SECURELIST

Author

Nikolay Grebennikov

Keyloggers: Implementing keyloggers in Windows. Part Two

This article is a continuation of the previous report on keyloggers. It offers a detailed analysis of the technical aspects and inner workings of keyloggers. As was noted in the first article, keyloggers are essentially designed to be injected between any two links in the chain whereby a signal is transmitted from a key being pressed to symbols appearing on the screen. This article provides both an overview of which links exist in this chain, and how both software and hardware keyloggers work.

This article is written for technical specialists and experienced users. Other users, who are not part of this target group, should simply be aware that Windows offers a multitude of ways in which data entered via the keyboard can be harvested, although the vast majority of keyloggers only use two of these methods (see: Designing keyloggers, the first part of the article).

It should be stressed that this article does not include any keylogger source code; we do not share the opinion of some researchers who believe it is acceptable to publish such code. Instead, we focus on understanding how keyloggers work, so we can better implement effective protection against them.

Processing data entered via the keyboard in Windows

There are several basic technologies which can be used to intercept keystrokes and mouse events, and many keyloggers use these technologies. However, before examining specific types of keylogger, it's necessary to understand how data entered via the keyboard is processed by Windows. To describe the process – from a key being pressed on the keyboard to the keyboard system interrupt controller being activated and an active WM_KEYDOWN message appearing, three sources have been used:

1. "Apparatnoe obeshpechenie IBM PC" by Alexander Frolov and Grigory Frolov, Volume 2, Book 1. Published by Dialog-MIFI, 1992. The second chapter, "The Keyboard" described how a keyboard functions, the ports used, and keyboard hardware interrupts;
2. The section related to "HID / Human Input Devices" of the MSDN library, which describes the low-level (driver) part of the process by which keyboard input is processed;
3. Jeffrey Richter's book 'Creating effective Win32 applications for 64-bit Windows". Chapter 27, "A model of hardware input and the local input condition" contains a description of the high level part of the process by which keyboard input is processed (in user mode).

The keyboard as a physical device: how it works

Today, the majority of keyboards are a separate device connected to the computer via a port – most frequently PS/2 or USB. There are two micro-controllers which support the processing of keyboard input data; one is part of the motherboard, the other is within the keyboard itself. Consequently, it could be said that a PC keyboard is itself a small computer system. It uses an 8042 microcontroller which constantly scans keys being pressed on the keyboard independently of central CPU activity.

Each key on the keyboard has a specific number assigned to it; this is linked to keyboard matrix map and is not directly dependent on the value shown on the surface of the key itself. This number is called a "scan code" (the name highlights the fact that the computer scans the keyboard to search for keystrokes). Scan codes are random values which were selected by IBM back in the days when the company created the first keyboard for PCs. A scan code does not

correspond to the ASCII code of a key; actually, a single key may correspond to several ASCII code values. A table of scan codes can be found in the twentieth chapter of “The Art of Assembly Language Programming”.

In actual fact, the keyboard generates two scan codes for each key: one for when the user presses the key, and another for when the user releases the key. The fact that there are two scan codes is important, as some keys only have a function when they are pressed and held (eg. Shift, Control or Alt).

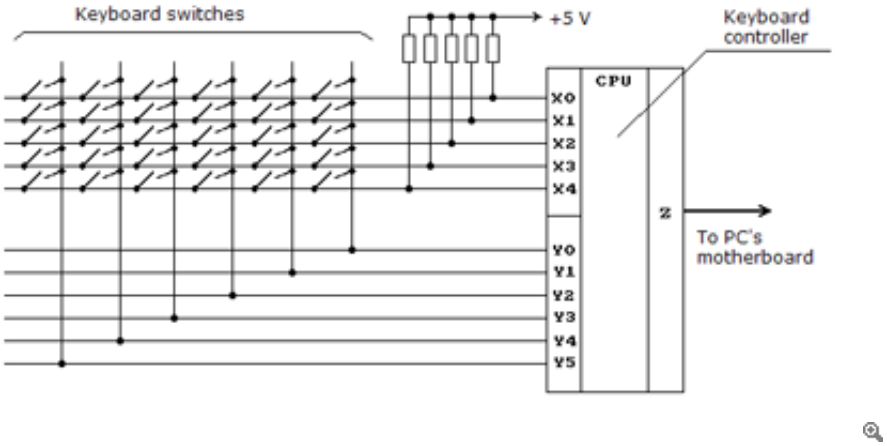

Fig. 1: The keyboard: a simplified scheme

When the user presses a key on the keyboard, s/he closes an electrical circuit. As a result, when performing the next scan, the micro-controller detects that a specific key has been pressed, and sends the scan code of the key pressed to the central computer, together with an interrupt request. The same action is performed when the user releases the key that s/he has previously pressed.

The second micro-controller receives the scan code, converts it, makes it accessible on the input/ output port 60h and then generates a central processor hardware interrupt. The handler which processes the interrupt can then get the scan code from the designated input/ output port.

It should be noted that the keyboard contains an internal 16 byte buffer which it uses to exchange data with the computer.

Low-level interaction with the keyboard via the input/ output port

Interaction with the keyboard system controller takes place via the input/ output port 64h. Reading a byte from this port makes it possible to determine the status of the keyboard controller; writing a byte makes it possible to send a command to the controller.

Interaction with the micro-controller within the keyboard itself takes places via the input/ output ports 60h and 64h. The 0 and 1 bits in the status byte (port 64h in read mode) make it possible to control the interaction: before writing data to these ports, bit 1 of port 64h should be 0. When data is read-accessible from port 60h, bit 1 of port 64h is equal to 1. The keyboard on/ off bits in the command byte (port 64h in write mode) determine whether or not the keyboard is active, and whether the keyboard controller will call a system interrupt when the user presses a key.

Bytes written to port 60h are sent to the keyboard controller, while bytes written to port 64h are sent to the keyboard system controller. A list of permissible commands which can be sent to the keyboard controller can be found in, for example, “8042 Keyboard Controller IBM Technical

Reference Manual" or in the twentieth chapter of "The Art of Assembly Language Programming".

Bytes read from port 60h come from the keyboard. When reading, port 60h contains the scan code of the last key pressed, and in write mode this is used for extended management by the keyboard. When using port 60h in write mode, a program has the following additional options:

- Establishing a wait period before the keyboard goes into auto repeat mode
- Establishing the interval for generating scan codes in auto repeat mode
- Management of LEDs located on the outer surface of the keyboard – Scroll Lock, Num Lock, Caps Lock.

To summarize, in order to read data entered via the keyboard, one only has to be able to read the values of input/ output port 60h and 64h. However, user-level applications in Windows are unable to work with ports; this function is fulfilled by operating system drivers.

The architecture of "interactive input devices"

So, what processes the hardware interrupts which are generated when data sent by the keyboard appears on port 60h? Clearly, it's done through the handler of the keyboard hardware interrupt IRQ 1. In Windows, this processing is conducted by the system driver i8042prt.sys. In contrast to MS DOS, when each system component was a law unto itself, as it were, in Windows all components are constructed in accordance with a clear architecture and work in accordance with strictly defined rules assigned by the program interface. So before examining i8042prt, let's take a look at the architecture of interactive input devices, within the confines of which all program components connected with the processing of keyboard (and 'mouse') input function.

In Windows, devices which are used to manage operations on the computer are called 'interactive input devices'. A keyboard is one such device, together with the mouse, joysticks, trackballs, game controllers, wheels, virtual reality helmets etc.

The architecture of interactive input devices is based on the USB Human Interface Device standard put forward by the USB Implementers Forum. However, this architecture is not limited to USB devices, and supports other input devices, such as Bluetooth keyboards, PS/2 keyboards and mice and devices connected to I/O port 201 (the gaming port).

Later in this article we will examine how the driver stack and the device stack for a PS/2 keyboard are constructed (given that historically the PS/2 keyboard was the first type of device described above). USB keyboards use a range of elements which were introduced during the development of program support for PS/2 keyboards).

Kernel mode drivers for PS/2 keyboards

Driver stack for system input devices

Regardless of how the keyboard is physically connected, keyboard drivers use keyboard class system drivers to process data. This happens regardless of the hardware used. Actually, these drivers are called class drivers because they support system requirements independent of the hardware requirements of a specific class of device.

The corresponding functional driver (port driver) supports the execution of input/ output operations in correlation with the device being used. In x86 Windows this is implemented in a single system keyboard driver (i8042) and mouse driver.

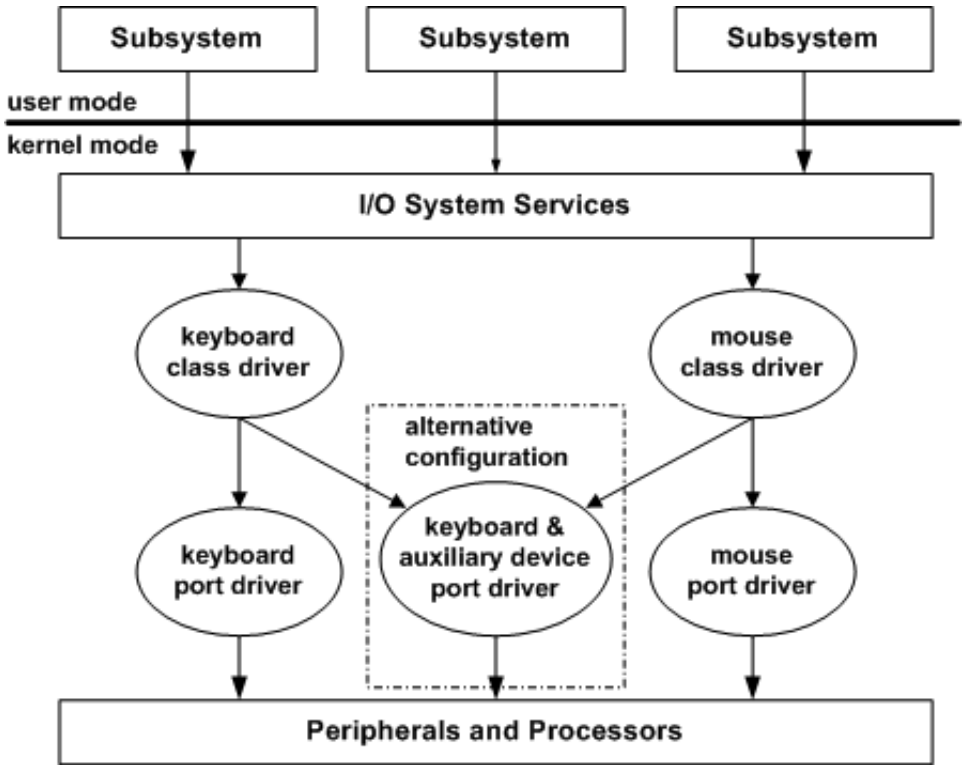

Fig 2.: Driver stack for system entry devices: keyboard and mouse

Driver stack for Plug and Play PS/2 keyboards

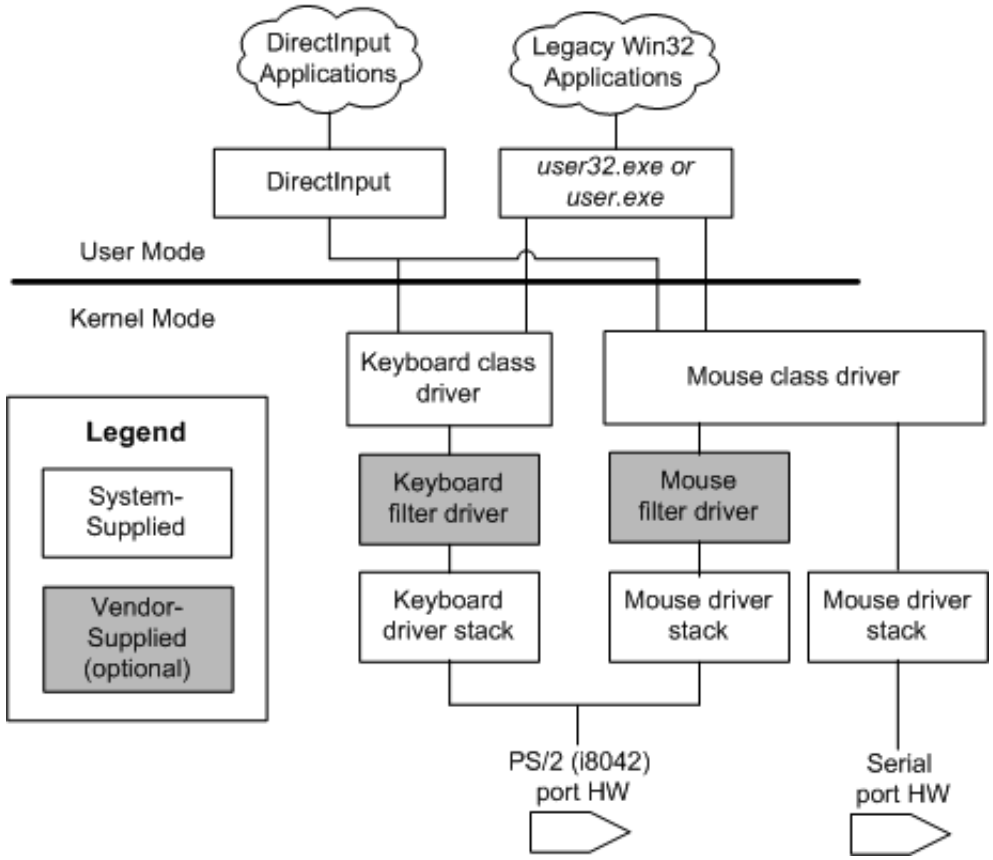

Fig 3. Driver stack for PS/2 keyboards

The driver stack (from top to bottom):

1. Kbdclass – high level filter driver, keyboard class;
2. optional high level filter driver, keyboard class;
3. i8042prt – functional keyboard driver
4. root bus driver

In Windows 2000 and older versions of Windows, the keyboard class driver is Kbdclass. The main tasks of this driver are:

- To support general and hardware-dependent operations of the device class
- To support Plug and Play, support power management and Windows Management Instrumentation (WMI)
- To support operations for legacy devices
- Simultaneous execution of operations from more than one device
- To implement the class service callback routine, which is called by the functional driver to transmit data from the device input buffer to the device driver data buffer.

In Windows 2000 and older versions of Windows, the functional driver for input devices using the PS/2 port (keyboard and mouse) is the i8042prt driver. The main functions are as follows:

- To support hardware dependent simultaneous operations of PS/2 input devices (the keyboard and mouse share the input and output ports, but use different interrupts, Interrupt Service Routines (ISR) and procedures for terminating interrupt processing;
- To supporting Plug and Play, power management and Windows Management Instrumentation (WMI);
- To supporting operations for legacy devices;
- To call the class service callback routine for classes of keyboards and mice in order to transmit data from the input data buffer i8042prt to the device driver data buffer;
- To call a range of callback functions which can be implemented in high level driver filters for flexible management by a device

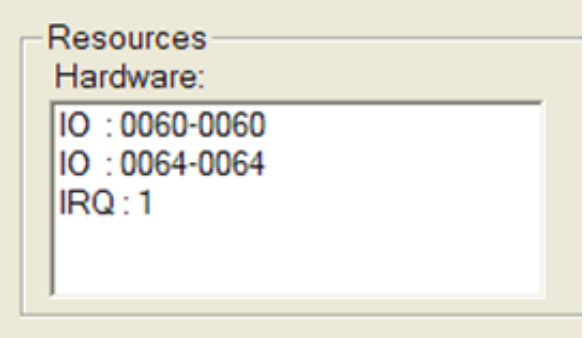

Fig. 4: Hardware resources which use the i8042prt driver

Figure 4 shows a list of the hardware resources which use the i8042prt driver. These can be viewed, for example, using DeviceTree (http://www.osronline.com/article.cfm?article=97), a utility developed by Open Systems Resources. If you've read the sections on 'The keyboard as a physical entity – how it works' and 'Low-level interaction with the keyboard via the input/ output ports" the values of the input/ output ports of 60h and 64h, and the hardware interrupt (IRQ) 1 will not come as any surprise.

A new **driver filter** can be added above the keyboard class driver in the driver stack shown above in order to, for instance, perform specific processing of data entered via the keyboard. This driver should support the same processing of all types of input/ output requests and management commands (IOCTL) as the keyboard class driver. In such cases, before data is transmitted to the user mode subsystem, the data is passed for processing to this driver filter.

Device stack for Plug and Play PS/2 keyboards

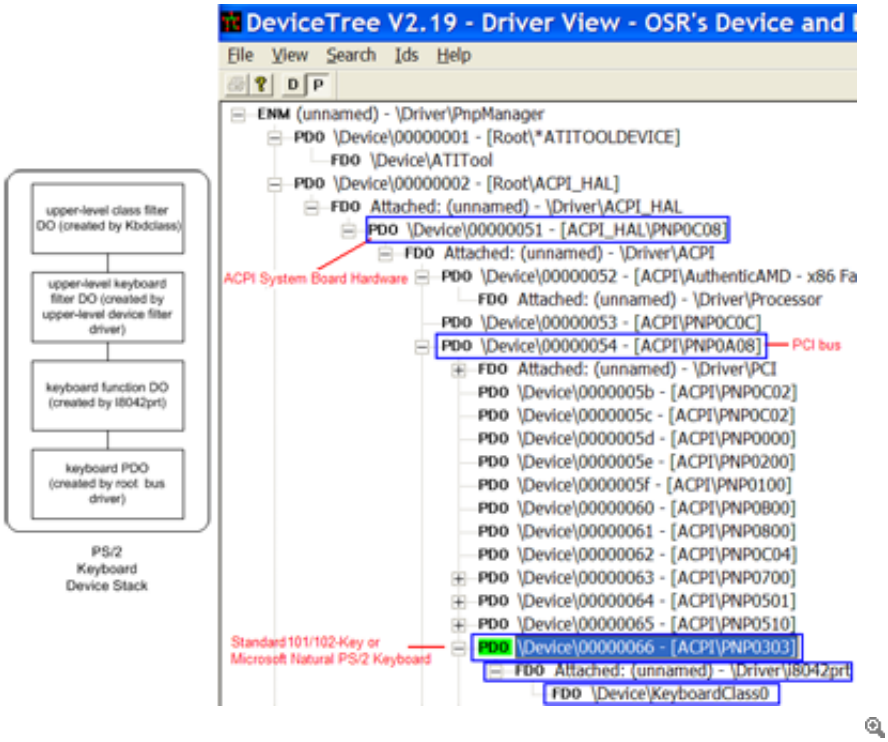

Fig. 5: Configuration of device objects for Plug and Play PS/2 keyboards.

Overall, the device stack (which more correctly should be called the device object stack) for a P/S2 keyboard is made up of:

1. The physical device object (PDO), created by the driver bus (in this case, the PCI bus) – \Device\00000066;
2. The functional device object (FDO), created and connected to the PDO by the i8042prt port – an unnamed object;
3. Optional filter objects for the keyboard device, created by the keyboard driver filters, which are developed by third party developers;
4. High level filter objects for the keyboard device class which are created by the Kbdclass class driver - \Device\KeyboardClass0.

Processing keyboard input via applications

Raw input thread (data received from the driver)

The previous section gives examples of how keyboard stacks are constructed in kernel mode. This section takes a look at how data about keystrokes is transmitted by applications in user mode.

The subsystem of Microsoft Win32 gets access to the keyboard by using the Raw Input Thread (RIT), part of the csrss.exe system process. On boot, the system creates the IRT and the system hardware input queue (SHIQ).

The RIT opens the keyboard class device driver for exclusive use and uses the ZwReadFile function to send it an input/ output request (IRP) of the type IRP_MJ_READ. Having received the request, the Kbdclass driver flags it as pending, places it in the queue and returns a STATUS_PENDING code. The RIT has to wait until the IRP terminates, and in order to determine this it uses the Asynchronous Procedure Call or ACP.

When the user presses or releases one of the keys, the keyboard system controller yields a hardware interrupt. The hardware interrupt processer calls a special procedure to process the IRQ 1 interrupt (the interrupt service routine, or ISR), which is registered in the system by the i8042prt driver. This procedure reads the data which has appeared from the internal keyboard controller queue. The processing of the hardware interrupt should be as quick as possible; because of this, the IRC places a Deferred Procedure Call (or DPC), I8042KeyboardIsrDpc and then terminates. As soon as is possible (the IRQL reverts to DISPATCH_LEVEL), the DPC is called by the system. At this moment the callback procedure KeyboardClassServiceCallback will be called, which is registered by the i8042 driver Kbdclass driver. KeyboardClassServiceCallback extracts the pending termination request (IRP) from its queue, completes the maximum amount of KEYBOARD_INPUT_DATA which provide all the information required about keys pressed and released and terminates the IRP. The raw input thread is activated again, processes the information received, and sends another IRP to the class driver, which will again be queued until the next key press/ release. This means that the keyboard stack always contains at least one pending termination request in the Kbdclass driver queue.

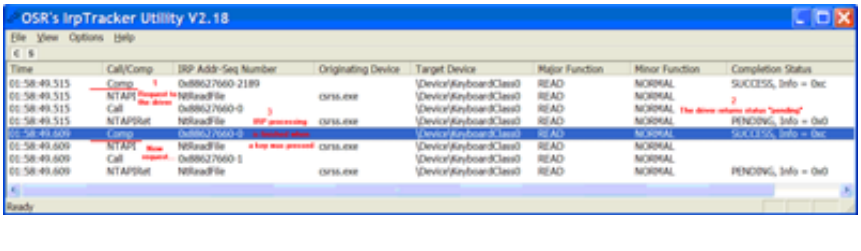

Fig. 6: Sequence of requests from RIT to the keyboard driver

Using a utility called IrpTracker, developed by the previously mentioned Open Systems Resources, it's possible to track the sequence of calls which takes place when keyboard input is processed.

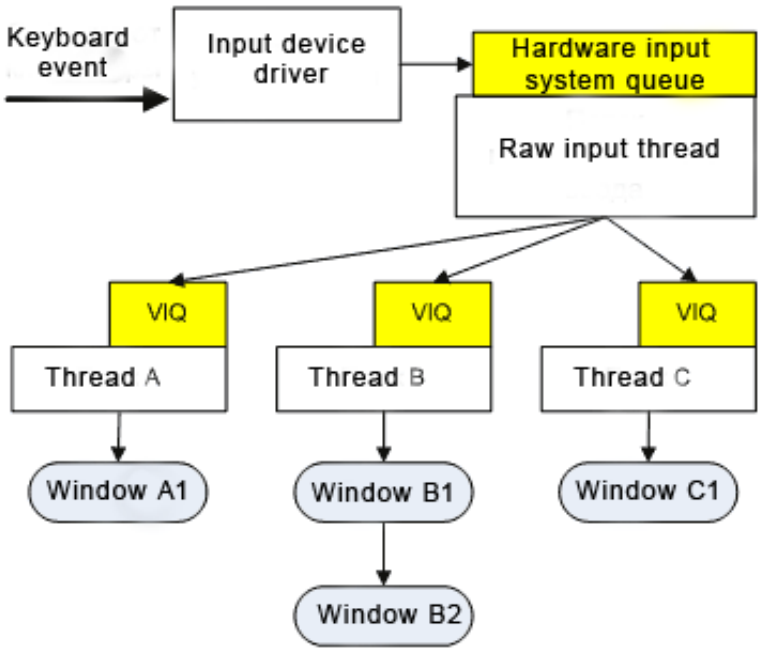

Fig. 7: Processing keyboard input in user mode

How does RIT process incoming data? All incoming keyboard events are placed in the hardware input system queue, and are in turn transformed into Windows messages (e.g. WM_KEY*, WM_?BUTTON* or WM_MOUSEMOVE) and are then placed at the end of the virtualized input queue, or VIQ of the active thread. The key scan codes in the Windows messages are replaced by virtual key codes which correspond not to the location of keys on the keyboard but the action that this key performs (function that it fulfils). The mechanism for transforming codes depends on the current (active) keyboard layout, simultaneous pressing of keys (e.g. SHIFT) and other factors.

When the user enters the system, the Windows Explorer process launches a thread which creates the task panel and the desktop (WinSta0_RIT). This thread binds to the RIT. If the user launches MS Word, then the MS WORD thread, having created a window, will immediately connect to the RIT. The Explorer process will then unhook from the RIT, as only one thread can be connected to RIT at any one time. When a key is pressed, the relevant element will appear in the SHIQ; this leads to the RIT becoming active, transforming the hardware input event into a message from the keyboard which will then be placed in the MS Word VIQ thread.

Processing of messages by a specific window

How does a thread process messages from the keyboard which have entered the thread's message queue?

The standard message processing cycle usually looks like this:

```
while(GetMessage(&msg, 0, 0, 0))
{
TranslateMessage(&msg);
DispatchMessage(&msg);
}
```

Using the GetMessage function, keyboard events are extracted from the queue and redirected using the DispatchMessage function to the window procedure which processes messages for the

window where input is currently focussed. The input focus is an attribute which can be assigned to a window created by an application or by Windows. If the window has an input focus, all keyboard messages from the system queue will reach the appropriate function of this window. An application can pass the input focus from one window to another, e.g. when another application is switched to using Alt+Tab.

The TranslateMessage function is usually called prior to the DispatchMessage function. This function creates the 'symbolic' messages WM_CHAR, WM_SYSCHAR, WM_DEADCHAR and WM_SYSDEADCHAR using the WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, WM_SYSKEYUP messages as a base. These 'symbolic' messages are placed in the application message queue; however, it should be noted that the original keyboard messages are not deleted from this queue.

Keyboard key status array

One of the aims when developing the Windows hardware input model was to ensure resilience. Resilience is ensured by independent processing of input by threads; this prevents conflicts between threads. However, this is not enough to isolate threads from each other, and the system therefore supports an additional concept: local input status. Each thread has its own input condition, and information about this is stored in THREADINFO. The information includes data about the virtual queue thread, and a group of variables. This last contains management information about the input thread status. The following notifications are supported for the keyboard: which window is currently in the focus of the keyboard, which window is currently active, which keys are pressed and the status of the input cursor.

Information about which keys are being pressed is saved to the synchronous status array of keys. This array is connected to the variables for the local input status of each thread. The array of asynchronous key status, which contains similar information, is shared by all threads. The arrays reflect the status of all keys at a given moment, and the GetAsyncKeyState function makes it possible to determine whether or not a specific key is being pressed at a given time. GetAsyncKeyState always returns 0 (i.e. not pressed) if it is called by a different thread (i.e. not the thread which created the window which is currently the focus of input status).

The GetKeyState function differs from GetAsyncKeyState in that it returns the status of the keyboard at the moment when the most recent keyboard message is extracted from the thread queue. This function can be called at any time regardless of which window is currently in focus.

Keyboard hooks

The mechanism used to intercept events using specific functions (e.g. sending Windows messages, data input via the mouse or keyboard) in Microsoft Windows is called 'hooking'. This function can react to an event and, in certain cases, modify or delete events.

Functions which receive notification of events are called filter functions; they differ from each other by which events they can intercept. In order for Windows to call a filter function, the function must be bound to a hook (for instance, to a keyboard hook). Binding one or more filter functions to a hook is called “setting a hook”. An application can use the Win32 API SetWindowsHookEx and UnhookWindowsHookEx functions to set or remove filter functions. Some hooks can either be set across the system as a whole, or for a specific thread.

To avoid conflicts if several filter functions are bound to a single hook, Windows implements a

function queue; in such cases, the function most recently bound to the hook will be at the start of the queue, with the first function bound being at the end of the queue. The function filter queue (see figure 8) is managed by Windows itself, which simplifies the writing of filter functions and optimizes operating system productivity.

The system also supports separate chains for each type of hook. A hook chain is a list of pointers to filter functions (specific callback functions determined by the application.) When an event linked to a particular type of hook takes place, the system consecutively sends the message for each type of filter function to the hook chain. The actions which filter functions may perform depend on the type of hook: some function can only track the appearance of an event, while others may modify message parameters or initiate message processing, by preventing the next filter function in the hook chain from being called, or the message processing function for the relevant window from being called.

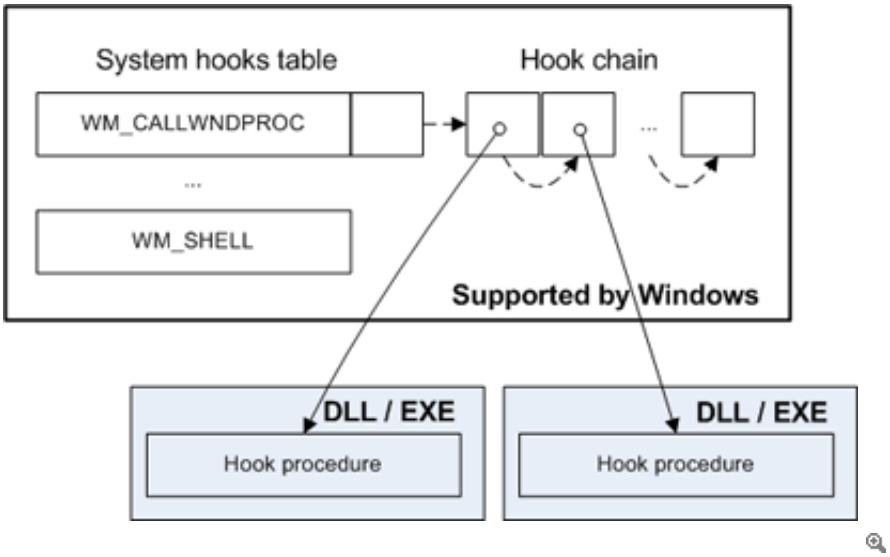

Figure 8. Filter function chain in Windows

When one or more filter functions are bound to a hook and an event takes place which leads to the hook being activated, Windows calls the first function from the filter function queue. With this, its responsibility is over. The filter function is then responsible for calling the next function in the chain, and the Win32 API CallNextHookEx function is used to do this.

The operating system supports several types of hooks, each of which provides access to one aspect of the Windows messaging process mechanism.

Nearly all types of hooks are of potential interest to the creators of keyloggers: WH_KEYBOARD, WH_KEYBOARD_LL (hooking keyboard events when they are added to the thread event queue), WH_JOURNALRECORD and WH_JOURNALPLAYBACK (writing and producing keyboard and mouse events), WH_CBT (intercepting multiple events, including remote keyboard events from the system hardware input queue), WH_GETMESSAGE (intercepting an event from the thread event queue.)

Processing

Let's sum up all the information above on the procedure of keyboard input in a single algorithm: the algorithm of the passing of a signal from a key being pressed by the user to the appearance of symbols on the screen can be presented as follows:

1. When starting, the operating system creates a raw input thread and a system hardware input queue in the csrss.exe process.
2. The raw input thread cyclically sends read requests to the keyboard driver, which remains in a waiting condition until an event from the keyboard appears.
3. When the user presses or releases a key on the keyboard, the keyboard micro-controller detects that a key has been pressed or released and sends both the scan code of the pressed/ released key to the central computer and an interrupt request.
4. The keyboard system controller gets the scan code, processes it then makes it accessible on input/output port 60h and generates a central processor hardware interrupt.
5. The interrupt controller signals the CPU to call the interrupt processing procedure for IRQ1 – ISR, which is registered in the system by the functional keyboard driver i8042prt.
6. The ISR reads the data which has appeared from the internal keyboard controller queue, transforms the scan codes to virtual key codes (independent values which are determined by the system) and queues “I804KeyboardIsrDPC”, a delayed procedure call.
7. As soon as possible, the system calls the DPC which in turn executes the callback procedure KeyboardClassServiceCallback registered by the Kbdclass keyboard driver.
8. The KeyboardClassServiceCallback procedure extracts a pending termination request from the raw input thread from its queue and returns it with information about the key pressed.
9. The raw input thread saves the information to the system hardware input queue and uses it to create the basic Windows keyboard messages WM_KEYDOWN, WM_KEYUP, which are placed at the end of the VIQ virtual input queue of the active thread.
10. The message processing cycle thread deletes the message from the queue and sends the corresponding window procedure for processing. When this happens, the system function TranslateMessage may be called, which uses basic keyboard messages to create the additional “symbol” messages WM_CHAR, WM_SYSCHAR, WM_DEADCHAR and WM_SYSDEADCHAR.

Raw input model

The keyboard input model described above has a number of shortcomings from the point of view of those who develop applications. In order to get input from a non-standard input device, the application has to perform a significant number of operations: mount the device, periodically query the device, consider the possibility of parallel use of the device by other applications etc. For this reason, later versions of Windows offer an alternative input model, called the raw input model, which simplifies the development of applications which use non-standard input devices (see Fig 3, DirectInput applications)

The raw input model differs from the original Windows input model. In the latter, the application gets device independent input in the form of a message (such as (WM_CHAR), which is sent to application windows. In the raw input model, the application has to register the device it wants to receive data from. The application then receives user input via WM_INPUT messages. Two methods of data transmission are supported: a standard method and a buffering method. In order to interpret entered data, the application has to get information about the input device, which can be done using the GetRawInputDeviceInfo function.

Implementing keyloggers: the variants

Now we take a look at the main methods used by malware authors for implementing keyloggers,

taking the model of processing keyboard input in Windows described above as a basis. Knowing how the model is structured makes it easier to understand which mechanisms can be used by keyloggers.

Keyloggers can be injected at any stage in the processing sequence, intercepting data about keys pressed which is transmitted by one processing subsystem to another subsystem. The methods examined below for creating software keyloggers are divided into user mode methods and kernel mode methods. Figure 9 shows all the subsystems processing keyboard input and their interdependencies. Next to some of the subsystems are numbers in red circles, and these indicate the section of the article which describes how keyloggers which use or substitute the corresponding subsystem can be created.

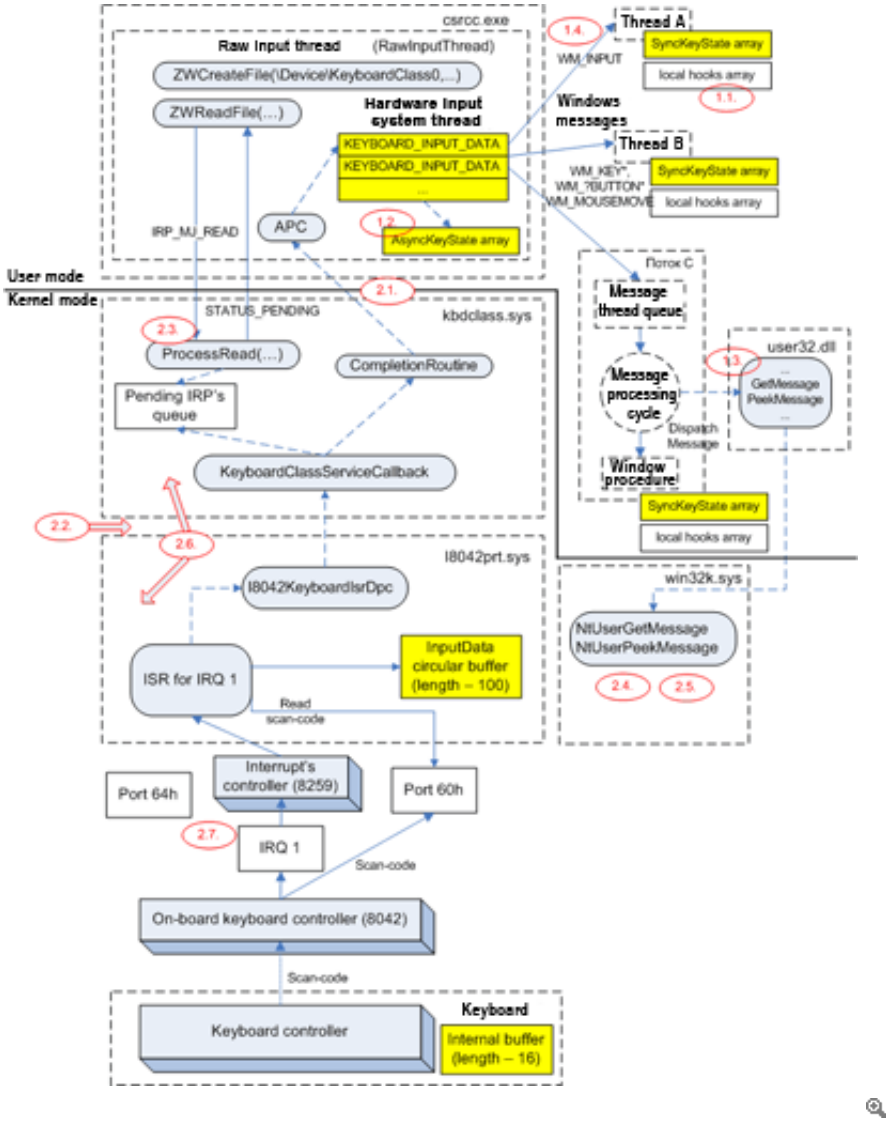

Fig 9: Overview of how Windows processes keyboard input

This section will not look at creating hardware keyloggers. It is enough to note that there are three types of hardware keylogger: keyloggers incorporated into the keyboard itself; keyloggers incorporated into the cable connecting the keyboard to the system block; keyloggers incorporated into the computer's system block itself. The most common of these is the second type of hardware keylogger, and one of the best known examples of this type is the "KeyGhost USB Keylogger" (http://www.keyghost.com).

Unfortunately, at the moment only afew antivirus solutions offer adequate protection against the potential threat posed by keyloggers. Such products are for instance Kaspersky Lab 6.0 products and newer.

1. User mode keyloggers

User mode keyloggers are the easiest to create, but also the easiest to detect, as they use well known and well documented Win32 API functions to intercept data.

1.1. Setting hooks for keyboard messages

This is the most common method used when creating keyloggers. Using SetWindowsHookEx, the keylogger sets a global hook for all keyboard events for all threads in the system (see the section on 'Keyboard hooks'). The hook filter function is located in a separate dynamic library which will be injected into all processes in the system. When any keyboard message thread is extracted from the message queue, the system calls the filter function installed.

One of the advantages of this method of interception is its simplicity and the guaranteed interception of all pressed keys. As for disadvantages, it should be noted that it's necessary to have a separate dynamic library file, and it is also relatively easy to detect the keylogger due to the fact that it has been injected into all processes.

Some keyloggers which use this approach are AdvancedKeyLogger, KeyGhost, Absolute Keylogger, Actual Keylogger, Actual Spy, Family Key Logger, GHOST SPY, Haxdoor, MyDoom and others.

Kaspersky Internet Security proactively detects this type of Keylogger as 'invader (loader); the option 'Windows hooks' in the 'Application activity analyzer' subsystem in the PDM module of KIS should be enabled.

1.2. Using cyclical querying of the keyboard

This is the second most common method used when implementing keyloggers. The GetAsyncnKeyState and GetKeyState are used to periodically query the status of all keys at rapid intervals. This function returns arrays of asynchronous or synchronous key status (see section Keyboard key status array); by analysing these it’s possible to understand which keys have been pressed or released since the last query was carried out.

The advantages of this method are that it is extremely easy to implement, there is no additional module (in contrast to the use of hooks); the disadvantages are that there is no guarantee that all keystrokes will be intercepted – some may be missed; and this method can easily be detected by looking for processes which query the keyboard with high frequency.

This method is used in keyloggers such as Computer Monitor, Keyboard Guardian, PC Activity Monitor Pro, Power Spy, Powered Keylogger, Spytector, Stealth KeyLogger, Total Spy.

Kaspersky Internet Security proactively detects this type of Keylogger as 'Keylogger'; the option 'Keylogger detection' in the 'Application activity analyzer' subsystem in the PDM module should be enabled.

1.3. Injection into processes and hooking message processing functions

This method is used infrequently. The keylogger is injected into all processes and intercepts the

GetMessage or PeekMessage functions from the user32.dll library. A range of methods can be used to do this: splicing (a method used to intercept API function calls; essentially, the method consists of substituting a few - normally five - of the first bytes of the JMP instruction function which passes control to the intercepting code), modifying the address table of IAT import functions, intercepting the GetProcAddress function which returns a function address from the library which has been loaded. A keylogger can be created either in the form of a DLL or by injecting the code directly into the target process.

The result is as follows: when the application calls, for example, the GetMessage function in order to get the next message from the message queue, this call will be passed to the hook code. The hook calls the outgoing GetMessage function from user32.dll and analyses the results returned for message type objects. When a keyboard message is received, all information about it is extracted from the message parameters and logged by the keylogger.

The advantages of this method are its effectiveness: due to the method being not very common, only a few programs are able to detect keyloggers of this type. Additionally, standard onscreen keyboards are useless against this type of keylogger, as the messages sent by them will also be intercepted.

The disadvantages include the fact that modifying the IAT table does not guarantee interception, as the function address of user32.dll may have been saved prior to the keylogger being injected; splicing has certain difficulties connected with, for example, the need to rewrite the body of the function on the fly.

Kaspersky Internet Security proactively detects this type of Keylogger as 'invader'; the option 'Intrusion into process (invaders)' in the 'Application activity analyzer' subsystem in the PDM module should be enabled.

1.4. Using the raw input model

At the time of writing, there is only one known example of this method: a utility used to test how well protected the system is against keyloggers. Essentially the method uses the new Raw Input module (see the section "Raw Input model"), where the keylogger registers the keyboard as a device from which it wants to receive input. The keylogger then starts to get data about keys which have been pressed and released using the WM_INPUT message.

The advantage is that this method is easy and effective to implement; in view of the fact that this method became well known only recently, almost none of today's protection software currently has the ability to combat such keyloggers.

The disadvantage is that fact that such keyloggers are easy to detect due to the need for special registration in order for the application to get raw input (by default, no system processes do this).

Kaspersky Internet Security 7.0 will proactively detect such keyloggers as 'Keylogger'; the option 'Keylogger detection' in the 'Application activity analyzer' subsystem in the PDM module should be enabled.

2. Kernel mode keyloggers

In terms of both implementation and detection, kernel mode keyloggers are significantly more complex than user mode keyloggers. In order to write such keyloggers, higher knowledge is

required, but this makes it possible to create keyloggers which will be completely unnoticed by all user mode applications. Both methods documented in the Microsoft Driver Development Kit and undocumented methods may be used for interception.

2.1. Using the keyboard driver filter Kbdclass

The hook method is documented in the DDK (see the section "Driver stack for Plug and Play PS/2 keyboard"). Keyloggers which use this approach intercept requests to the keyboard by installing a filter above the device “\Device\KeyboardClass0”, created by the Kbdclass driver (see Driver stack for Plub and Play PS/2 keyboards). Only IRP_MJ_READ requests need to be filtered as it is these which make it possible to get the codes of keys which have been pressed and released.

The advantage is the guarantee that all keystrokes will be intercepted; such keyloggers cannot be detected without using a driver, and due to this, not all anti-keyloggers will detect them. The disadvantage is that the driver itself has to be installed.

The best known keyloggers which use this approach are ELITE Keylogger and Invisible KeyLogger Stealth.

Kaspersky Internet Security proactively detects such keyloggers as Keylogger by monitoring the keyboard device stack (the “Detect keyboard intercepts” option in the “Application activity analysis” option in the proactive detection module (PDM) should be enabled).

2.2. Using the filter driver of the i8042prt functional driver

This method of interception is documented in the DDK. Keyloggers based on this method intercept requests to the keyboard by installing a filter on top of an unnamed device, created by the i8042prt driver for the Device\KeyboardClass0 (see section “Device stack for Plug and Play PS/2 keyboards"). The i8042prt driver is a program interface for adding an additional IRQ1 (IsrRoutine) interrupt processing function, within with data received from the keyboard can be analysed.

The advantages and disadvantages of this method are the same as those detailed in the previous point. However, there is an additional disadvantage – it is dependent on the type of keyboard. The i8042prt driver processes requests only from PS/2 keyboards, and because of this, this method is not suitable for intercepting data entered via a USB or wireless keyboard.

Kaspersky Internet Security proactively detects such keyloggers as Keylogger by monitoring the keyboard device stack (the “Detect keyboard intercepts” option in the “Application activity analysis” option in the proactive detection module (PDM) should be enabled).

2.3. Modifying the dispatch table of the Kbdclass driver

Keyloggers based on this principle intercept requests to the keyboard by changing the IRP_MJ_READ entry point in the dispatch table for the Kbdclass driver. This is functionally similar to the Kbdclass driver filter (see section “2.1 Driver filters for kbdclass driver”.) The specifics are the same. Another variant hooks a different function for processing requests: IRP_MJ_DEVICECONTROL. In this case, the keylogger becomes a driver filter similar to the i8042 driver (see section “2.2 Using the filter driver of the i8042prt functional driver”).

Kaspersky Internet Security proactively detects such keyloggers as Keylogger by monitoring the keyboard stack devices (the “Detect keyboard interception” in the “Application activity analysis”

option in the proactive detection module (PDM) should be enabled.)

2.4. Modifying the system service table KeServiceDescriptorTableShadow

This is a relatively common method for implementing keyloggers, which is similar to the user mode method described in section “1.3 Injection into processes and hooking message processing functions”. Keyloggers which use this method intercept requests to the keyboard by patching the entry point for NtUserGetMessage/PeekMessage in the second table of system services (KeServiceDescriptorTableShadow of the win32k.sys driver. Information about keys pressed is transmitted to the keylogger when a call to the GetMessage or PeekMessage function is terminated within a thread.

The advantage of this method is that it is difficult to detect. The disadvantage is that it is difficult to implement (searching for KeServiceDescriptorTableShadow is not an easy task; in addition to this, other drivers may already have patched the entry point in this table) and the need to install a separate driver.

Kaspersky Internet security proactively detects such keyloggers as Keylogger by monitoring the keyboard stack devices (the “Detect keyboard interception” in the “Application activity analysis” option in the proactive detection module (PDM) should be enabled.)

2.5. Modifying the code of the NtUserGerMessage or NtUserPeekMessage function by splicing

This is an extremely rare type of keylogger. Keyloggers which use this method intercept requests to the keyboard by modifying the code of the NtUserGetMessage or NtUserPeekMessage using splicing. These functions are implemented in the system kernel through the win32k.sys driver and are called by corresponding functions in the user32.dll library. As shown in section 1.3 “Injection into process and hooking message processing functions”, these functions make it possible to filter all messages received by applications and get data about the pressing/ releasing of keys from keyboard messages.

The advantage of this method is that it is difficult to detect. The disadvantage is that it is difficult to implement (it’s necessary to rewrite the body of the function on the fly, and is dependent on the operating system version and software installed) and it’s also necessary to install a driver.

2.6. Substituting a driver in the keyboard stack of drivers

This method involves substituting a Kbdclass driver or a low-level keyboard driver with a driver expressly developed for the purpose. The clear disadvantage of this method is that it is difficult to implement, as it is impossible to know in advance which type of keyboard is being used. It is therefore relatively easy to detect driver substitution. As a consequence, this method is almost never used.

2.7. Implementing a handler driver for interrupt 1 (IRQ 1).

This involves writing a kernel mode driver which will hook the keyboard interrupt (IRQ1) and which directly contacts the keyboard input/ output ports (60h, 64h), As this is difficult to implement, and due to the fact that it is not entirely clear how it interacts with the system processor for interrupt IRQ1 (i8042prt.sys), this method remains, at the moment, purely theoretical.

Conclusion

This article surveys the progression of data from a key being pressed by the user to the appearance of symbols on the screen; the links in the chain of processing the signal; and methods for implementing keyloggers which intercept keyboard input at specific stages of the process.

1. The process of keyboard input in Windows is relatively complex, and it's possible to install a hook at any stage. Keyloggers have already been created for some of the stages, while for some they do not yet exist.
2. There is a connection between how common a keylogger is and the difficulty of creating such a keylogger. For instance, the more common methods of interception – such as hooking input events and cyclical querying of the keyboard – are the simplest to implement. Even a person who has only been programming for a week would be able to write a keylogger which uses these methods.
3. The majority of keyloggers currently in existence are relatively simple and can easily be used with malicious intent, with the main aim being to steal confidential information entered via the keyboard.
4. Most important of all: Antivirus companies should protect their users against the threat posed by malicious keyloggers.

As protection mechanisms become more sophisticated, the cybercriminals who create keyloggers will be forced to implement more complex methods using Windows kernel drivers – there are still many unexploited possibilities in this area.