Driver\ACPI
   Device queue is not busy.
kd> ldevobj 857a5a28 ?
Driver object (857a5a28) is for:
   \Driver\18042prt
   Driver Extension List: (id , addr)
   Device Object list:
   85768520 857689d0

   DriverEntry: 8e132138 18042prt\GeDriverEntry
   DriverStartIo: 8e1227bc 18042prt\I8xStartIo
   DriverUnload: 8e12ea31 18042prt\I8xUnload
   AddrDevice: 9e12f6c3 18042prt\I8xAddDevice

Dispatch routines:
[00] IRP_MJ_CREATE 8e12b96b 18042prt\I8xCreate
[01] IRP_MJ_CREATE_NAMED_PIPE 823d20e5 nt!IcpInvalidDeviceRequest
[02] IRP_MJ_CLOSE 8e12e3c1 18042prt\I8xClose
[03] IRP_MJ_READ 823d20e5 nt!IcpInvalidDeviceRequest
[04] IRP_MJ_WRITE 823d20e5 nt!IcpInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION 823d20e5 nt!IcpInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION 823d20e5 nt!IcpInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA 823d20e5 nt!IcpInvalidDeviceRequest
[08] IRP_MJ_SET_EA 823d20e5 nt!IcpInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS 8e125f54 18042prt\I8xFlush
```
Here is why

- Port driver doesn’t provide a read routine because it expects a “Keyboard Class Service Callback” to be registered by a class driver.

- Class driver gets the requests from the RIT and waits for KeyboardClassServiceCallback to get called by the keyboard port driver’s ISR DPC.

- This callback is registered by sending an IRP carrying a structure called as CONNECT_DATA with an IOCTL_INTERNAL_KEYBOARD_CONNECT code.

- This in turn makes the port driver record this callback routine for calling whenever an interrupt occurs.

- When ever the service callback gets called by port driver’s DPC, class driver completes the request of RIT which makes the RIT send another request.
KeyboardClassServiceCallback

- Routine which dequeues an IRP each time it gets called by the port driver’s ISR DPC.
- As soon as data is copied to the IRP, it completes the IRP with STATUS_SUCCESS.
#12 Inline hooking for ClassCallback
Hook the class callback

- We have already hacked this callback routine but in a different way. It was just a replacement of a pointer in ConnectData structure residing in port driver’s DeviceExtension.

- This time, another approach.

- Put an inline hook into KeyboardClassServiceCallback which will make us the king of scancodes 😊

- As easy as putting a 5 byte prolog into the routine.
Let’s talk about “Raw Input Thread”

- A thread of `csrss.exe` which continuously makes a read request to keyboard class device.

- It is the guy who retrieves keystrokes from the class driver and posts them to appropriate queues.

- It’s mainly a loop which makes a request and waits for that request to complete which in turn makes another request and so forth...

- Key method here is `StartDeviceRead` which sends a read request to class driver asynchronously with an APC object.
How it functions?

1. Make an async read request: `StartDeviceRead`
2. Wait for it to complete
3. Process the keyboard data in APC routine
4. Go to step one
5. Calls `ProcessKeyboardInput`
#13 Hacking Device Templates
What is a Device Template?

- A structure for keeping device specific attributes such as keyboard and mouse.

- This is where the word “KbdClass” comes from 😊

- Also contains a function pointer which is responsible for processing the Keyboard or Mouse input hence the name: “ProcessKeyboardInput”
Device Template

```asm
kd> ul: win32k!InputApc
win32k!InputApc:
91631747 8bff     nov  edi,edi
91631749 55       push ebp
9163174a 8beo     nov  ebp,esp
9163174c 56       push esi
9163174d 8cc5f08  nov  esi,dword ptr [ebp+8]
91631750 ff4e48   dec  dword ptr [esi+40h]
91631753 6460e80  test byte ptr [esi+0Eh],80h
91631757 7420     je  win32k!InputApc+0x32 (91631779)

win32k!InputApc+0x12:
91631759 e33be6a00  call  win32k!EnterCrit (916d3f99)
9163175e e8f3510600  call  win32k!EnterDeviceInfoListCrit_ (91606956)
91631763 89660dfd   and  byte ptr [esi+0Dh],0FDh
91631767 56       push esi
91631768 e8ef34ffff  call  win32k!FreeDeviceInfo (91624e5c)
9163176d e83510600  call  win32k!LeaveDeviceInfoListCrit_ (91606945)
91631772 e401e6a00  call  win32k!UserSessionSwitchLeaveCrit (916d3b7)
91631777 eb22     jmp  win32k!InputApc+0x54 (916317b9)

win32k!InputApc+0x32:
91631779 8bf80c     nov  eax,dword ptr [ebp+0Ch]
9163177e 833e00    cmp  dword ptr [eax].0
9163177f 7c14     jl  win32k!InputApc+0x4e (91631795)

win32k!InputApc+0x3a:
91631791 837e1c00   cmp  dword ptr [esi+1Ch].0
91631795 740c     je  win32k!InputApc+0x4e (91631795)

win32k!InputApc+0x40:
91631797 0f64c65c   novzx  eax,byte ptr [esi+0Ch]
9163179b 6bc03c    inul  eax,eax.3Ch
9163179c 56       push esi
9163179e ff90c78191  call  dword ptr win32k!aDeviceTemplate+0x2c (9181c72c)[eax]

win32k!InputApc+0x4e:
9163179f 56       push esi
9163179e e97c1ff3ff  call  win32k!StartDeviceRead (91631515)

win32k!InputApc+0x54:
9163179b 5c       pop  esi
9163179c 5d       pop  ebp
9163179d c20c00   ret  0Ch
```
It’s dump time

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>kd&gt; dps win32k!aDeviceTemplate</td>
</tr>
<tr>
<td>9181c700 0000014c</td>
</tr>
<tr>
<td>9181c704 91804784 win32k!GUID_DEVINTERFACE_MOUSE</td>
</tr>
<tr>
<td>9181c708 00000024</td>
</tr>
<tr>
<td>9181c70c 91808878 win32k!'string'</td>
</tr>
<tr>
<td>9181c710 91808840 win32k!'string'</td>
</tr>
<tr>
<td>9181c714 91808814 win32k!'string'</td>
</tr>
<tr>
<td>9181c718 0000f000</td>
</tr>
<tr>
<td>9181c71c 00000050</td>
</tr>
<tr>
<td>9181c720 00000000c</td>
</tr>
<tr>
<td>9181c724 0000005c</td>
</tr>
<tr>
<td>9181c728 000000f0</td>
</tr>
<tr>
<td>9181c72c 916317a5 win32k!ProcessMouseInput</td>
</tr>
<tr>
<td>9181c730 85f00f40</td>
</tr>
<tr>
<td>9181c734 00000000</td>
</tr>
<tr>
<td>9181c738 fffffffff</td>
</tr>
<tr>
<td>9181c73c 0000000ec</td>
</tr>
<tr>
<td>9181c740 91804774 win32k!GUID_DEVINTERFACE_KEYBOARD</td>
</tr>
<tr>
<td>9181c744 00000025</td>
</tr>
<tr>
<td>9181c748 91808800 win32k!'string'</td>
</tr>
<tr>
<td>9181c74c 9180873d0 win32k!'string'</td>
</tr>
<tr>
<td>9181c750 91808794 win32k!'string'</td>
</tr>
<tr>
<td>9181c754 000b0000</td>
</tr>
<tr>
<td>9181c758 00000050</td>
</tr>
<tr>
<td>9181c75c 0000001c</td>
</tr>
<tr>
<td>9181c760 00000074</td>
</tr>
<tr>
<td>9181c764 00000078</td>
</tr>
<tr>
<td>9181c768 91709e2 win32k!ProcessKeyboardInput</td>
</tr>
<tr>
<td>9181c76c 86124450</td>
</tr>
<tr>
<td>9181c770 00000000</td>
</tr>
<tr>
<td>9181c774 00000001</td>
</tr>
<tr>
<td>9181c778 00000058</td>
</tr>
<tr>
<td>9181c77c 91804624 win32k!GUID_DEVINTERFACE_HID</td>
</tr>
</tbody>
</table>
#14 Hook ProcessKeyboardInput
ProcessKeyboardInput

ProcessKeyboardInputWorker
Inside ProcessKeyboardInput

- Find the first call to worker function.
- EBX points to scancode,
- Worker function is also a good target.
#15 Hook ProcessKeyboardInputWorker
Inline Hook ProcessKeyboardInputWorker

- Pretty obvious 😊
- You can easily see that it is a 3 parameter function with the 1\textsuperscript{st} parameter as ScanCode.
#16 Hacking xxxProcessKeyEvent
xxxProcessKeyEvent

- Called by ProcessKeyboardInputWorker until each input event gets consumed.

- Lets take a look at the parameters:
  - Pointer to a Keyboard Event structure,
  - An ULONG_PTR value carrying extra information,
  - A flag indicating if key is from hardware or not.

- Performs some language specific operations.
Break on xxxProcessKeyEvent

```
kd> uf win32k!xxxProcessKeyEvent
win32k!xxxProcessKeyEvent:
  91d588d3 8bff       mov     edi,edi
  91d588d5 55          push    ebp
  91d588d6 8bec        mov     ebp,esp
  91d588d8 51          push    ecx
  91d588d9 a1b83de291  mov     eax,dword ptr [win32k!gptiCurrent (91e23db8)]
  91d588de 53          push    ebx
  91d588df 56          push    esi
  91d588e0 8b7508      mov     esi,dword ptr [ebp+8]
  91d588e3 8954fc      mov     dword ptr [ebp-4],eax
  91d588e6 8a4602      mov     al,byte ptr [esi+2]
  91d588e9 57          push    edi
  91d588ea 884508      mov     byte ptr [ebp+8].al
  91d588ed e85a0ccccf  call    win32k!GetActiveHKL (91d5354c)
  91d588f2 25ff030000   and     eax,3FFh
  91d588f7 6683f812      cmp     ax,12h
  91d588fb 0fb74602     movzx    eax,word ptr [esi+2]
  91d588ff bb00800000   mov     ebx,8000h
  91d58904 753f         jne     win32k!xxxProcessKeyEvent+0x72 (91d58945)
```
Virtual Key vs. Scan Code

- **Scancode**: Hardware Dependent
- **Virtual Key**: Independent
Virtual Key vs. Scan Code

kd> ba e1 91d588e3 "\echo KEYLOGGER;db @esi;\echo --------;g;"
kd> g
KEYLOGGER

| 8c4cca30 | 1e 58 41 00 00 00 00 00-3f 00 0b 00 00 00 1e 00 | FA |
| 8c4cca40 | 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 6c ca 4c 8c |
| 8c4cca50 | aa 9a cf 91 00 00 af ff-08 90 af 41 01 00 00 00 |
| 8c4cca60 | 80 78 5f 86 08 90 af ff-48 3b 81 82 7c ca 4c 8c |
| 8c4cca70 | f5 c2 91 08 3b af ff-48 2d 57 86 c4 ca 4c 8c |
| 8c4cca80 | f4 5a 8f 82 08 90 af ff-30 90 af ff 00 00 00 00 |
| 8c4cca90 | 80 78 5f 86 48 3b 81 82-00 01 00 00 20 fd 96 01 |
| 8c4ccaa0 | a7 17 c2 91 00 00 00 00-30 90 af ff 08 90 af ff |

---

KEYLOGGER

| 8c4cca30 | 1e 00 41 80 00 00 00 00-3f 00 0b 00 00 00 1e 00 | A |
| 8c4cca40 | 01 00 00 00 00 00 00 00 00-00 00 00 00 00 00 6c ca 4c 8c |
| 8c4cca50 | aa 9a cf 91 00 00 af ff-08 90 af 41 01 00 00 00 |
| 8c4cca60 | 80 78 5f 86 08 90 af ff-48 3b 81 82 7c ca 4c 8c |
| 8c4cca70 | f5 17 c2 91 08 3b af ff-48 2d 57 86 c4 ca 4c 8c |
| 8c4cca80 | f4 5a 8f 82 08 90 af ff-30 90 af ff 00 00 00 00 |
| 8c4cca90 | 80 78 5f 86 48 3b 81 82-00 01 00 00 20 fd 96 01 |
| 8c4ccaa0 | a7 17 c2 91 00 00 00 00-30 90 af ff 08 90 af ff |
xxxProcessKeyEvent

xxxProcessKeyEvent

UpdateRawKeyState

xxxKeyEvent
Raw Key State Table

- Just a simple array holding UP / DOWN states of keys.
- Represents the physical state of keyboard.
- Let’s put a BP on it.
Hook UpdateRawKeyState

- Two params:
  - VirtualKey
  - Key State (Make / Break)
#17 RawKeyState Sniffer
Sniffing Raw Key State Table

- Can be easily retrieved by disassembling UpdateRawKeyState.
- First LEA instruction points to it,
- AV buster 😊
Raw Key State Sniffer

- Put a BP on `UpdateRawKeyState`
- 2 bits for each VKEY (Down/Up – Toggled)
Raw Key State Sniffer Demo

- Put a BP on `UpdateRawKeyState` end address.

Offset: `gafRawKeyState + (VK * 2 bits)`
#18 Hacking xxxKeyEvent
xxxKeyEvent

- Very critical function!
- Performs the POST operation of key into input queue.
- Called by xxxProcessKeyEvent for every input event.
- Responsible from calling window hooks (wait for next slides)

**Params:**
- Virtual Key with flags,
- ScanCode
xxxKeyEvent

- Call Low Level Keyboard Hook
- Update Async Key State Table
- Post Input Message
xxxKeyEvent

- Very critical function!
- Performs the POST operation of key into input queue.
- Called by xxxProcessKeyEvent for every input event.
- Responsible of calling window hooks (wait for next slides)

Params:
- Virtual Key with flags,
- ScanCode
#19 Hacking UpdateAsyncKeyState
UpdateAsyncKeyState

- Looks same as the method for UpdateRawKeyState
- Async keystate table could also be sniffed.
#20 Hacking PostInputMessage
PostInputMessage

- What it does?
  - Calls StoreQMessage for saving the message into queue. Another target for hooking 😊
  - Foreground thread queue receives the input event.
PostInputMessage

- Put a BP on PostInputMessage.

<table>
<thead>
<tr>
<th>Evaluate expression</th>
<th>KEYLOGGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>63308032 = 03c60100</td>
<td></td>
</tr>
</tbody>
</table>

Evaluate expression:
- 0 = 00000000
- 63308032 = 03c60100

Evaluate expression:
- 256 = 00000100
- 65 = 00000041
- 1966081 = 001e0001

Evaluate expression:
- 257 = 00000101
- 65 = 00000041
- 1966081 = 001e0001

Evaluate expression:
- 512 = 00000200
- 0 = 00000000
- 63373556 = 03c700f4

- WM_KEYDOWN

- Virtual Key

- Scan Code + Flags
Here comes the second part 😊

- Thread now has an input event in its queue. Kernel is over!
- What's next?
Create Window API

- Creates a window with a Window Class.
- What is a window class?

CreateWindowEx function

Creates an overlapped, pop-up, or child window with an extended window style; otherwise, this function is identical to the CreateWindow function. For more information about creating a window and for full descriptions of the other parameters of CreateWindowEx, see CreateWindow.

```c
typedef struct tagWNDCLASS {
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCTSTR     lpszMenuName;
    LPCTSTR     lpszClassName;
} WNDCLASS, *PWNDCLASS;
```
Classes vs. Windows

Window Class

Window 1

Window 2

Window 3

Window Procedure
WNDPROC Function

- Function defined as:

  LRESULT CALLBACK WindowProc(
    _In_  HWND hwnd,
    _In_  UINT uMsg,
    _In_  WPARAM wParam,
    _In_  LPARAM lParam
  );

- Every window has one WNDPROC. This is the entry point for window messages.
#21 Hacking Window Procedures
WNDPROC Function

- We can either inline hook the WndProc or we can set a new WndProc by using GetWindowLong / SetWindowLong APIs.

```c++
LONG WINAPI GetWindowLong(
    _In_    HWND hWnd,
    _In_    int nIndex
);
```

- `GWL_WNDPROC`
- `DWL_DLGPLPROC`
#22 Subclassing a Window
Subclassing

- MSDN Blog: When you subclass a window, you set the window procedure to a function of your choosing, and you remember the original window procedure so you can pass it to the CallWindowProc function when your subclass function wants to pass the message to the original window procedure.
Subclassing

- SetWindowSubclass API is pretty good for that.
- CallWndProc could be used for retrieving keys from subclassed windows.
Classes vs. Windows

Window 1

Window 2

Window 3

Window Class
Message Loops

- Each UI Thread has one message loop for processing window messages.
#23 Hacking GetMessage / PeekMessage
GetMessage / PeekMessage

- Used for getting a message from the thread’s message queue.

```c
BOOL WINAPI GetMessage(
    _Out_    LPMSG lpMsg,
    _In_opt_ HWND hWnd,
    _In_     UINT wMsgFilterMin,
    _In_     UINT wMsgFilterMax
);

BOOL WINAPI PeekMessage(
    _Out_    LPMSG lpMsg,
    _In_opt_ HWND hWnd,
    _In_     UINT wMsgFilterMin,
    _In_     UINT wMsgFilterMax,
    _In_     UINT wRemoveMsg
);
```
GetMessage / PeekMessage

- Sniff GetMessage API call.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Process</th>
<th>Message Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:24:56.909</td>
<td>notepad.exe</td>
<td>GetMessageW (0x0008fc0c, NULL, 0, 0)</td>
</tr>
<tr>
<td>7:24:56.909</td>
<td>notepad.exe</td>
<td>GetMessageW (0x0008fc0c, NULL, 0, 0)</td>
</tr>
<tr>
<td>7:24:56.954</td>
<td>notepad.exe</td>
<td>GetMessageW (0x0008fc0c, NULL, 0, 0)</td>
</tr>
<tr>
<td>7:24:56.984</td>
<td>notepad.exe</td>
<td>GetMessageW (0x0008fc0c, NULL, 0, 0)</td>
</tr>
<tr>
<td>7:24:57.271</td>
<td>notepad.exe</td>
<td>GetMessageW (0x0008fc0c, NULL, 0, 0)</td>
</tr>
<tr>
<td>7:24:57.467</td>
<td>notepad.exe</td>
<td>GetMessageW (0x0008fc0c, NULL, 0, 0)</td>
</tr>
</tbody>
</table>

**W (User32.dll)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Pre-Call Value</th>
<th>Post-Call Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpMsg</td>
<td>0x0008fc0c</td>
<td>0x0008fc0c</td>
</tr>
<tr>
<td>hwnd</td>
<td>0x000e0294</td>
<td>0x000e0294</td>
</tr>
</tbody>
</table>

- message: WM_CHAR, wParam = 97, lParam = 1075707905, time = 286000
- pt: {x = 961, y = 461}, wParam = 65
- hWnd: NULL
- wMsgFilterMin: 0
- wMsgFilterMax: 0
#24 Hacking Translate and Dispatch
TranslateMessage / DispatchMessage

- Sniff TranslateMessage / DispatchMessage API calls.
TranslateMessage

- Translate to what?

```
TranslateMessage
  GetMessage
    WM_KEYDOWN
    WM_CHAR
```

140
DispatchMessage

- Calls the Window Procedure of a Window Class.
- Hooking it will definitely give you a lot power.
#25 Hacking Counterparts
Kernel Mode Counterparts

- The APIs which are used for message handling and delivering such as DispatchMessage, GetMessage, PeekMessage.

- All of them have their kernel mode counterparts starting with NtUser*. NtUserGetMessage, NtUserPeekMessage, NtUserTranslateMessage.

- These could be inline hooked by kernel mode keyloggers.

- Best example for this is “Elite Keylogger” (newest versions)

- Pretty effective!
Inspecting Kernel Mode Counterparts

- Anti-rootkits such as GMER, KernelDetective or Tuluka could be used for detecting these kind of modifications.

<table>
<thead>
<tr>
<th>Function</th>
<th>Start Address</th>
<th>End Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NtUserGetListBoxInfo</td>
<td>0x9283DC2D</td>
<td>0x9283DC2D</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetMenuBarInfo</td>
<td>0x9283B634</td>
<td>0x9283B634</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetMenuIndex</td>
<td>0x9283E19B</td>
<td>0x9283E19B</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetMenuItemRect</td>
<td>0x927D191E</td>
<td>0x927D191E</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetMessage</td>
<td>0x927D7E7</td>
<td>0x927D7E7</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetMouseMovePointsEx</td>
<td>0x9283E8D1</td>
<td>0x9283E8D1</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetObjectInformation</td>
<td>0x9275E06C</td>
<td>0x9275E06C</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetPriorityClipboardFormat</td>
<td>0x9283E4A8</td>
<td>0x9283E4A8</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetProcessWindowStation</td>
<td>0x927682E7</td>
<td>0x927682E7</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetRawInputBuffer</td>
<td>0x928418D3</td>
<td>0x928418D3</td>
<td>-</td>
</tr>
<tr>
<td>NtUserGetRawInputData</td>
<td>0x92841309</td>
<td>0x92841309</td>
<td>-</td>
</tr>
</tbody>
</table>

Should be within the limits of win32k.sys address space.
#26 SSDT Shadow Hooking
What is SSDT Shadow?

- Just a simple table residing in win32k.sys module.
- Holds the addresses of system services.
- This table is the glue between user mode APIs and the kernel mode counterparts.
- Hooking this table is so easy, and also effective.
How it is used?

NtUserGetMessage

User Mode

Kernel Mode

sysenter

SSDT Shadow

W32pServiceTable

NtUserGetMessage
How to check?

- We can use anti-rootkits
- Windbg can also be used for displaying SSDT Shadow Table.

```
kd> dps win32k!W32pServiceTable L200
9234b000 922d7b91 win32k!NtGdiAbortDoc
9234b004 922efae8 win32k!NtGdiAbortPath
9234b008 92147216 win32k!NtGdiAddFontResourceW
9234b00c 922e6aff win32k!NtGdiAddRemoteFontToDC
9234b010 922f1296 win32k!NtGdiAddFontMemResourceEx
9234b014 922d83ae win32k!NtGdiRemoveMergeFont
9234b018 922d8442 win32k!NtGdiAddRemoteMMInstanceToDC
9234b01c 921ff7fb win32k!NtGdiAlphaBlend
9234b020 922f0ac1 win32k!NtGdiAngleArc
9234b024 922f0d75 win32k!NtGdiArcInternal
9234b028 922f0fb2 win32k!NtGdiBeginGdiRendering
9234b030 922efb5c win32k!NtGdiBeginPath
9234b034 921f45cb win32k!NtGdiBitBlt
9234b038 922f0f05 win32k!NtGdiCancelDC
9234b03c 922f3b38 win32k!NtGdiCheckBitmapBits
9234b040 922efa63 win32k!NtGdiCloseFigure
9234b044 9222686a win32k!NtGdiClearBitmapAttributes
9234b048 922f103c win32k!NtGdiClearBrushAttributes
9234b050 922f352c win32k!NtGdiColorCorrectPalette
9234b054 921b3ca5 win32k!NtGdiCombineRgn
9234b058 9225ad3d win32k!NtGdiCombineTransform
```
Conversion Functions

- MapVirtualKey / MapVirtualKeyEx
- ToAscii / ToAsciiEx
- VkKeyScan / VkKeyScanEx
#27 GetKeyState / GetAsyncKeyState
GetKeyState / GetAsyncKeyState

- APIs for determining the state of a key at some point in time.
- Difference is:
  - GetKeyState is more specific and doesn’t reflect the interrupt-level state information,
  - GetAsyncKeyState reflects the interrupt-level state of keys.
- One of the most widely used techniques by keyloggers.
#28 GetKeyboardState
GetKeyboardState

- API for determining the state of a keyboard.
- Fills an array of virtual keys.
- One of the most widely used method used by keyloggers.
#29 Text Output APIs
Text Output APIs

- APIs used by applications to output text.
- Examples:
  - TextOut
  - ExtTextOut
  - DrawText / DrawTextEx
#30 GetWindowText
GetWindowText

- Can be used within an injected thread.
-Copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. However, GetWindowText cannot retrieve the text of a control in another application.
#31 WM_GETTEXT Message
WM_GETTEXT Message

- Can be used for retrieving another applications window content.

```c++
#define WM_GETTEXT 0x000D
```

**Parameters**

- **wParam**
  - The maximum number of characters to be copied, including ANSI applications may have the string in the buffer reduced from ANSI to Unicode.

- **lParam**
  - A pointer to the buffer that is to receive the text.
#32 SetWindowsHookEx
SetWindowHookEx

- Another term for saying “Keylogger” 😊
- Definitely the MOST WIDELY USED technique for keylogging!!!
- Nearly %95 of keyloggers use it 😊
Why?

- It is a way for providing callbacks to developers but widely used by malware authors.
- Have pretty much variations such as “Low Level Hook”, “Get Message Hook” and etc.
Hook Types

- **WH_CALLWNDPROC**: Installs a hook procedure that monitors messages before the system sends them to the destination window procedure.

- **WH_CALLWNDPROCRET**: Installs a hook procedure that monitors messages after they have been processed by the destination window procedure.

- **WH_CBT**: Installs a hook procedure that receives notifications useful to a Computer Based Training (CBT) application.

- **WH_DEBUG**: Installs a hook procedure useful for debugging other hook procedures.
Hook Types

- **WH_GETMESSAGE**: Installs a hook procedure that monitors messages posted to a message queue.

- **WH_JOURNALRECORD**: Installs a hook procedure that records input messages posted to the system message queue.

- **WH_KEYBOARD**: Installs a hook procedure that monitors keystroke messages.

- **WH_KEYBOARD_LL**: Installs a hook procedure that monitors low-level keyboard input events.
Low Level Hooks

- Starting from this slide
- What is a Hook Function?
- Only low level hooks are allowed in Raw Input Thread.
- Ability to block some input events using these hooks.
- Will be described separately.

- WH_CALLWNDPROC and WH_CALLWNDPROCRET
- WH_CBT
- WH_DEBUG
- WH_FOREGROUNDIDLE
- WH_GETMESSAGE
- WH_JOURNALPLAYBACK
- WH_JOURNALRECORD
- WH_KEYBOARD_LL
- WH_KEYBOARD
- WH_MOUSE_LL
- WH_MOUSE
- WH_MSGFILTER and WH_SYSMSGFILTER
- WH_SHELL
#33 DirectX Keylogger
DirectX

- Not widely used but a good way for logging keystrokes.
How?

- Pretty easy to implement with DirectInputCreateEx API.
- CreateDevice API is used for keyboard device creation.
#34 Browser Extensions
Browser Extensions

- Sneaky creatures!
- Not widely used but a great for bypassing security measures.
Inspecting

- XPI files are just zip files.
- Unzip it and analyze what it does.
Demos

- Go to Materials/Keyloggers folder:
  - Analyze martin.exe
  - Analyze AKLT_3.0.exe
  - Analyze refog_personal_manager.exe
  - Analyze Elite Keylogger
  - Analyze java keylogger
  - Analyze Free Keylogger
Thanks
Questions?