Neutralizing Keyloggers, an intimate story between the keyboard and the system

DAVID Baptiste, PhD

TROOPERS CONFERENCE IN 2022
TROOPERS
JUNE 27 TO
JULY 01, 2022

How to protect against keyloggers?

→ How does the keyboard system work in Windows?

→ How do keyloggers work?

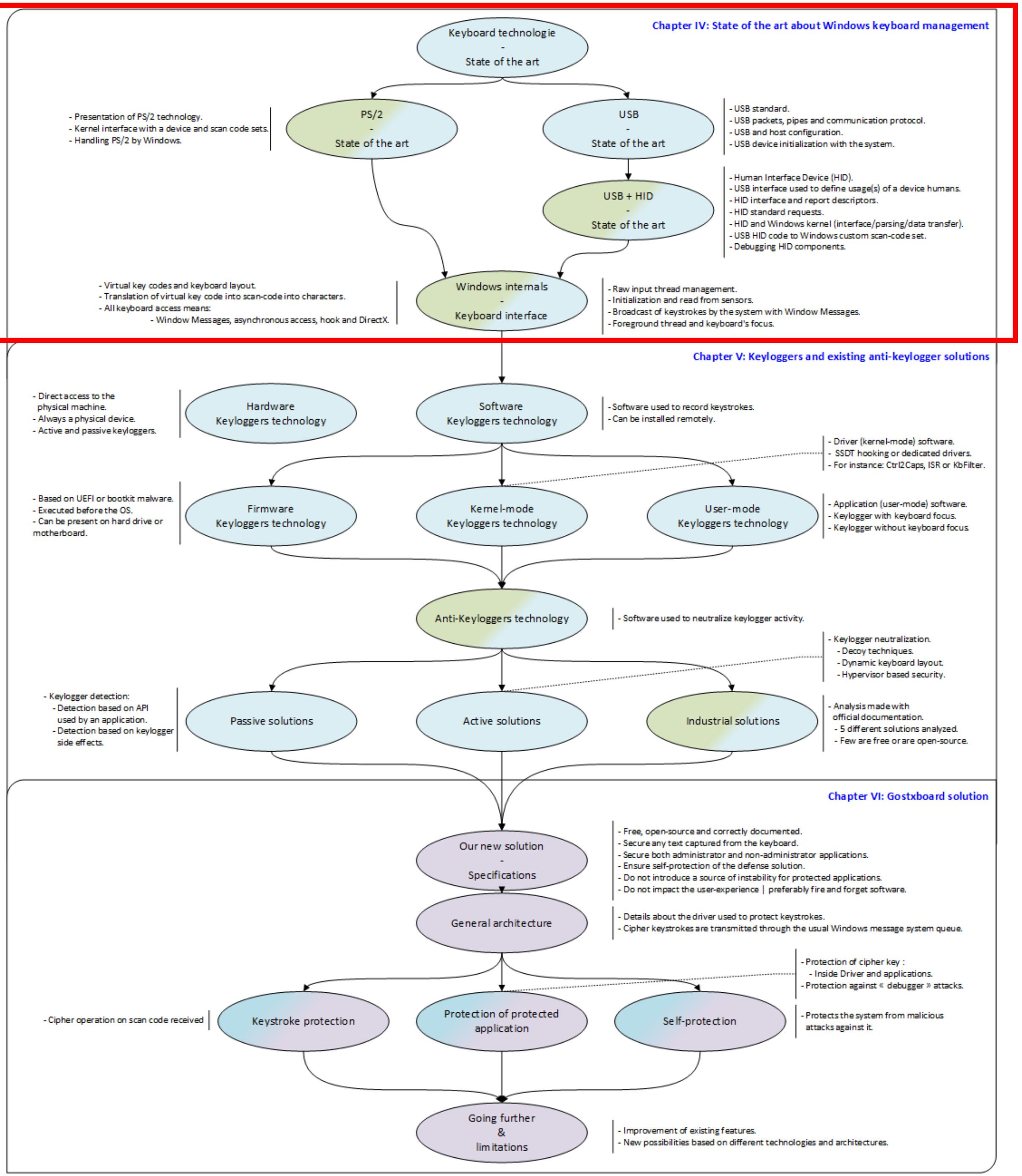
→ How do the existing anti-keylogger solutions work?

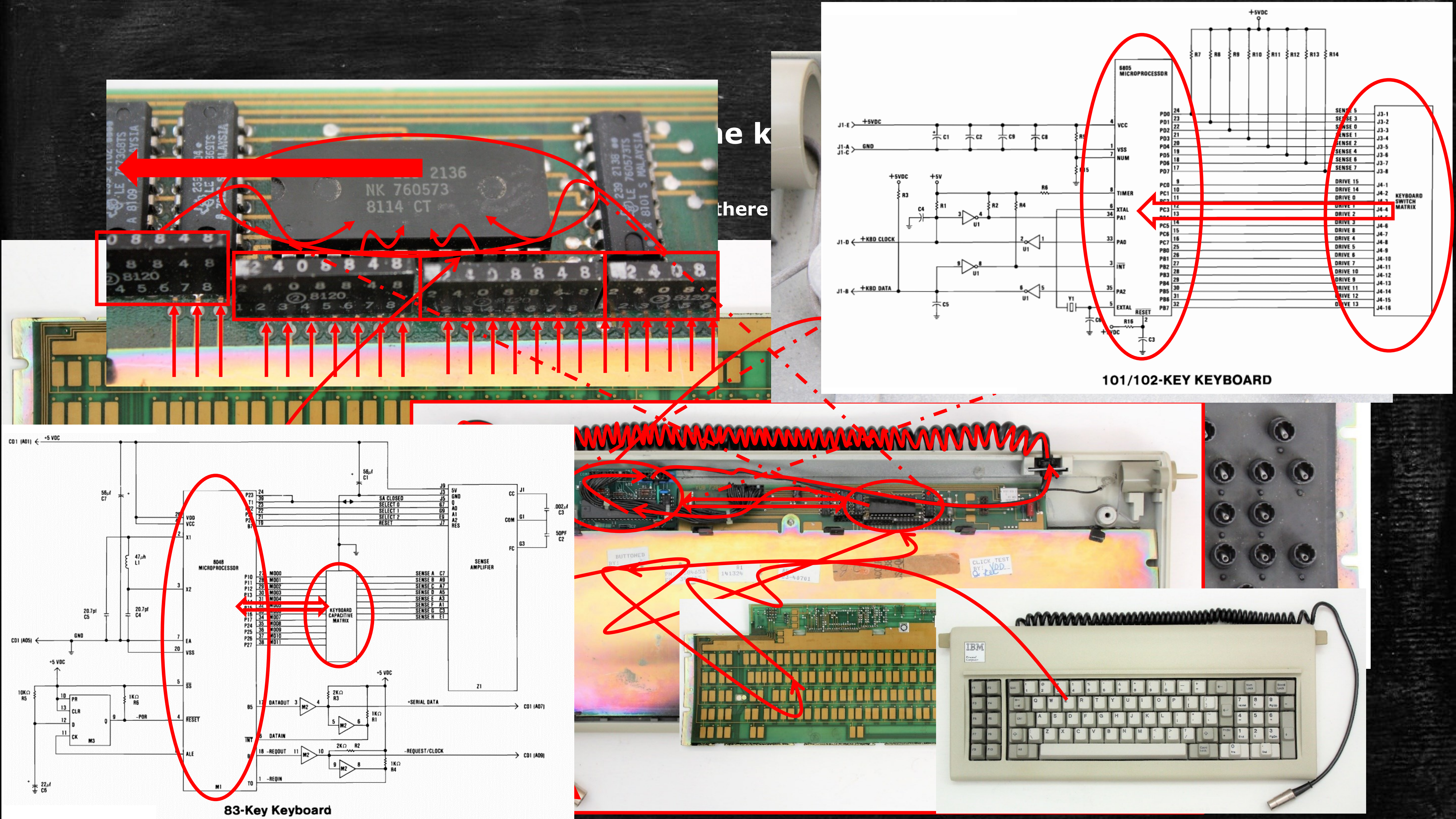
→ How can we do better?

We are talking about **protecting** the **integrity** of the **data** handled **within a system**.

So you wanna go to such a journey?

Protection against keylogger malware





2136
NK 760573
8114 CT
6805 MICROPROCESSOR
KEYBOARD SWITCH MATRIX
101/102-KEY KEYBOARD
8048 MICROPROCESSOR
KEYBOARD CAPACITIVE MATRIX
SENSE AMPLIFIER
83-Key Keyboard
CLICK TEST
IBM

Different codes used by different keyboards

→ The values present in a **scan-code** set are **hardware manufacturer defined**.
→ Historically speaking, there are three main scan-code sets.
→ Scan code set 1: used by IBM PC XT ~ 1983.
→ Scan code set 2: used by IBM PC AT ~ 1984.
→ Scan code set 3: used by IBM PC 3270 ~ 1987.

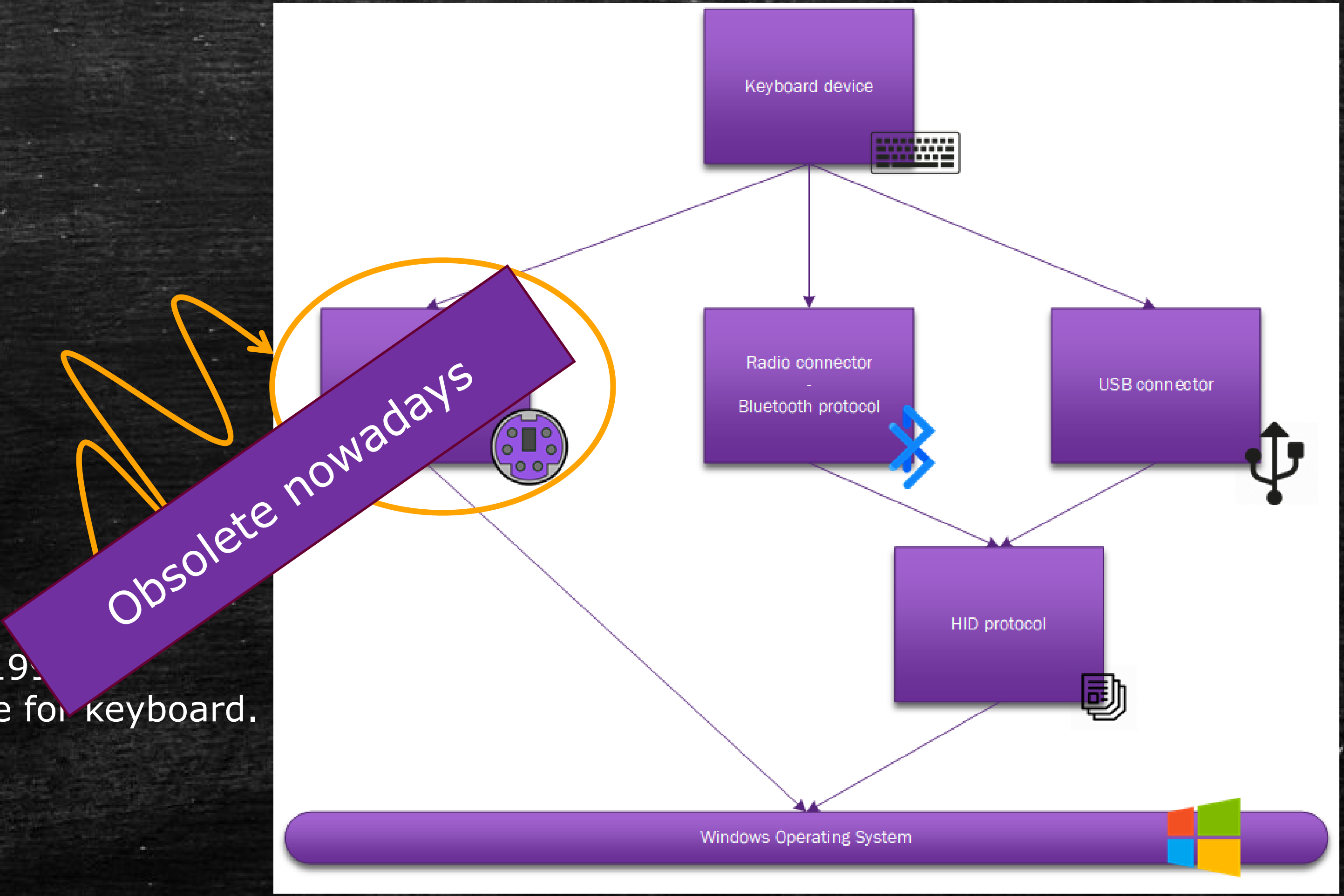
→ A different scan-code set could be used by a keyboard manufacturer.
→ But it would be its responsibility to translate it into a supported scan-code by the computer's operating system.

→ In practice, the scan-code set 1 is used the most by device manufacturers.
→ Still **supported** by operating systems due to **backward compatibility**.

IBM Key No.	Set 1 Make/Break	Set 2 Make/Break	Set 3 Make/Break	Base Case	Upper Case
1	29/A9	0E/F0 0E	0E/F0 0E	`	~
2	02/82	16/F0 16	16/F0 16	1	!
3	03/83	1E/F0 1E	1E/F0 1E	2	@
4	04/84	26/F0 26	26/F0 26	3	#
5	05/85	25/F0 25	25/F0 25	4	$
6	06/86	2E/F0 2E	2E/F0 2E	5	%
7	07/87	36/F0 36	36/F0 36	6	^
8	08/88	3D/F0 3D	3D/F0 3D	7	&
9	09/89	3E/F0 3E	3E/F0 3E	8	*
10	0A/8A	46/F0 46	46/F0 46	9	(
11	0B/8B	45/F0 45	45/F0 45	0	)
12	0C/8C	4E/F0 4E	4E/F0 4E	-	_
13	0D/8D	55/F0 55	55/F0 55	=	+
15	0E/8E	66/F0 66	66/F0 66	Backspace	
16	0F/8F	0D/F0 0D	0D/F0 0D	Tab	
17	10/90	15/F0 15	15/F0 15	q	Q
18	11/91	1D/F0 1D	1D/F0 1D	w	W
19	12/92	24/F0 24	24/F0 24	e	E
20	13/93	2D/F0 2D	2D/F0 2D	r	R
21	14/94	2C/F0 2C	2C/F0 2C	t	T
22	15/95	35/F0 35	35/F0 35	y	Y
23	16/96	3C/F0 3C	3C/F0 3C	u	U
24	17/97	43/F0 43	43/F0 43	i	I
90	45/C5	77/F0 77	76/F0 76	Num Lock	
91	47/C7	6C/F0 6C	6C/F0 6C	Keypad 7	
92	4B/CB	6B/F0 6B	6B/F0 6B	Keypad 4	
93	4F/CF	69/F0 69	69/F0 69	Keypad 1	
95	E0 35/E0 B5 (base)	E0 4A/E0 F0 4A (base)	77/F0 77	Keypad /	
96	48/C8	75/F0 75	75/F0 75	Keypad 8	
97	4C/CC	73/F0 73	73/F0 73	Keypad 5	
98	50/D0	72/F0 72	72/F0 72	Keypad 2	
99	52/D2	70/F0 70	70/F0 70	Keypad 0	
100	37/B7	7C/F0 7C	7E/F0 7E	Keypad *	
101	49/C9	7D/F0 7D	7D/F0 7D	Keypad 9	
102	4D/CD	74/F0 74	74/F0 74	Keypad 6	
103	51/D1	7A/F0 7A	7A/F0 7A	Keypad 3	
104	53/D3	71/F0 71	71/F0 71	Keypad .	
105	4A/CA	7B/F0 7B	84/F0 84	Keypad -	
106	4E/CE	79/F0 79	7C/F0 7C	Keypad +	
108	E0 1C/E0 9C	E0 5A/E0 F0 5A	79/F0 79	Keypad Enter	
110	01/81	76/F0 76	08/F0 08	Esc	
112	3B/BB	05/F0 05	07/F0 07	F1	
113	3C/BC	06/F0 06	0F/F0 0F	F2	
114	3D/BD	04/F0 04	17/F0 17	F3	
115	3E/BE	0C/F0 0C	1F/F0 1F	F4	
116	3F/BF	03/F0 03	27/F0 27	F5	
117	40/C0	0B/F0 0B	2F/F0 2F	F6	
118	41/C1	83/F0 83	37/F0 37	F7	
119	42/C2	0A/F0 0A	3F/F0 3F	F8	
120	43/C3	01/F0 01	47/F0 47	F9	
121	44/C4	09/F0 09	4F/F0 4F	F10	
122	57/D7	78/F0 78	56/F0 56	F11	
123	58/D8	07/F0 07	5E/F0 5E	F12	
124	E0 2A E0 37/E0 B7 E0 AA	E0 12 E0 7C/E0 F0 7C E0 F0 12	57/F0 57	Print Screen	
125	46/C6	7E/F0 7E	5F/F0 5F	Scroll Lock	
126	E1 1D 45/E1 9D C5	E1 14 77 E1/F0 14 F0 77	62/F0 62	Pause Break	
29 or 42*	2B/AB	5D/F0 5D	5C/F0 5C or 53/F0 53	\	-

Table 4.1: List of different scan codes from all different scan code sets — IBM PS/2 Model 50 and 60 Technical Reference.

Many technologies to interface with the keyboard

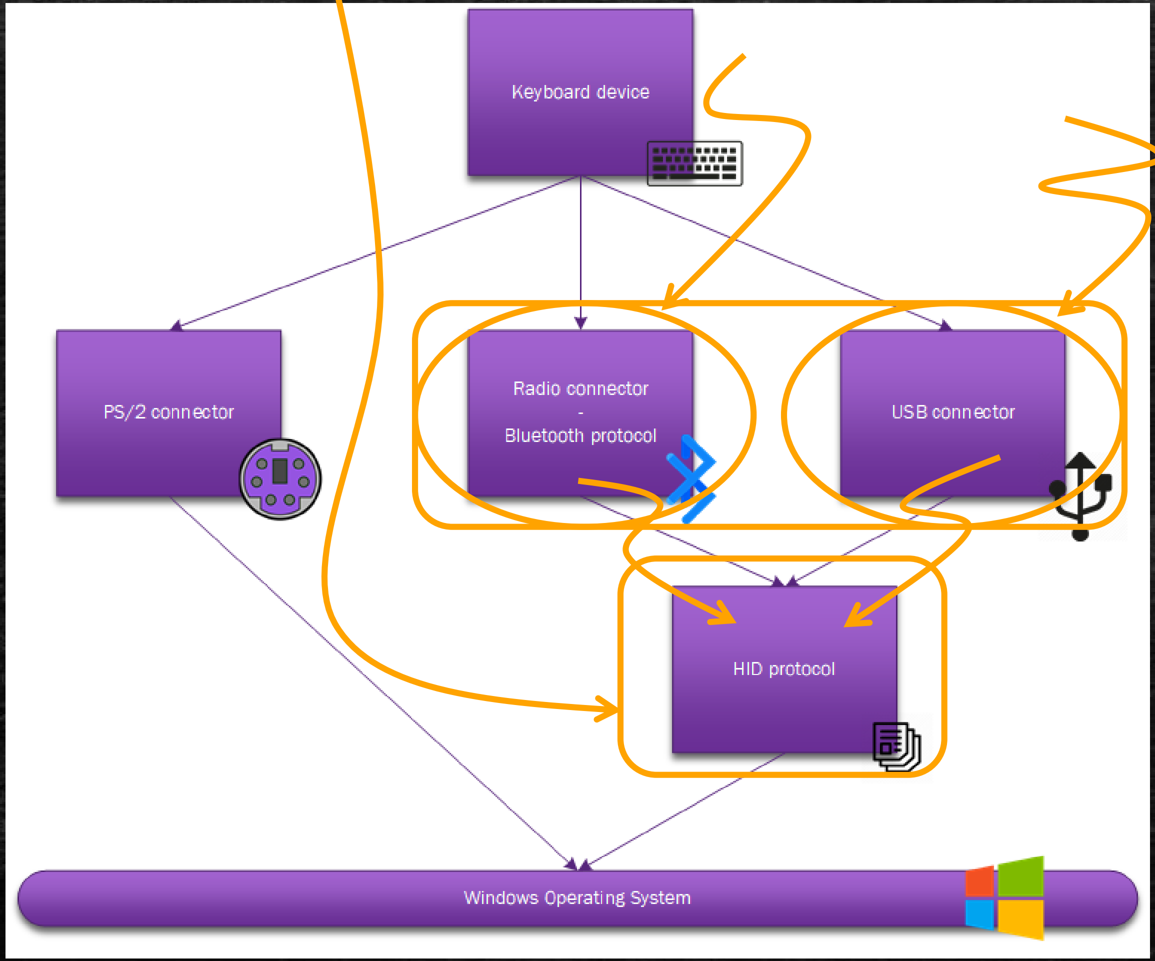


→ PS/2 raised in 1987.
→ Extended with ATX in 19
→ One for mouse and one for keyboard.

Many technologies to interface with the keyboard

→ HID stands for “Human Interface Device” class.
→ It references all devices interfacing with humans.
→ Device self-describing and manufacturer-defined interface to allow generic software applications.

→ Wireless connection with the machine.
→ Use Bluetooth protocol.
→ Quite similar to the USB protocol.



→ Use Universal Bus Serial protocol.
→ Each device has an “address” on the bus.
→ Use “Interrupt Data Transfers”.

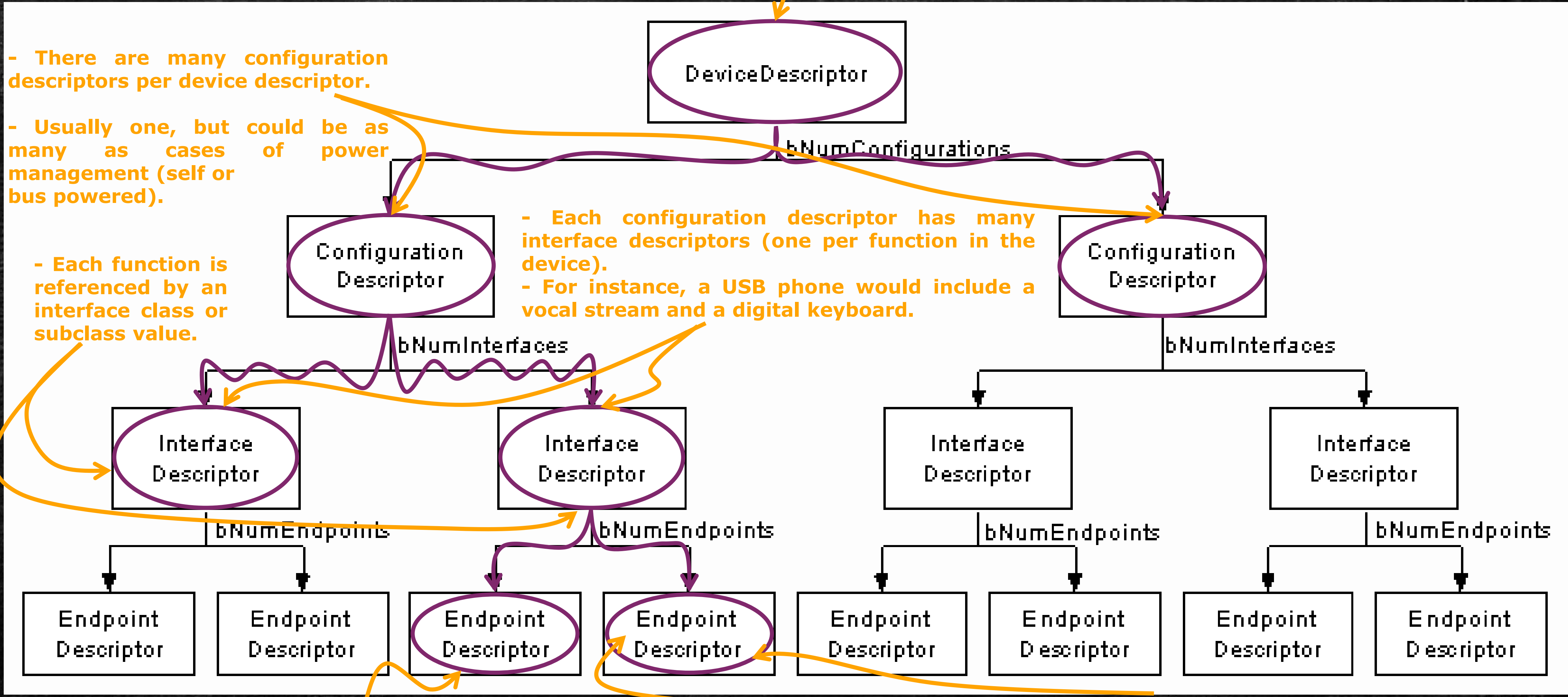
One per device, it provides USB version number, the type of device (class, subclass), vendor id and product id.

- There are many configuration descriptors per device descriptor.

- Usually one, but could be as many as cases of power management (self or bus powered).

- Each function is referenced by an interface class or subclass value.

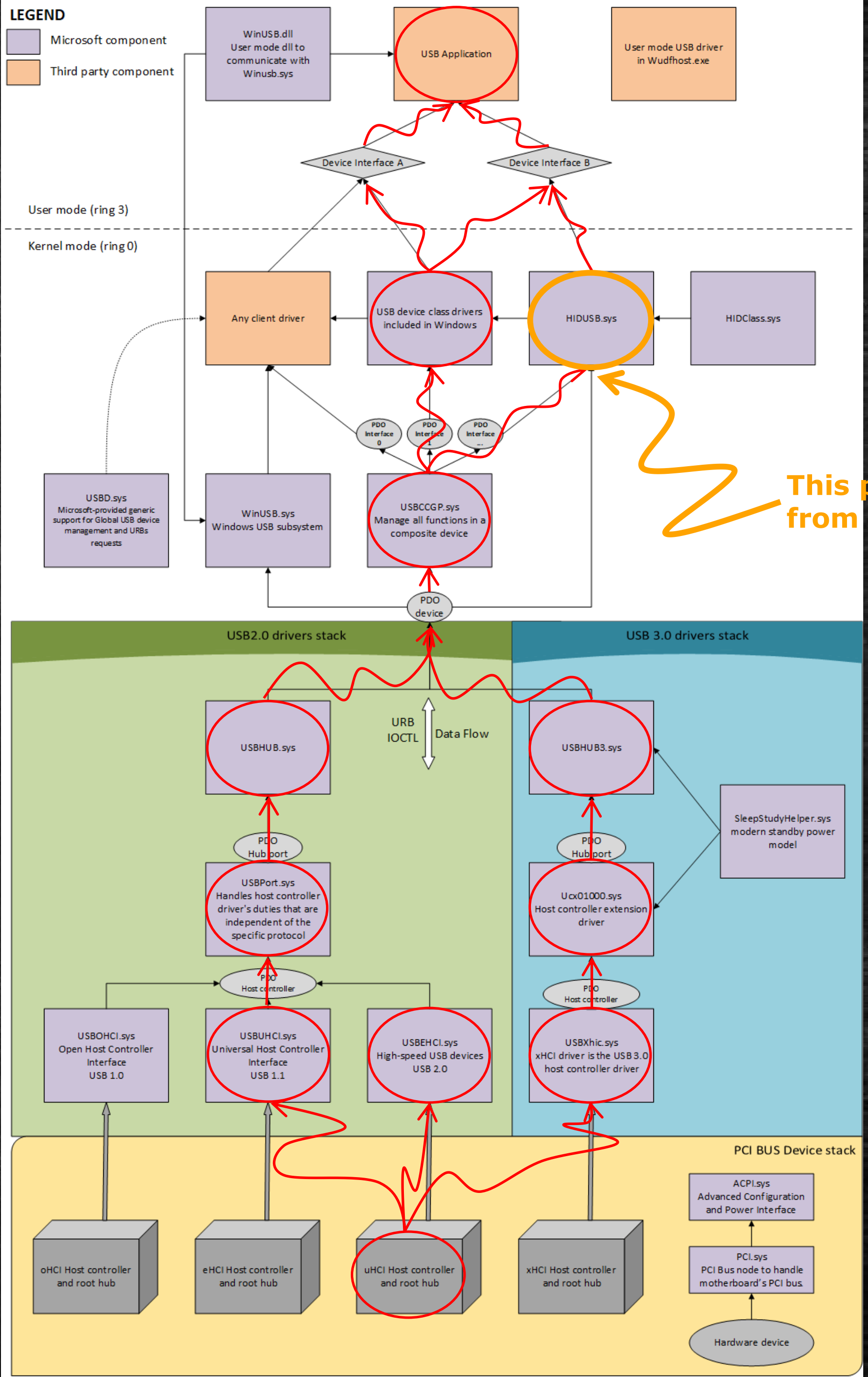
- Each configuration descriptor has many interface descriptors (one per function in the device).
- For instance, a USB phone would include a vocal stream and a digital keyboard.



- Each function can use more than one endpoint.

- Endpoint descriptors define how to communicate with a function.

- It provides bandwidth requirements, the direction (IN/OUT), transfer type and maximum packet size.



This point is relevant since it translate from USB to device drier (i.e.: keyboard).

USB/HID – Protocol description:

From **USB** to **HID**:

→ The HID interface corresponds to **Human Interface Device**.

→ HID is a special class defined through **USB interface descriptor** which allows a device to interface with humans easily.

→ HID is defined in USB interface descriptors to **define usage(s) (functions)** of a device.

- **Device self-describing interface** to allow **generic software applications**.
- One driver on the host to handle HID data whatever the HID device is.

Item Tag (Value)	Raw Data	Description
Usage Page (Generic Desktop)	05 01	
Usage (Keyboard)	09 06	
Collection (Application)	A1 01	
Usage Page (Keyboard/Keypad)	05 07	
Usage Minimum (Keyboard Left Control)	19 E0	
Usage Maximum (Keyboard Right GUI)	29 E7	
Logical Minimum (0)	15 00	
Logical Minimum (1)	25 01	
Report Size (1)	75 01	
Report Count (8)	95 08	
Input (Data,Var,Abs,NWrp,Lin,Pref,NNul,Bit)	81 02	(1) One byte to define modifier keys (1 bit per key)
Report Count (1)	95 01	
[illegible]	[illegible]	
Report Size (8)	75 08	
Logical Minimum (0)	15 00	
Logical Maximum (164)	26 A4 00	
Usage Page (Keyboard/Keypad)	05 07	
Usage Minimum (Undefined)	19 00	
Usage Maximum (Keyboard ExSel)	29 A4	
Input (Data,Ary,Abs)	81 00	(5) 6 bytes to buffer the current (most common) keystrokes
End Collection		

Table 4.12: Interface 0 HID Report Descriptor Keyboard.

```
typedef struct _HID_REPORT_INTERFACE_0_INPUT {
    unsigned char ModifierKeys;
    unsigned char Reserved;
    unsigned char Keystrokes[6];
} HID_REPORT_INTERFACE_0_INPUT;
```

Code 4.4: HID report interface 0

```
typedef struct _HID_REPORT_INTERFACE_0_OUTPUT
    {
    unsigned char LED_NUM_LOCK : 1;
    unsigned char LED_CAPS_LOCK : 1;
    unsigned char LED_SCROLL LOCK : 1;
    unsigned char LED_COMPOSE : 1;
    unsigned char LED_KANA : 1;
    unsigned char Reserved : 3;
} HID_REPORT_INTERFACE_0_OUTPUT;
```

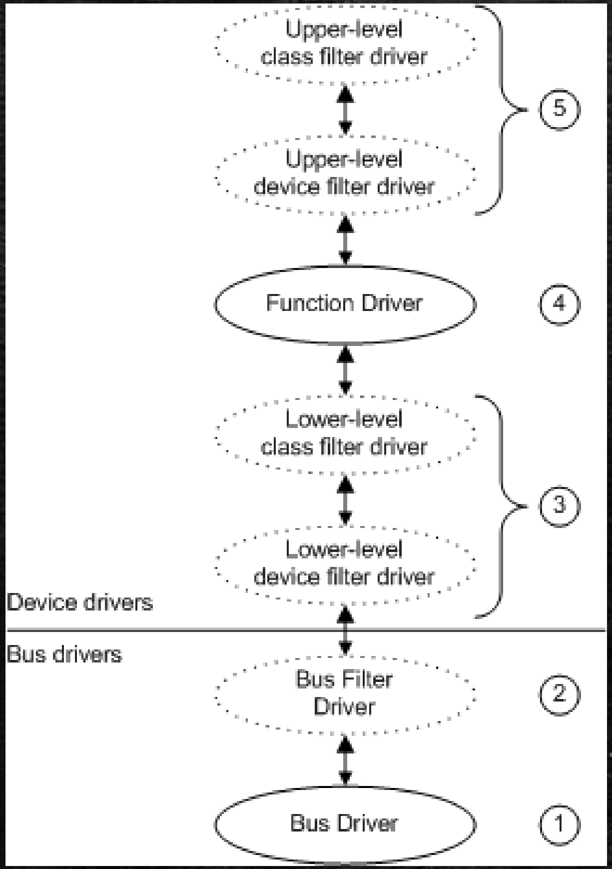
Code 4.5: Hid report interface 1

→).

→ USB

co

HID - Windows' kernel architecture



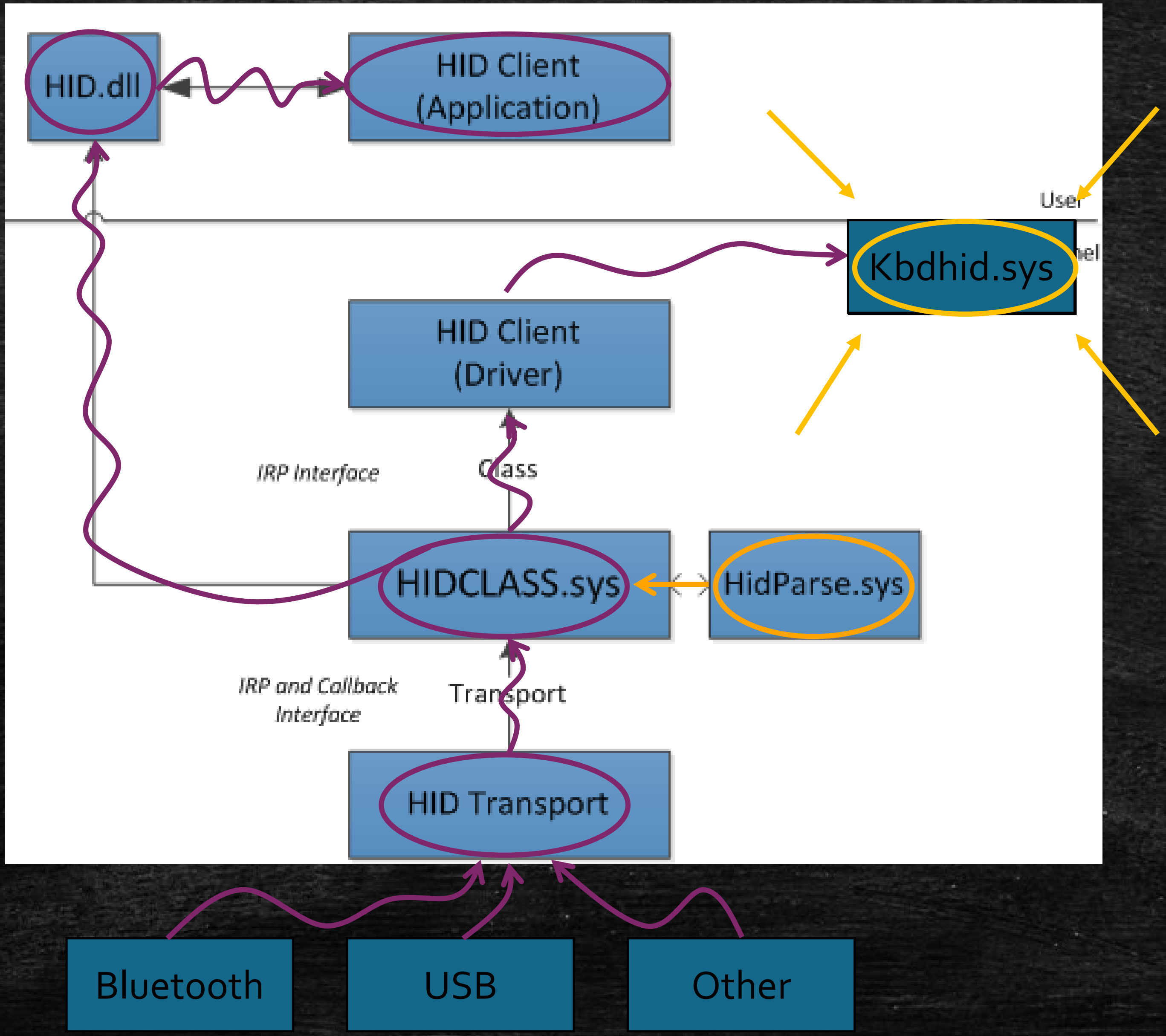
→ All **requests** from **devices** that **interface** with **HID** are redirected to the **HIDClass.sys driver**.

- The request can come from **Bluetooth**, **USB**, … **devices**.
- It is possible to **register** a **HID driver** to **filter HID reports**.
- Windows API allows to write **HID Client driver** (third party drivers) in a simplified way

→ In practice, the **freedom** of HID devices with report descriptors is **not as wide** as thought.

- **Windows** must be **able** to **adapt itself** to this freedom.
- HID driver should be reserved for specific circumstances.
  - Already a lot of supported HID Clients…

→ It is possible to restrict access to a HID devices only to “system” privileged processes.


HID.dll
HID Client
(Application)
User
Kbdhid.sys
HID Client
(Driver)
IRP Interface
Class
HIDCLASS.sys
HidParse.sys
IRP and Callback
Interface
Transport
HID Transport
Bluetooth
USB
Other

Keyboard management

→ The sy

e).

→ Micro

can code set 1.

e **past**!).

de set 1.

```
.data:00000001C0008190 ; __int64 HidP_KeyboardToScanCodeTable
.data:00000001C0008190 HidP_KeyboardToScanCodeTable dd 0FFh
.data:00000001C0008190        HID USB Code    Scan code set 1
.data:00000001C0008194                001h    dd 0FFh
.data:00000001C0008198                002h    dd 0FFh
.data:00000001C000819C                003h    dd 0FFh
.data:00000001C00081A0                004h    dd 1Eh
.data:00000001C00081A4                005h    dd 30h
.data:00000001C00081A8                006h    dd 2Eh
.data:00000001C00081AC                007h    dd 20h
.data:00000001C00081B0                008h    dd 12h
.data:00000001C00081B4                009h    dd 21h
.data:00000001C00081B8                00ah    dd 22h
.data:00000001C00081BC                00bh    dd 23h
.data:00000001C00081C0                00ch    dd 17h
.data:00000001C00081C4                00dh    dd 24h
.data:00000001C00081C8                00fh    dd 25h
.data:00000001C00081CC                010h    dd 26h
.data:00000001C00081D0                011h    dd 32h
.data:00000001C00081D4                012h    dd 31h
.data:00000001C00081D8                013h    dd 18h
.data:00000001C00081DC                014h    dd 19h
```

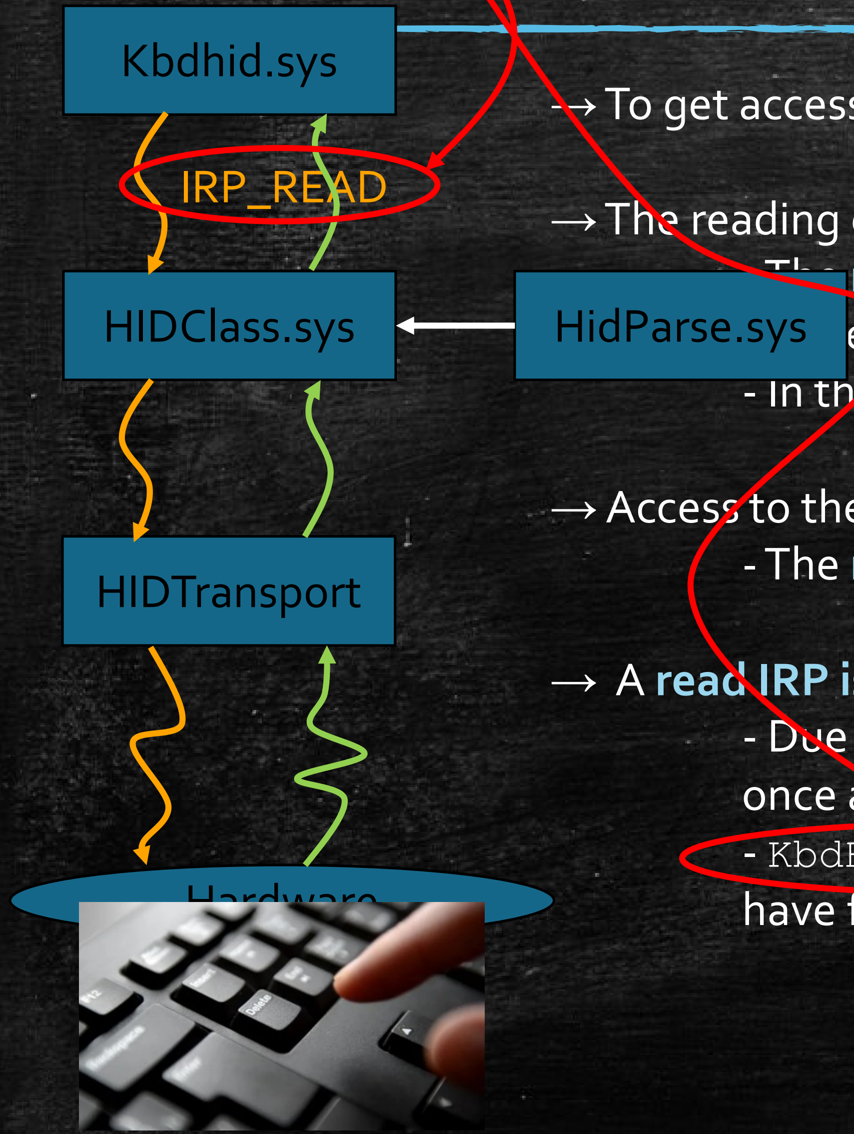
Figure 4.53: Beginning of the content of HidP_KeyboardToScanCodeTable.

read.

essed is repeated.

→ For th the translation with tables of corresponding values (as a chart).

Keyboard management



→ To get access to the keystroke, HID driver must **read from the device keyboard**.

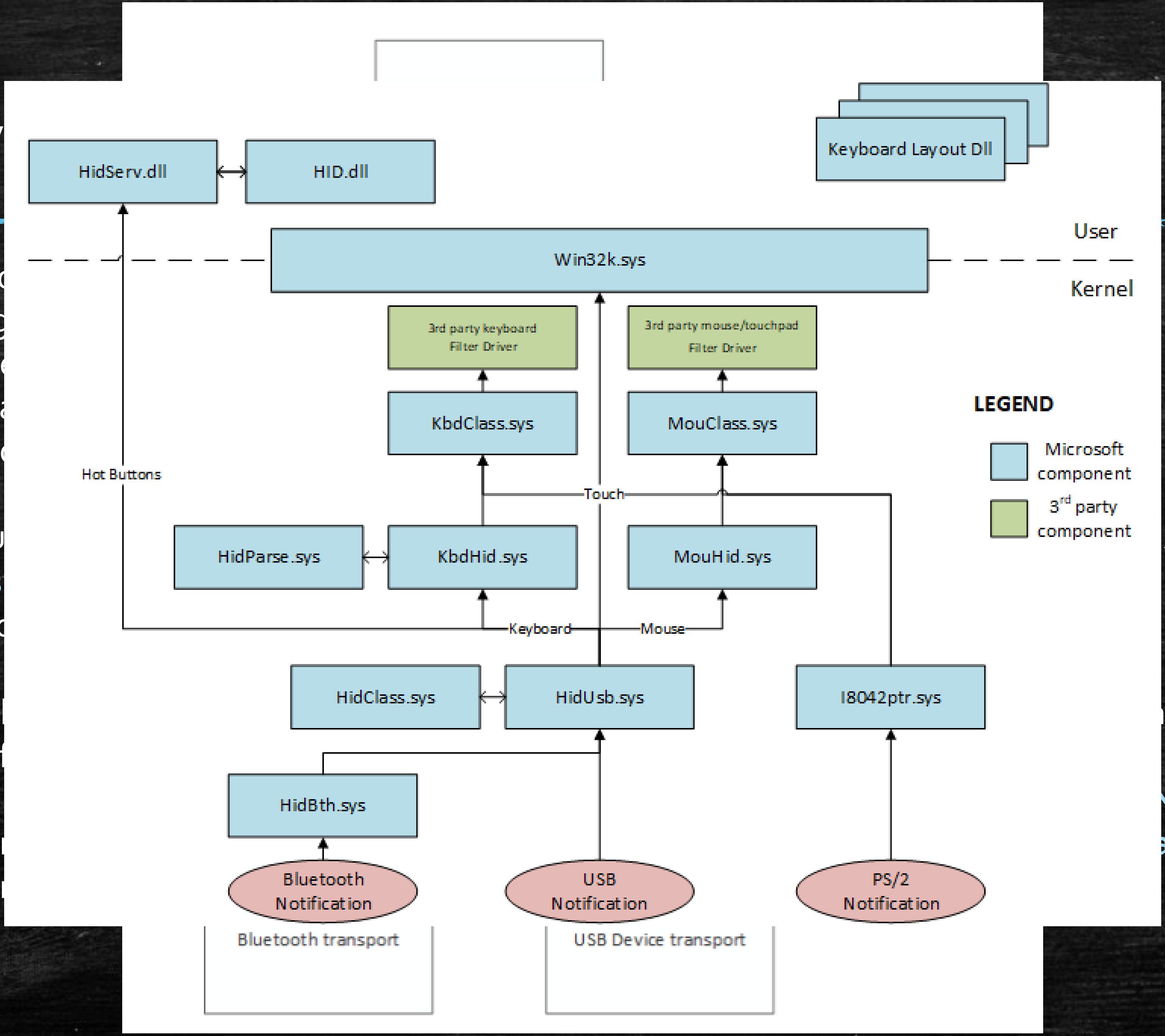
→ The reading operation is engaged by the driver which waits until a key is pressed.
- The **reading order goes down** to the device (such order is called an IRP read).
- eans the reading order comes from an “upper” driver.
- In the case of HID keyboards, this driver is “**kbdhid.sys**” driver.

→ Access to the **key code** is only possible **once the reading operation** has been **completed**.
- The **reading order** is **sent back** to the driver.

→ A **read IRP is always pending** in the system to always read keystrokes.
- Due to `KbdHid_InitiateStartRead` routine which (re)-engages the reading IRP once a read operation has succeeded.
- `KbdHid_ReadComplete` routine is called once all underlying drivers in the device stack have finished to process the IRP.
  - This routine gets access to the keystroke scan code!

Key

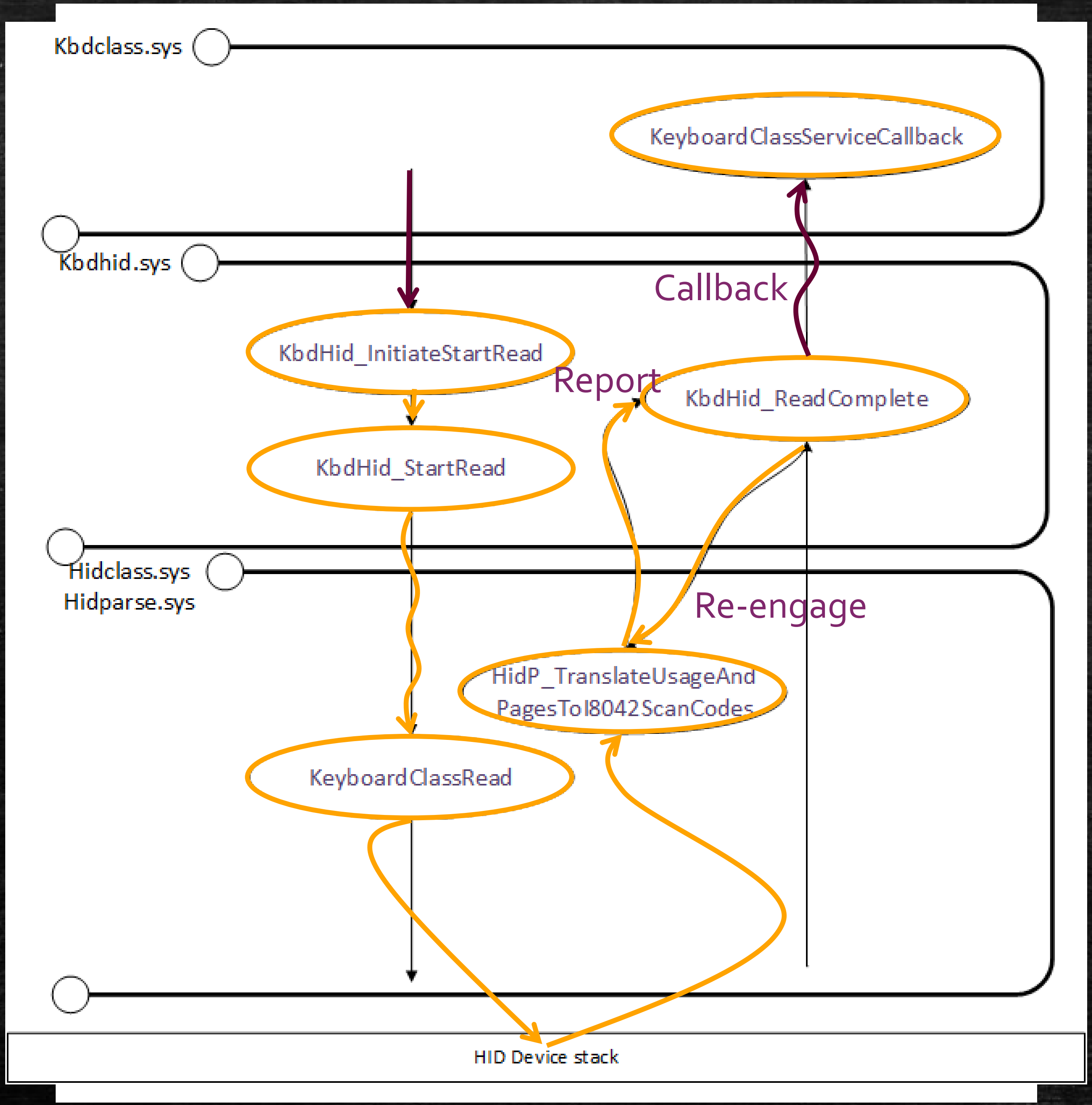
→ HID keyboard
- It is g
- There
- In pra
- This



→ **kbdhid.sys** u ... llback.
→ **i8042prt.sys**
- It sho

→ This also expl ... n PS/2, is translated into scan code set 1 f

→ The callback ... NAL_KEYBOARD_CONNECT.
- This ... ack.
- This


Kbdclass.sys
KeyboardClassServiceCallback
Kbdhid.sys
Callback
KbdHid_InitiateStartRead
Report
KbdHid_ReadComplete
KbdHid_StartRead
Hidclass.sys
Hidparse.sys
Re-engage
HidP_TranslateUsageAnd
PagesToI8042ScanCodes
KeyboardClassRead
HID Device stack

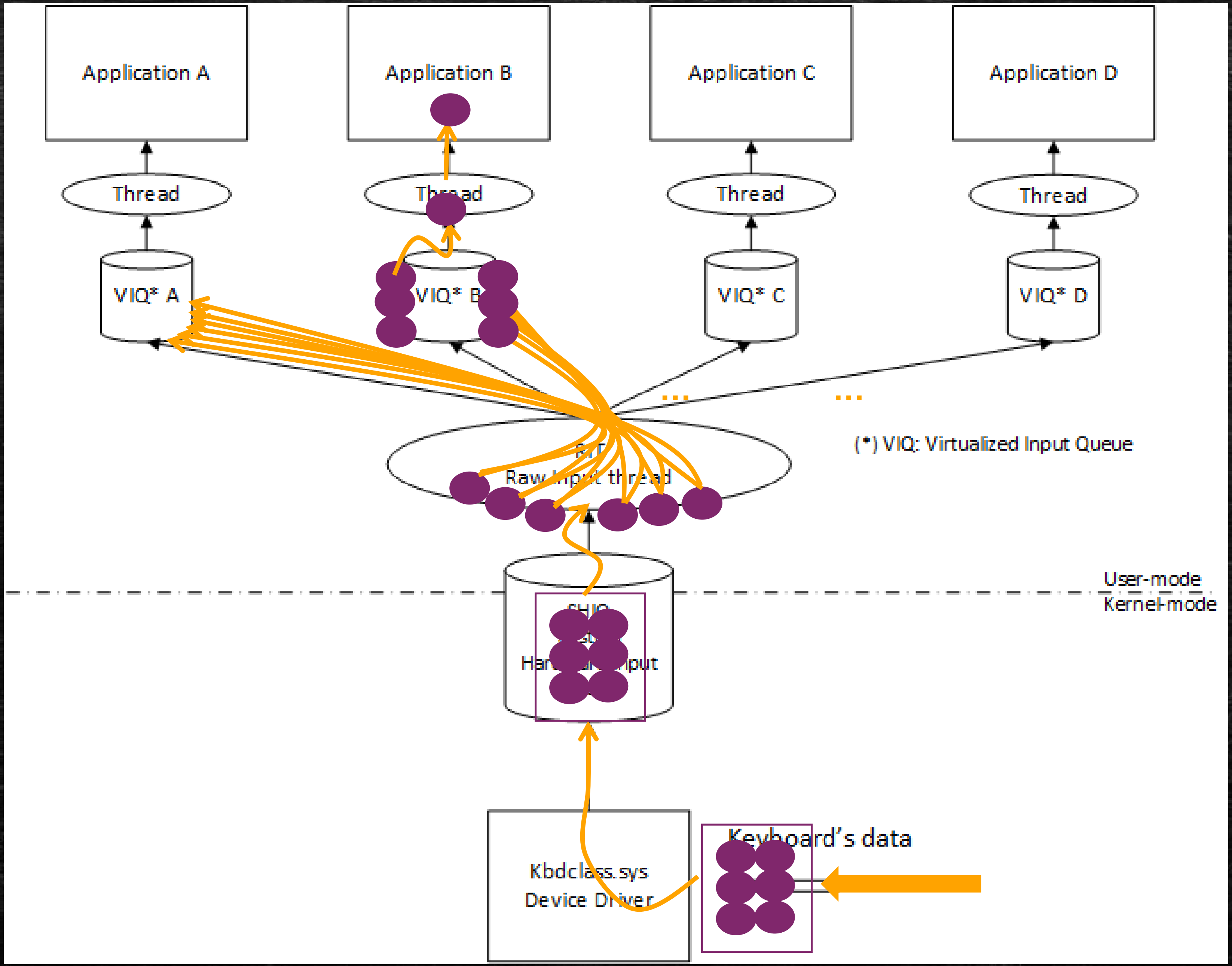
Kbdclass and Windows subsystem

→ **Transition** from **kernel** to **user mode** architecture:
- The goal is to transfer keyboard information from kbdclass.sys to user-mode applications.

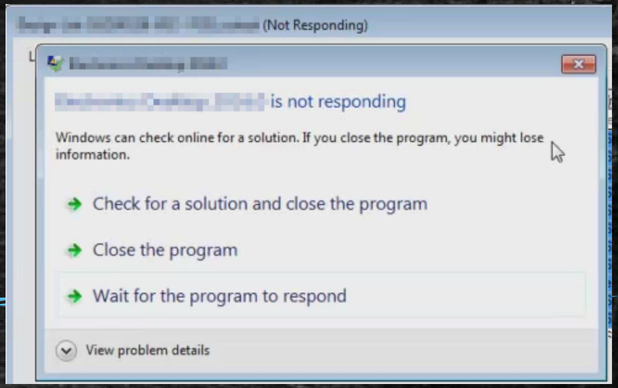
→ Done via the **Raw Input Thread** (RIT) with a **message system** used by GUI processes.
- The System of messages used for GUI windows is asynchronous nowadays.
- The **RIT** is a **kernel-mode thread** hosted by **csrss.exe**.

RIT is a centralized system that manages keystrokes to distribute them through messages in an asynchronous way.

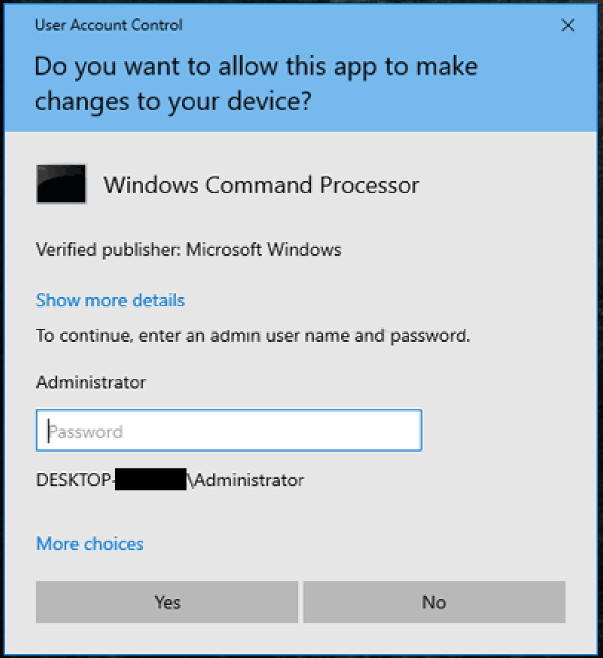
→ **RIT initializes** the **read IRP** processed by kbdclass.sys driver.


Application A
Application B
Application C
Application D
Thread
Thread
Thread
Thread
VIQ* A
VIQ* B
VIQ* C
VIQ* D
(*) VIQ: Virtualized Input Queue
Raw Input thread
User-mode
Kernel-mode
Kbdclass.sys
Device Driver
Keyboard's data

Keyboard management



→ Few **details** about the **Raw Input Thread**:
- The RIT is managing initialization and **hot-keys** (**keyboard shortcuts**) registration.
- The RIT is able to adapt to **tablets** that often have **extra buttons**.
- The RIT **manages applications** that **hang on the screen**.
- The RIT **avoids reading** from the keyboard device if this one is in **sleep mode**.
- The RIT is involved in the **clipboard management** procedure.



→ One **RIT** is initialized per **session** (from session 0 or other ones).
- There are **different desktops** (in a **Windows Station**) in a given **user's session**.
- RIT is able to manage desktop **switching** in a session.
  - For instance, User Account Control interface purposes with CTRL+ALT+DEL.
  - This keystroke combination is held by Winlogon only at boot time for its exclusive use.
- **Desktops** provide **a security** whenever switching by:
  - Message **system isolation** & **reset keyboard state**.

```
NTSTATUS __fastcall RIMStartDeviceRead(PVOID ApcContext, __int64 a2, void *Buffer, ULONG Length)
{
  _DWORD *ApcContext_1; // rbx@1
  NTSTATUS result; // eax@1

  ApcContext_1 = ApcContext;
  result = ZwReadFile(
             *((HANDLE *)ApcContext + 28),
             0i64,
             (PIO_APC_ROUTINE)rimInputApc,
             ApcContext,
             (PIO_STATUS_BLOCK)ApcContext + 16,
             Buffer,
             Length,
             &gZero,
             0i64);
  ApcContext_1[0x44] = result;
  if ( result >= 0 )
  {                                               // Gl
    *((_QWORD *)ApcContext_1 + 0x10F) = MEMORY[0xFFFF
    result = ApcContext_1[0x44];
  }
  return result;
}
```

→ Routine `rimInputApc` is
- It reengages the
- Thus, **after each**

→ The completion procedu
- i.e. once keystro
- Routines `rimPro`
kernel-mode to u

Syntax

C++ Copy

```
NTSYSAPI NTSTATUS ZwReadFile(
  [in]           HANDLE           FileHandle,
  [in, optional] HANDLE           Event,
  [in, optional] PIO_APC_ROUTINE  ApcRoutine,
  [in, optional] PVOID            ApcContext,
  [out]          PIO_STATUS_BLOCK IoStatusBlock,
  [out]          PVOID            Buffer,
  [in]           ULONG            Length,
  [in, optional] PLARGE_INTEGER   ByteOffset,
  [in, optional] PULONG           Key
);
```

Parameters

`[in] FileHandle`

Handle to the file object. This handle is created by a successful call to ZwCreateFile or ZwOpenFile.

`[in, optional] Event`

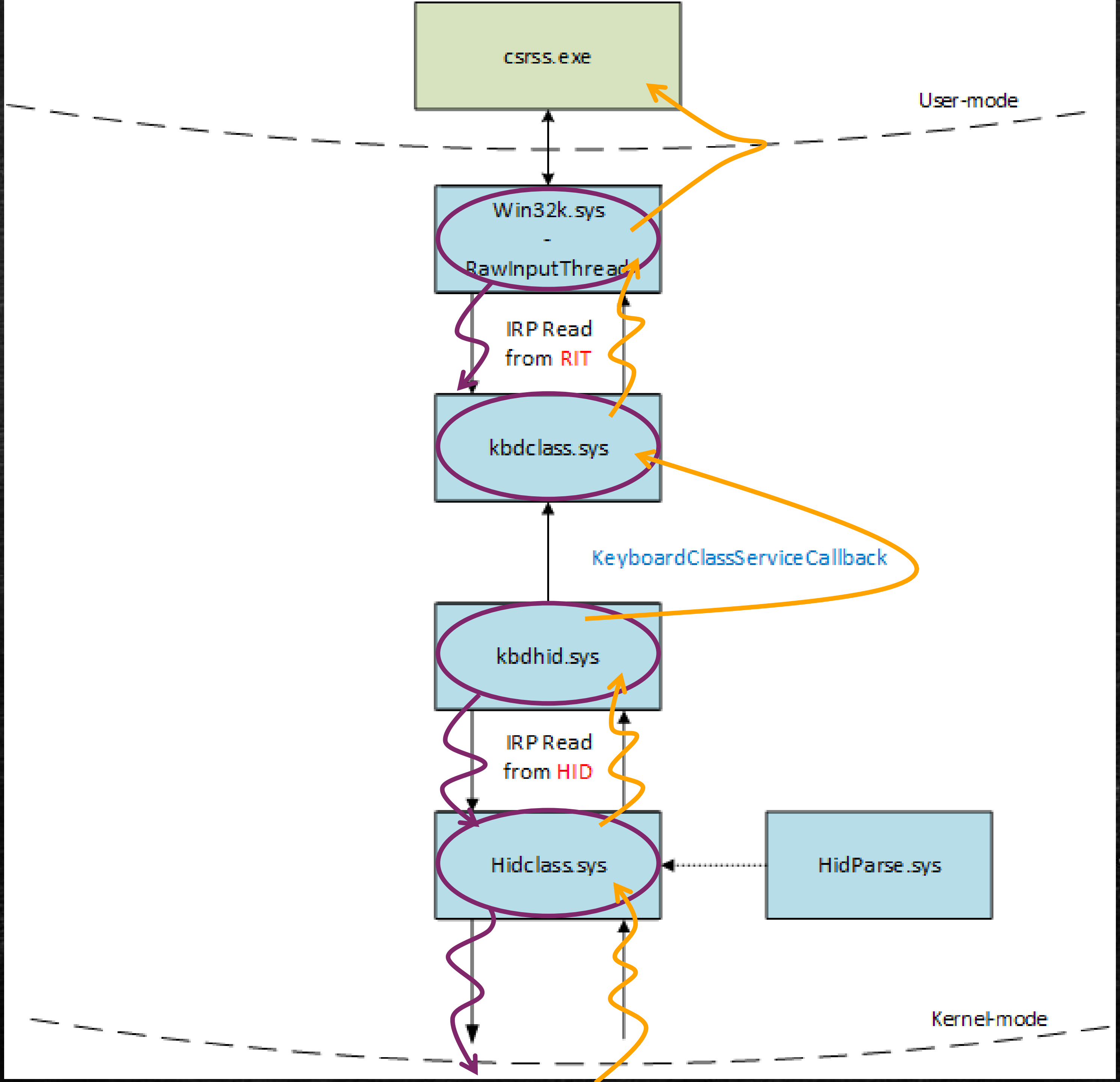
Optionally, a handle to an event object to set to the signaled state after the read operation completes. Device and intermediate drivers should set this parameter to **NULL**.

`[in, optional] ApcRoutine`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.

`[in, optional] ApcContext`

This parameter is reserved. Device and intermediate drivers should set this pointer to **NULL**.



csrss.exe
User-mode
Win32k.sys
-
RawInputThread
IRP Read
from RIT
kbdclass.sys
KeyboardClassServiceCallback
kbdhid.sys
IRP Read
from HID
Hidclass.sys
HidParse.sys
Kernel-mode

Management of keystroke content

→ When a key is received, it is received as a scan-code which is manufacturer defined.

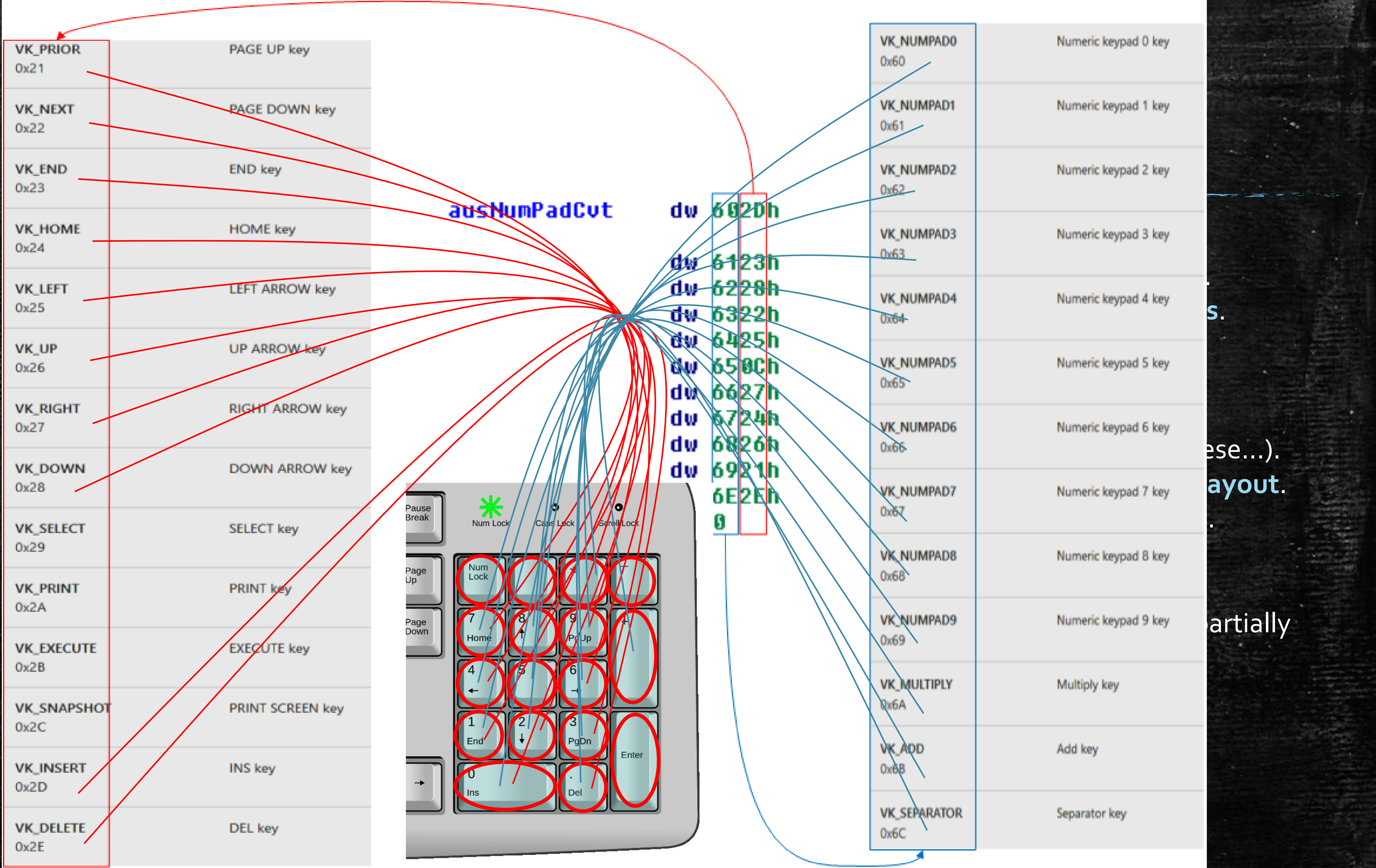
- To ease application development, Windows uses a **universal key representation code**.
- This code is called **virtual key code** (**VKC**).
- Conversion from scan-code to VKC is done in `CKeyboardSensor::ProcessInput` routine.

→ The **conversion** is a **two-steps procedure** acting as a **post-processing** on the **data** from devices.

- 1) It **normalizes** the scan code received (with `MapScancode`).
- 2) The **scan-code** is converted into a **virtual key code** (with `VKFromVSC` and `InternalMapVirtualKeyEx` routines).

→ If one of the **operations fails**, it means that the **key is invalid**.

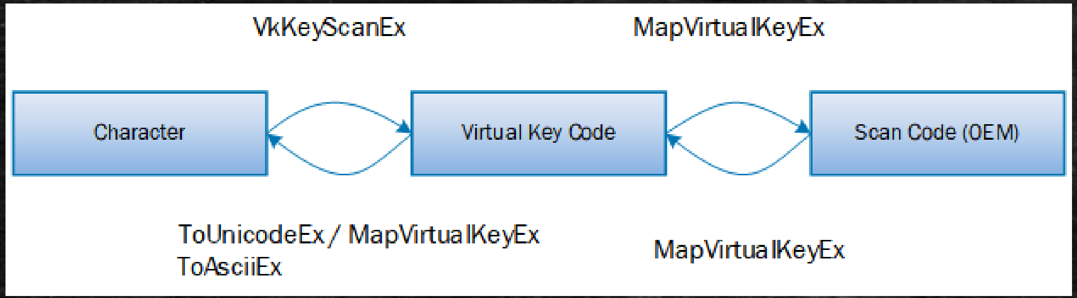
- In such a case, the **keystroke** is **dropped** and **ignored** by the system.


VK_PRIOR 0x21 PAGE UP key
VK_NEXT 0x22 PAGE DOWN key
VK_END 0x23 END key
VK_HOME 0x24 HOME key
VK_LEFT 0x25 LEFT ARROW key
VK_UP 0x26 UP ARROW key
VK_RIGHT 0x27 RIGHT ARROW key
VK_DOWN 0x28 DOWN ARROW key
VK_SELECT 0x29 SELECT key
VK_PRINT 0x2A PRINT key
VK_EXECUTE 0x2B EXECUTE key
VK_SNAPSHOT 0x2C PRINT SCREEN key
VK_INSERT 0x2D INS key
VK_DELETE 0x2E DEL key
ausNumPadCvt dw 602Dh
dw 6123h
dw 6228h
dw 6322h
dw 6425h
dw 650Ch
dw 6627h
dw 6724h
dw 6826h
dw 6921h
6E2Eh
0
Pause Break
Num Lock
Caps Lock
Scroll Lock
Page Up
Page Down
Num Lock
7 Home
8
9 PgUp
4
5
6
1 End
2
3 PgDn
0 Ins
. Del
Enter
VK_NUMPAD0 0x60 Numeric keypad 0 key
VK_NUMPAD1 0x61 Numeric keypad 1 key
VK_NUMPAD2 0x62 Numeric keypad 2 key
VK_NUMPAD3 0x63 Numeric keypad 3 key
VK_NUMPAD4 0x64 Numeric keypad 4 key
VK_NUMPAD5 0x65 Numeric keypad 5 key
VK_NUMPAD6 0x66 Numeric keypad 6 key
VK_NUMPAD7 0x67 Numeric keypad 7 key
VK_NUMPAD8 0x68 Numeric keypad 8 key
VK_NUMPAD9 0x69 Numeric keypad 9 key
VK_MULTIPLY 0x6A Multiply key
VK_ADD 0x6B Add key
VK_SEPARATOR 0x6C Separator key
s.
ese...).
ayout.
artially

Management of keystroke content

→ How is the translation from scan code to virtual key code and character done?

- The translation can be done in either direction.
- At **application level**, there are three ("four") main functions:
  - `MapVirtualKey(Ex)`, `VkKeyScan(Ex)`, and `ToUnicode(Ex)` / `ToAsciiEx`.
- In practice, most of the work is performed in `InternalMapVirtualKeyEx` routine.



→ Conversion from **virtual key code** to **character** can produce both **ASCII** (`ToAsciiEx`) or **Unicode** (`ToUnicodeEx`) characters.

→ In practice, the **translation is automatically** performed by the **RIT** in the **input messages** delivered to any application.

How do we access keyboard from application?

→ There are two main interfaces to access keyboard content.
- The **synchronous message** system provided by the Raw Input Thread.
- The **asynchronous system** by other means.

→ While the **message system** is the **backbone** of keystroke transmission but **there are other ways**.
- In practice, these are alternative ways of accessing various resources maintained by the RIT.
- This does not question the **central position** of the **raw input thread**.

→ More directly, all these **methods** are **used** by **legitimate applications** and ... **malicious** ones.
- Keyloggers are just **applications** that make **malicious use** of the input keyboard data.
- But they use the **same means of action** as **legitimate applications** (they have no choice).

Broadcast of keystrokes by the system with Window Messages

→ Let us practice a **simple experiment**:

- If **we press any key**, **only** the **application** displayed in the **foreground** of the **screen receives** the input content.
- **Applications** in the **background** receive **nothing** → There is a "*distribution **privilege***".

→ A **foreground thread** is the **default thread** created by the system when a **GUI window** is created.

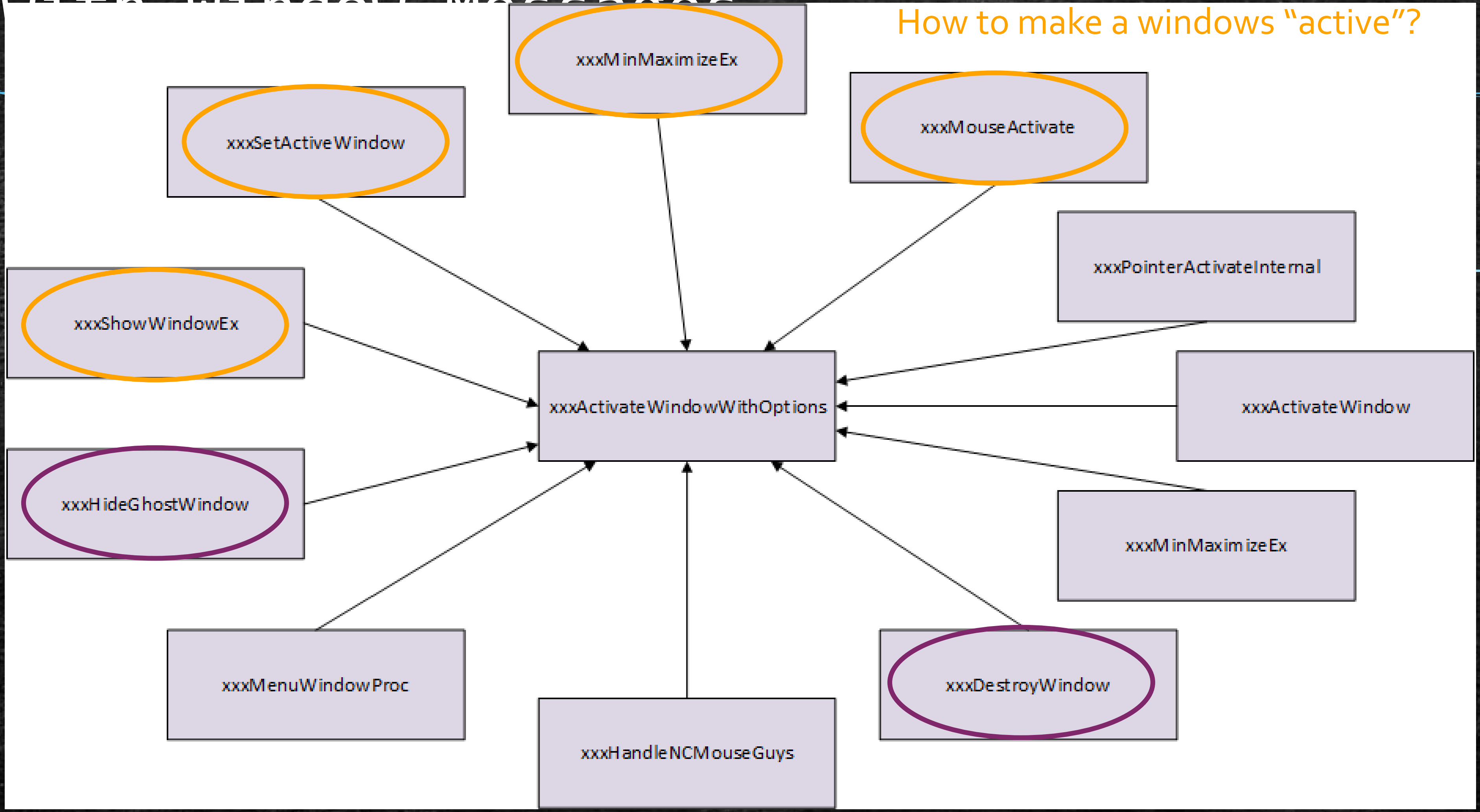
- It **owns** the **window** and its **associated message queue**.
- It **deals with messages** while its **associated window is foreground** on the screen.

→ A **window** that is in the **foreground** and **active** for the user is said to have **focus property**.

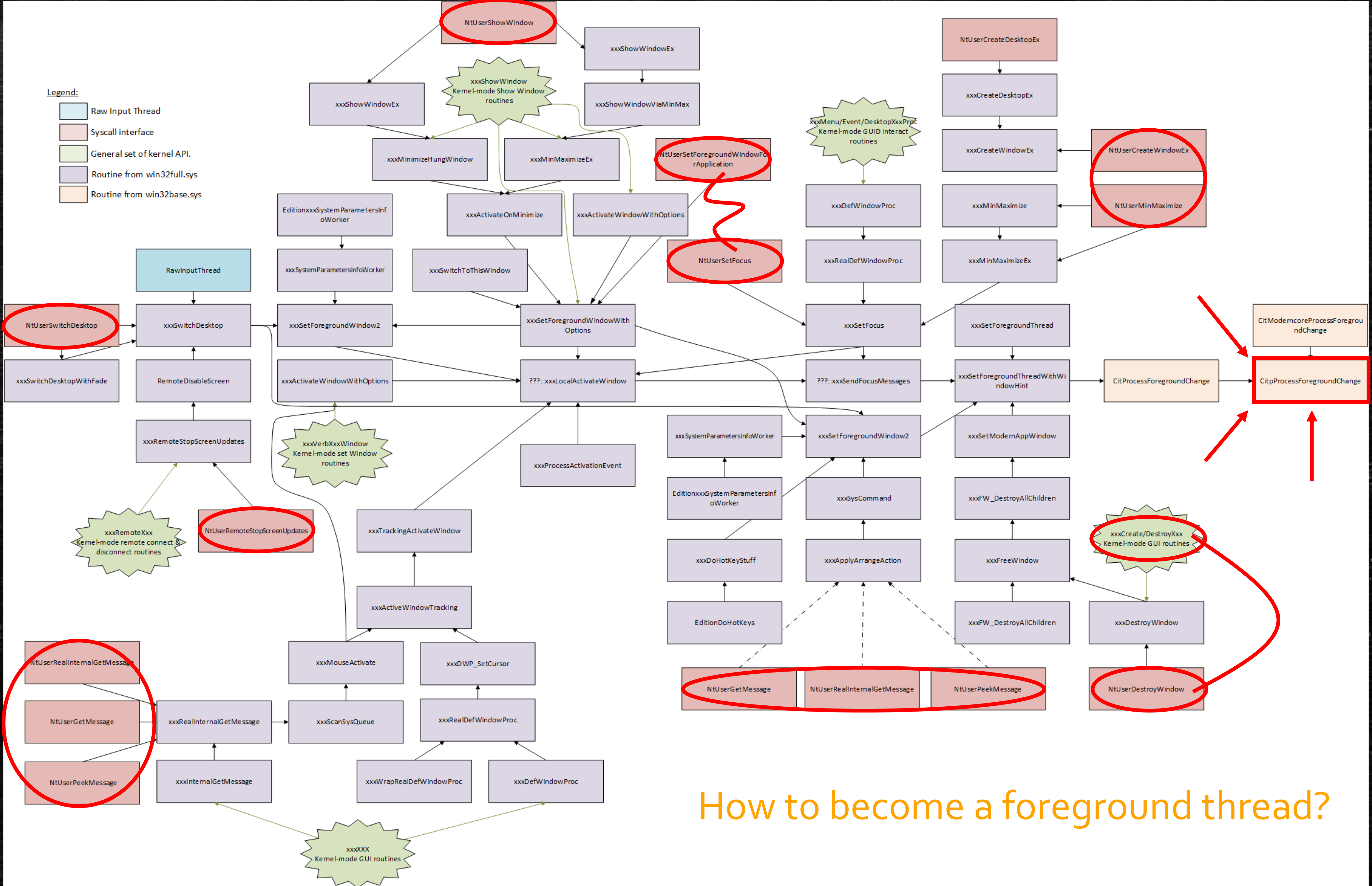
For short:

→ **Foreground thread** belongs to a **single thread** at time (and is *a kernel-mode* property).
→ **Focus** belongs to **a single GUI window** (and is *a user-mode* property).

Broadcast of keystrokes by the system with Window Messages



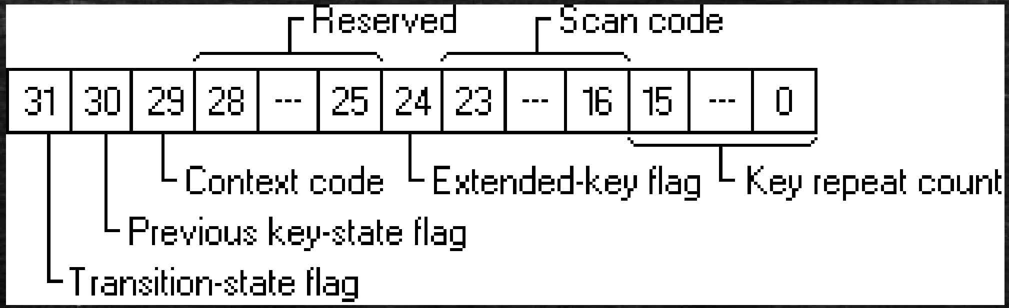
that



How to become a foreground thread?

How to interface with messages?

→ Technically speaking, **a message is a value**, but some messages may have data associated.
- This is the case with keyboard input messages.



→ There is a system of **registration** of **callbacks functions** used to interface with different messages.
- Messages are represented as constant values starting by WM_Xxx.
- For instance, **WM_KEYDOWN** and **WM_SYSKEYDOWN** to interface keyboard's messages.
- Message **WM_INPUT** is used to directly interact with the keyboard as HID content.

Other means to access keyboard

→ Internally within **RIT**, there are **different structures** that **represent the state** of the keyboard keys.

- The **state** can be **pressed** or **released** (or held).
- Used to know whenever a key is pressed if another one is also pressed:
  - **Shift** for **uppercase management** or multiple keys combinations within shortcuts.

→ Access **can be synchronized** with the reception of a message **or completely asynchronous**.

- For **synchronous access**, it is necessary to have **focus** from the keyboard.
- For **asynchronous access**, there is **no need of focus** (the freedom is much greater).

→ In the case of a **synchronous access**, the keyboard status changes as a thread retrieves keyboard messages from its message queue.

- **Rarely used**, `GetKeyboardState` function to get the **full representation of a keyboard** when a message is received.
- `GetKeyState` function is used to know **if another targeted key** has been **pressed** when a message is received.

Other means to access keyboard

→ In the case of an **asynchronous access**, it is possible to **listen** to the whole **keyboard stealthily**.
- It **does not depend** on current thread's **message queue**.
- **Neither subject** to keyboard **focus** nor concerned by **foreground thread property**.

→ This is the **preferred approach** used by **most malware** (with `GetAsyncKeyState` function).
- Test only **one virtual key code at time**.
- **Simple to use** (testing **each VK code** in a loop from 0 to 255) for **efficient results**.

→ But not free from drawbacks:
- **It could miss some keystrokes** in the loop enumeration (**balance** between **CPU consumption** and **efficiency**).
- **Listening is limited** to the **current desktop only** (for **security** purposes).

→ `GetAsyncKeyState` is **based on internal RIT structures** (mainly `gafAsyncKeyState`) that represent the current state of the keyboard.

Oth

→ An ... es from applications.

→ The ... ing mechanism.

→ The ... only.

→ Reg

Hook type	Scope of the hook	Reference	Description
WH_CALLWNDPROC	Thread or global	[974]	Monitor messages sent to window procedures. Called before passing the message to the receiving window procedure.
WH_CALLWNDPROCRET	Thread or global	[975]	Monitor messages sent to window procedures. Called after the window procedure has processed the message.
WH_CBT	Thread or global	[976, 977]	Called before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window; before completing a system command; before removing a mouse or keyboard event from the system message queue; before setting the input focus; or before synchronizing with the system message queue.
WH_DEBUG	Thread or global	[978]	Called before calling hook procedures associated with any other hook in the system.
WH_FOREGROUNDIDLE	Thread or global	[979]	Called when the application's foreground thread is about to become idle.It is useful for low priority tasks during times when foreground thread is idle.
WH_GETMESSAGE	Thread or global	[973]	It enables an application to monitor messages about to be returned by the GetMessage or PeekMessage function. Can be used to monitor mouse and keyboard input.
WH_JOURNALRECORD	Global only	[980]	It enables a hook procedure to monitor and record input events. Useful to record a sequence of mouse and keyboard events to play back later by using WH_JOURNALPLAYBACK.
WH_JOURNALPLAYBACK	Global only	[981]	Used to play back a series of mouse and keyboard events recorded earlier by using WH_JOURNALRECORD. It is possible to insert messages into the system message queue. Regular mouse and keyboard input is disabled as long as this hook is installed. It returns a time-out value to tell the system how many milliseconds to wait before processing the current message from the playback hook.
WH_KEYBOARD	Thread or global	[982]	Used to monitor keyboard input posted to a message queue. It monitors message traffic for WM_KEYDOWN and WM_KEYUP messages about to be returned by GetMessage or PeekMessage functions.
WH_KEYBOARD_LL	Global only	[983]	Enables an application to monitor keyboard input events about to be posted in a thread input queue.
WH_MOUSE	Thread or global	[984]	Used to monitor keyboard input posted to a message queue. It monitors message traffic for mouse messages (WH_MOUSE) about to be returned by GetMessage or PeekMessage functions.
WH_MOUSE_LL	Global only	[985]	Enables an application to monitor mouse input events about to be posted in a thread input queue.
WH_MSGFILTER	Thread or global	[986]	It monitor messages passed to a menu, scroll bar, message box, or dialog box created by the application. It allows to filter messages during modal loops that is equivalent to the filtering done in the main message loop.
WH_SHELL	Thread or global	[987]	Used to receive important notifications. When the shell application is about to be activated and when a top-level window is created or destroyed
WH_SYSMSGFILTER	Global only	[988]	Same as WH_MSGFILTER but monitors messages for each application.

Table 4.18: List of hooks types with their scope associated (from [1, 2]).

Miscellaneous about accessing keyboard

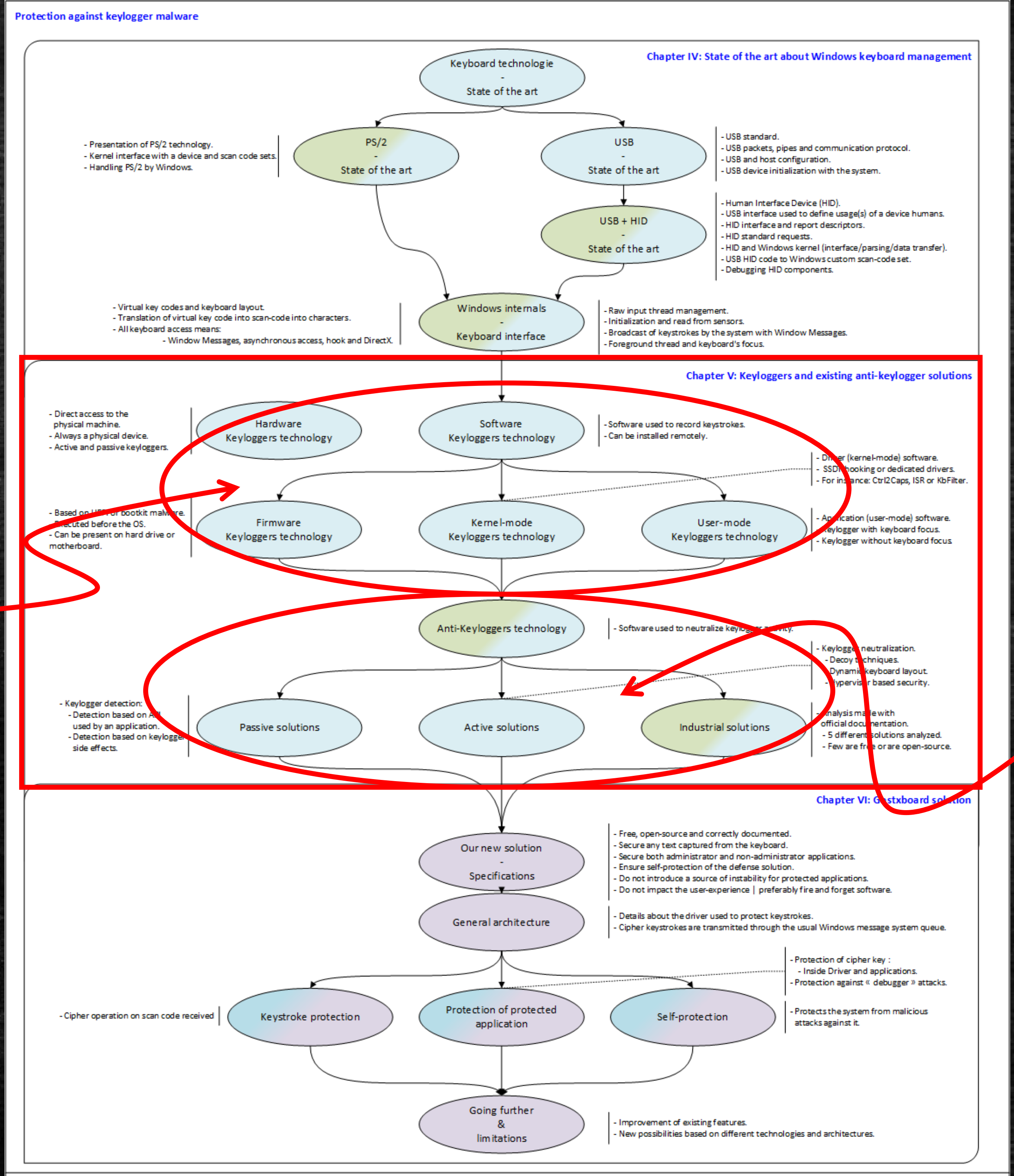
→ In practice, there are **libraries** that **manage the keyboard** directly.

→ To do this, **two approaches** are possible:

- A **wrapper** of the **Windows API** (an overlay) hiding the complexity with a nice interface.
  - Qt, SDL, OpenCV, Tk, Gtk, script languages…
- **Bypasses** (or ignores) the **Windows message system** to manage the keyboard directly.
  - **DirectX** or any **home-made kernel level** (with a user-mode interface) library.

→ **DirectX** (mostly used by video games for performances reasons):

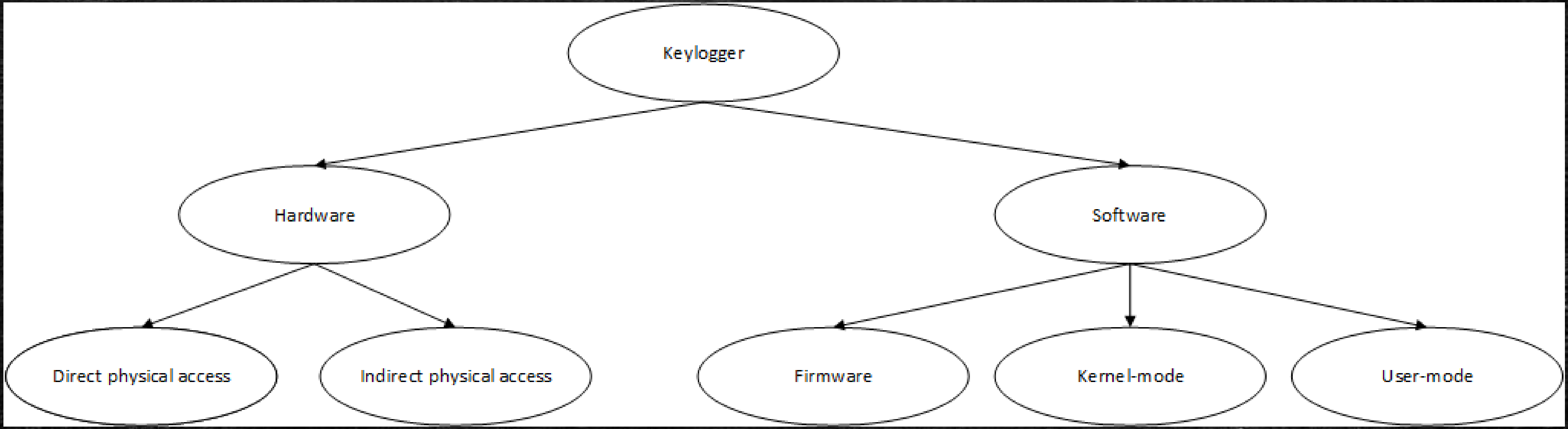
- This implies **a loss** of all the **interactions** and **facilities** offered by the RIT.
- It can be seen as a **parallel channel** to **convey** the **keystrokes**.
- The **keyboard** must be **acquired** to be read.
  - It can be **released thereafter**.
  - **Other applications** (using Windows API) can be **deprived** of keyboard keys.
- DirectX does not use the virtual key code but its own code.
  - Based to the position of physical keys (video games ignores meanings of keys' labels).

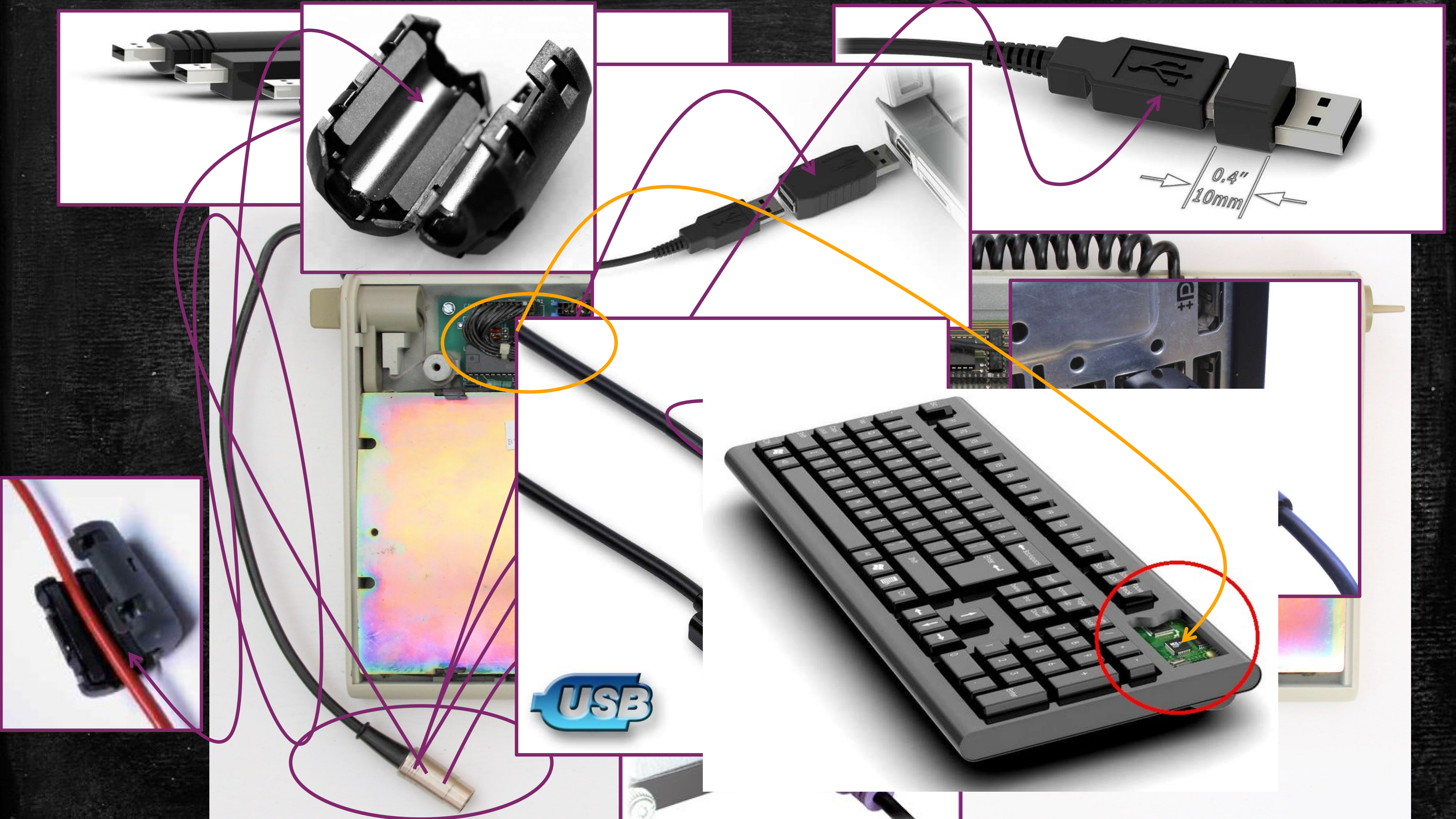


Threat situation - Keyloggers

Defense situation - Anti-Keylogger solutions

The key-loggers' families




0.4"
10mm
USB

Hardware keyloggers – indirect access

→ **Indirect access** hardware keylogger devices:

- It aims to **capture** a **signal**: **electromagnetic**, **sound** or coming from another source.
- Then, it **analyses** a signal to **deduce** the **keystrokes** from the keyboard.

→ Some attacks are **fully operational**; others are more **experimental**...

→ **Wireless** keylogger:

- Bluetooth interfaces use a range from 27 MHz up to 2.4 GHz radio frequency (RF)
- A transmission range limited to a radius of six feet (close to 2 meters).
- But it can be captured up to the **distance of 100 meters** by dedicated hardware.
- Wireless keyboard manufacturers encrypt RF transported keystroke characters.
    - But the **encryption**, at least on Microsoft keyboards in 2008, **can be very weak**.

Hardware keyloggers – indirect access

→ **Acoustic keylogger**:

- Detection based on the sound of individual keystrokes thanks to special parabolic microphones.
- Each keystroke have a particular sound which can be distinguishable.
  - This is due to the plate underneath the keys that is not uniform on regular keyboards.
  - Particularly efficient on mechanical keyboards which are noisy
- Use of quieter keyboards may also reduce vulnerability.
  - Required by US department of defence NACSEM 5103, 5104, and 5105 (classified).

→ **Solutions** against hardware keyloggers?

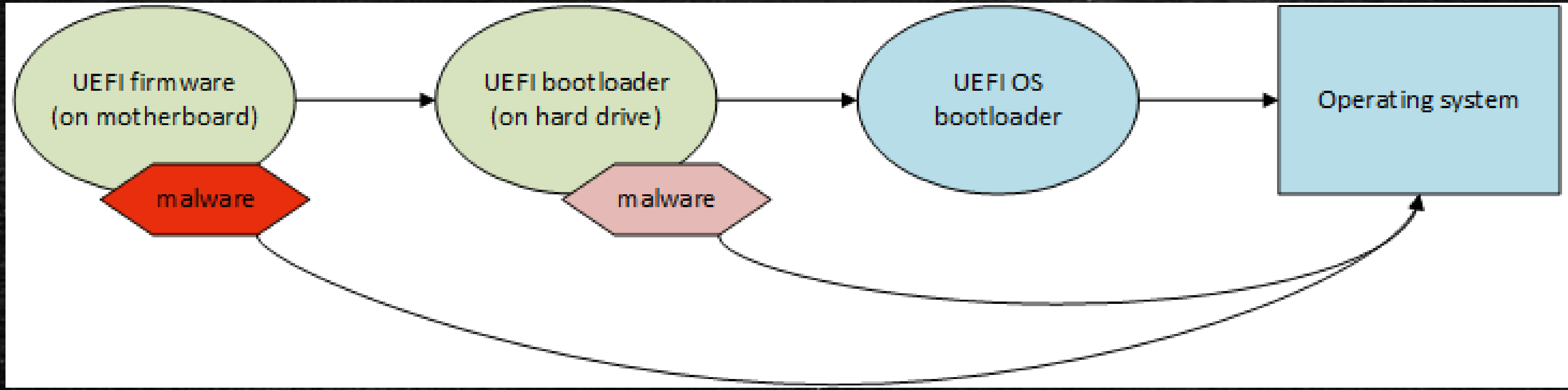
- It must be assumed if an attacker has a physical access to the victim's computer, the war is lost.
- "**If someone can gain physical access to your computer, it is not your computer anymore**".

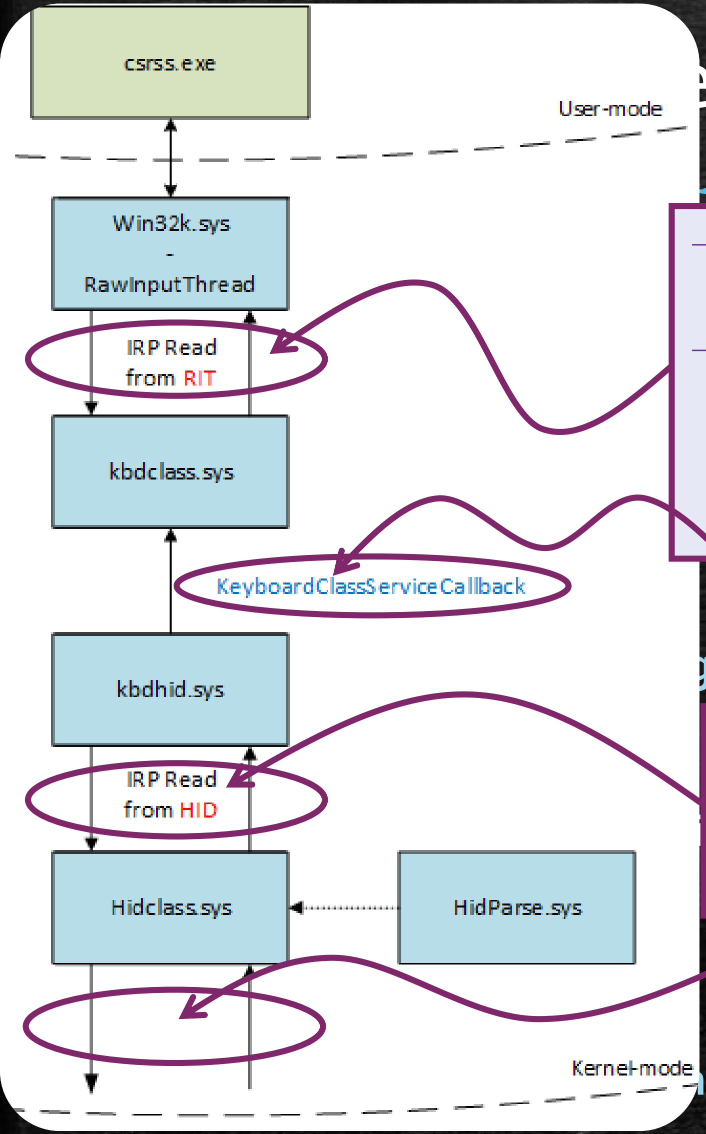
Software keyloggers

→ We propose to divide the taxonomy of software keyloggers into three categories:

- **Firmware**: **Before** the **operating system** is started, at motherboard or **UEFI/BIOS** level.
- **Kernel-mode**: With the **highest level of privileges** within the system, a driver for instance.
- **User-mode**: As a **regular application** in the system.

→ **Firmware**:





e keyloggers

→ **Handling IRP** as a legacy filter driver for device.
- IOCLT handler routines are used to handle READ – WRITE – ADD device operations.

→ **Ctrl2Caps** driver developed by *Mark Russinovich* uses this technology.
- Swap two keys from the physical keyboard.

→ **Hijacking** the `KeyboardClassServiceCallback` routine chain.
- `IOCTL_INTERNAL_KEYBOARD_CONNECT` **request** is **sent** for each keyboard device plugged in the system.
- It provides a `CONNECT_DATA` structure updatable with a **callback routine** interfacing keyboard driver.
- The **callback** can **read**, **modify** or **delete** any **keystroke received** from the keyboard.
- **Required** to **maintain** the **chain of callbacks** (the "*new*" callback must call the "*former*" one).

→ **KbFiler** driver is a "**tutorial**" WDF driver **provided** by **Microsoft**.

- T

g a

el-land.

the registry.

→ **Handling IRP** between keyboard HID and HID Class driver.
- Possible by tricky: we need to deal with HID reports directly.
- A lot of lines of code, high probability of parsing bugs (HID is self-defined).
- Few examples are close source.

→ **Handling HID IRP** a low level driver.
- Need to **interface ALL HID devices** (where the keyboard only matters).
- **A lot more work** about HID parsing with high requirements for performances.
- No example online except the kernel of Windows.

ne same API as any regular application.

- The difference is in the final aim of the data captured from the keyboard.

Anti-keylogger solutions

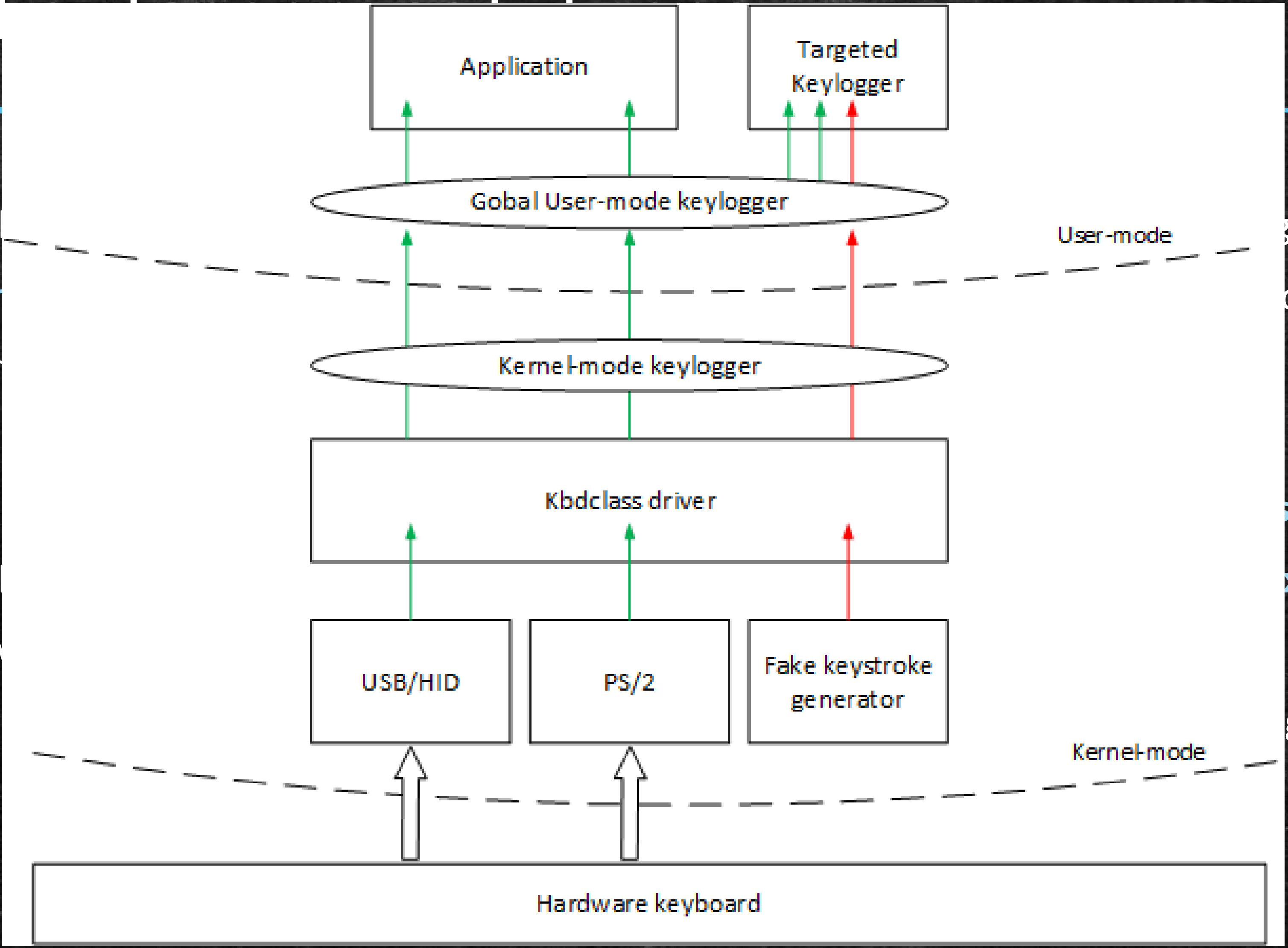
→ Practically speaking, there are two ways to fight keyloggers.

- "**Antivirus like**" detection of keyloggers (**before** or during **execution**) → **Passive** solutions.
- **Mitigate** of consequences done by keyloggers (**during execution**) → **Active** solutions.

→ The **detection** of **keylogger** threats (regardless of the method used) is a **very complex problem**.

- Keyloggers capture keystrokes just like any other legitimate program
- And it is **very difficult** to **characterize** the ***intent*** of a **program**.
- We prefer to **neutralize** them ;-) .

Anti-k



→ **Active** so … gger.

- In … ce.

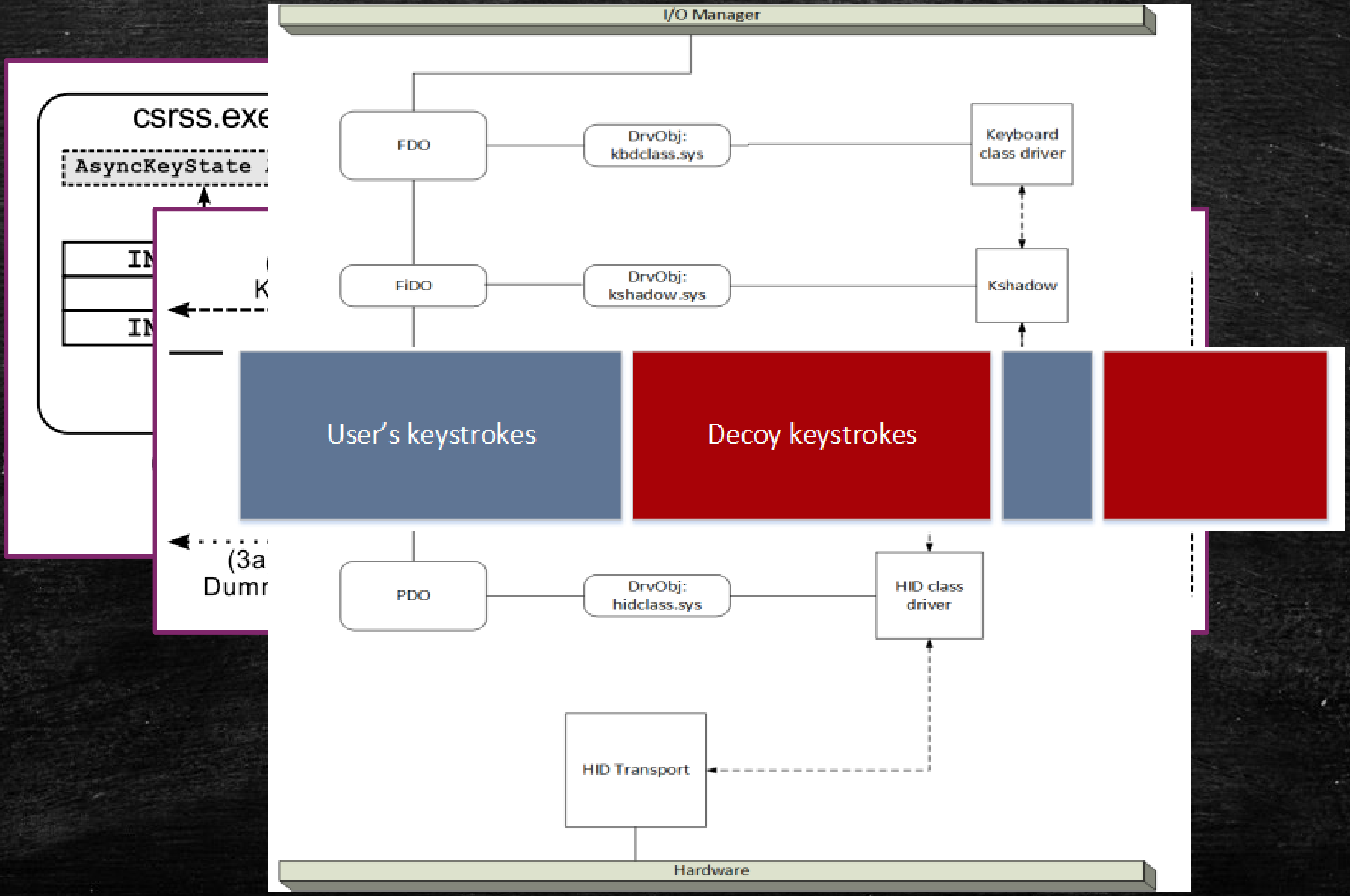
- In

→ **Jamming** … **strokes**.

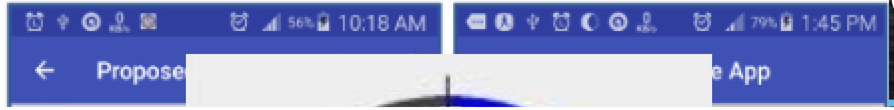
- Th … **xperience**.

- Tw

… gers.


csrss.exe
I/O Manager
FDO
DrvObj: kbdclass.sys
Keyboard class driver
FiDO
DrvObj: kshadow.sys
Kshadow
User’s keystrokes
Decoy keystrokes
PDO
DrvObj: hidclass.sys
HID class driver
HID Transport
Hardware

Miscellaneous Anti-keylogger solutions

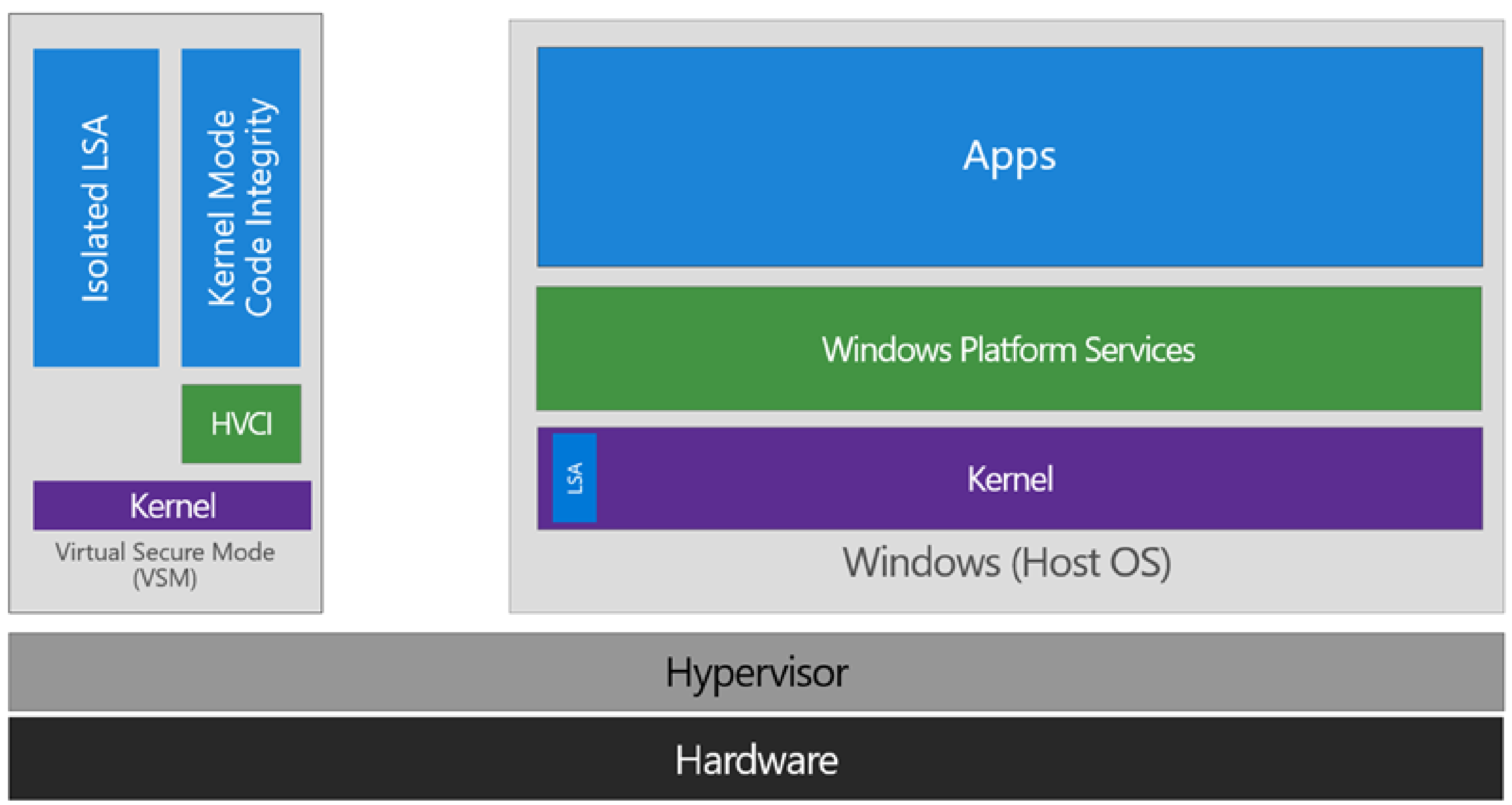


→ **Dynamic layout** technique:

- The idea is to change t
    - Good to prot
    - But not conv
    - In addition, t … efficient.

→ **Hypervisor** based solutions:

- The use of a **hyperviso**
- **Hypervisor** is used as a
- But it is **hard to interfa**
    - Where to "re
    - “Escorting” k
        - Ha



- There is an **induced delay** in the processing time of the keys, **but imperceptible for the user**.
- **Windows 10** uses hypervisor with **virtualization-based security** to enhance system of the security.
    - We cannot add another hypervisor beside the first one.

Indu

→ Many … ows (ie: RIT).

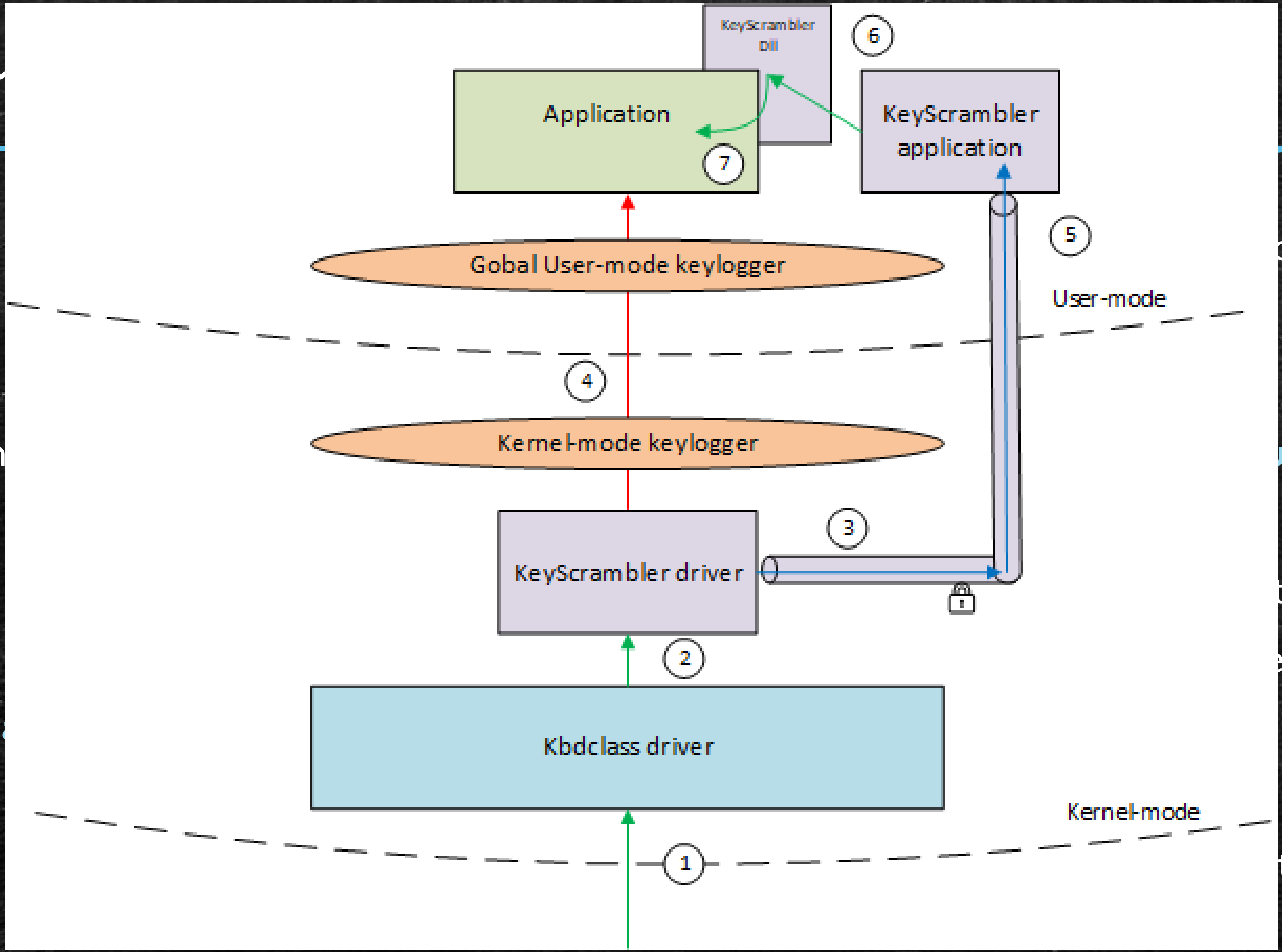
→ We sh … us.

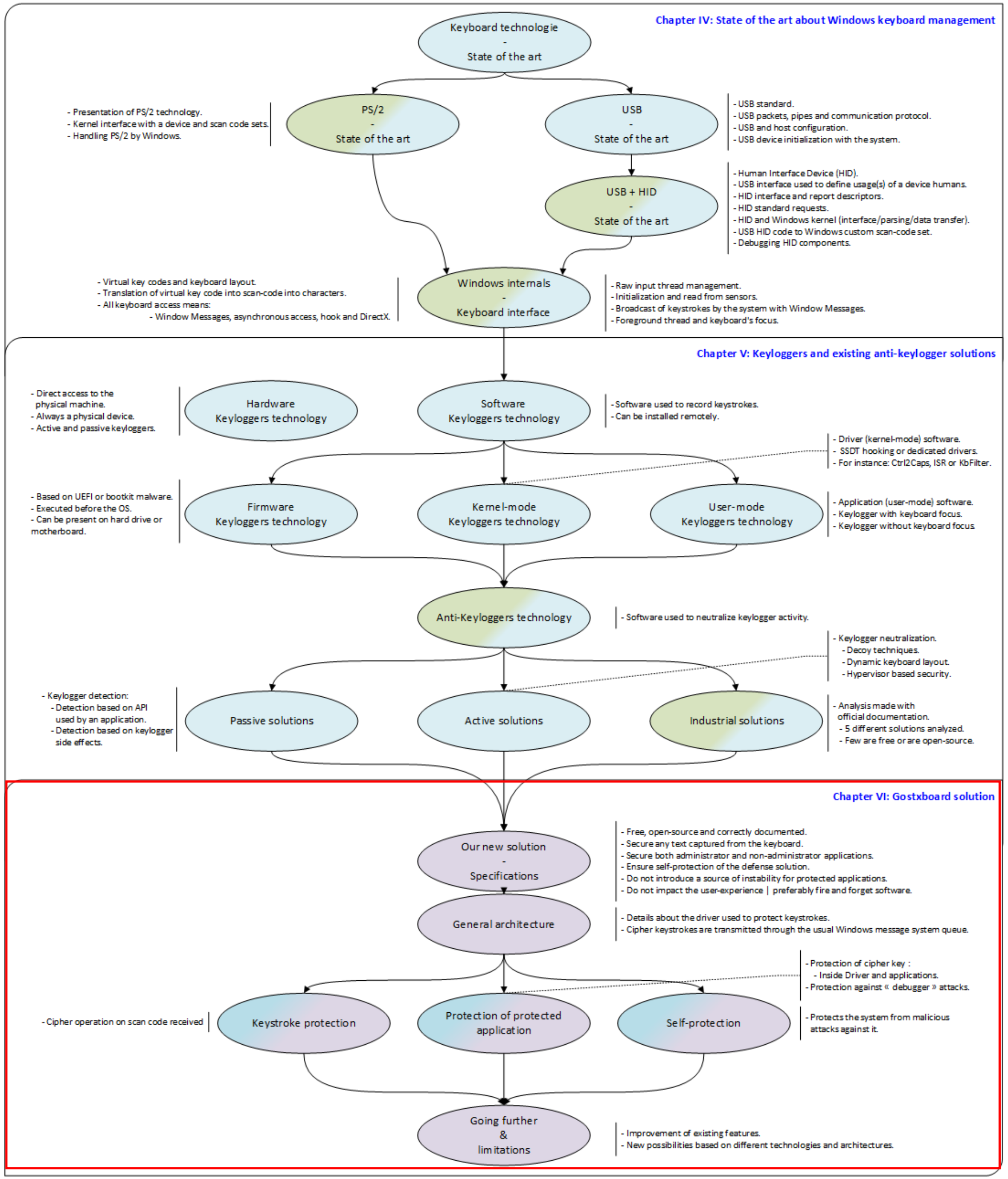
the protected application.

ebuggers, etc.).

→ **Malw … ll** any security software.

t sessions/desktops…




Chapter IV: State of the art about Windows keyboard management
Keyboard technologie - State of the art
PS/2 - State of the art
- Presentation of PS/2 technology.
- Kernel interface with a device and scan code sets.
- Handling PS/2 by Windows.
USB - State of the art
- USB standard.
- USB packets, pipes and communication protocol.
- USB and host configuration.
- USB device initialization with the system.
USB + HID - State of the art
- Human Interface Device (HID).
- USB interface used to define usage(s) of a device humans.
- HID interface and report descriptors.
- HID standard requests.
- HID and Windows kernel (interface/parsing/data transfer).
- USB HID code to Windows custom scan-code set.
- Debugging HID components.
Windows internals - Keyboard interface
- Virtual key codes and keyboard layout.
- Translation of virtual key code into scan-code into characters.
- All keyboard access means:
- Window Messages, asynchronous access, hook and DirectX.
- Raw input thread management.
- Initialization and read from sensors.
- Broadcast of keystrokes by the system with Window Messages.
- Foreground thread and keyboard's focus.
Chapter V: Keyloggers and existing anti-keylogger solutions
Hardware Keyloggers technology
- Direct access to the physical machine.
- Always a physical device.
- Active and passive keyloggers.
Software Keyloggers technology
- Software used to record keystrokes.
- Can be installed remotely.
- Driver (kernel-mode) software.
- SSDT hooking or dedicated drivers.
- For instance: Ctrl2Caps, ISR or KbFilter.
Firmware Keyloggers technology
- Based on UEFI or bootkit malware.
- Executed before the OS.
- Can be present on hard drive or motherboard.
Kernel-mode Keyloggers technology
User-mode Keyloggers technology
- Application (user-mode) software.
- Keylogger with keyboard focus.
- Keylogger without keyboard focus.
Anti-Keyloggers technology
- Software used to neutralize keylogger activity.
- Keylogger neutralization.
- Decoy techniques.
- Dynamic keyboard layout.
- Hypervisor based security.
Passive solutions
- Keylogger detection:
- Detection based on API used by an application.
- Detection based on keylogger side effects.
Active solutions
Industrial solutions
- Analysis made with official documentation.
- 5 different solutions analyzed.
- Few are free or are open-source.
Chapter VI: Gostxboard solution
Our new solution - Specifications
- Free, open-source and correctly documented.
- Secure any text captured from the keyboard.
- Secure both administrator and non-administrator applications.
- Ensure self-protection of the defense solution.
- Do not introduce a source of instability for protected applications.
- Do not impact the user-experience | preferably fire and forget software.
General architecture
- Details about the driver used to protect keystrokes.
- Cipher keystrokes are transmitted through the usual Windows message system queue.
Keystroke protection
- Cipher operation on scan code received
Protection of protected application
- Protection of cipher key :
- Inside Driver and applications.
- Protection against « debugger » attacks.
Self-protection
- Protects the system from malicious attacks against it.
Going further & limitations
- Improvement of existing features.
- New possibilities based on different technologies and architectures.

Solutions?

→ First, to specify that there is **no perfect solution** … but we can do *much* **better**.

1) **Malware detection** is **too limited** when dealing with keylogger threat.

→ **Keyloggers** are just **regular software** using **regular API** for **bad purposes**…

→ The problem is much more on **philosophy side** than on technical one…

→ Try to **limit** potential **impact**/theft coming from this problem.

→ Secures the **keystrokes data flow**, like a **bodyguard**.

Solutions?

2) **Design software** to take care of **secure input means** from the **beginning**.

→ "**By design**" to ensure **that specific part** in the software must be **secure**.
- Problem is not languages or frameworks, but **what we do with data**.

→ Do **not inject Dll** inside an **existing software**:
- Nobody expect you ...
- Most of targeted software are not ready for that ...
- In some cases, it may crash the application → instability.

→ A better **API** (why not handled by Hyper-V) could be a good move...

Solutions?

3) **Protect protected application** from being hacked in memory.

→ **Avoid "debugger"** stuff from access/modifying other's memory.

→ "**Administrator**" is not a protection against Read/WriteProcessMemory.

→ And **not all applications** should run with such privileges…

→ Different ways to **prevent Dll injections**, but that's another story ;-).

- Protected process (light) is one ☺.

Solutions?

4) **Secure** you **protection system**.

→ It is sometime **easier** to **deactivate/control/hijack** the protection than the target…

→ But there is a **balance** between **protection** and **user's rights** to uninstall/resume a software.

- It's user's own machine, after all ☺.

Solutions?

5) **Work for** your **users** first.

→ Is our new so-called **golden feature** worth **ruining** the **user experience**?

→ Do not use **undocumented API**, it is source of **instability**.

→ **Knows** how work the system, the threat and what you can do…

- Bring **visibility into data flows** and not just loss prevention and system reliability.

Thank you for your attention ☺

Do you have any question?