14:05 Anti-Keylogging with Random Noise

by Mike Myers

In PoC||GTFO 12:7, we learned that malware is inherently "drunk," and we can exploit its inebriation. This time, our *entonnoir de gavage* will be filled with random keystrokes instead of single malt.

Gather 'round, neighbors, as we learn about the mechanisms behind the various Windows user-mode keylogging techniques employed by malware, and then investigate a technique for thwarting them all.

Background

Let's start with a primer on the data flow path of keyboard input in Windows.

Figure 2 is a somewhat simplified diagram of the path of a keystroke from the keyboard peripheral device (top left), into the Windows operating system (left), and then into the active application (right). In more detail, the sequence of steps is as follows:

1. The user presses down on a key.

2. The keyboard's internal microcontroller converts key-down activity to a device-specific "scan code," and issues it to keyboard's internal USB device controller.

3. The keyboard's internal USB device controller communicates the scan-code as a USB message to the USB host controller on the host system. The scan code is held in a circular buffer in the kernel.

4. The keyboard driver(s) converts the scan code into a virtual key code. The virtual key code is applied as a change to a real-time system-wide data struct called the Async Key State Array.

5. Windows OS process `Csrcc.exe` reads the input as a virtual key code, wraps it in a Windows "message," and delivers it to the message queue of the UI thread of the user-mode application that has keyboard focus, along with a time-of-message update to a per-thread data struct called the Sync Key State Array.

6. The user application's "message pump" is a small loop that runs in its UI thread, retrieving Windows messages with `GetMessage()`, translating the virtual key codes into usable characters with `TranslateMessage()`, and finally sending the input to the appropriate callback function for a particular UI element (also known as the "Window proc") that actually does something with the input (displays a character, moves the caret, *etc.*).

For more detail, official documentation of Windows messages and Windows keyboard input can be found in MSDN MS632586 and MS645530.

User-Mode Keylogging Techniques in Malware

Malware that wants to intercept keyboard input can attempt to do so at any point along this path. However, for practical reasons input is usually intercepted using hooks within an application, rather than in the operating system kernel. The reasons include: hooking in the kernel requires Administrator privilege (including, today, a way to meet or circumvent the driver code-signing requirement); hooking in the kernel before the keystroke reaches the keyboard driver only obtains a keyboard device-dependent "scan code" version of the keystroke, rather than its actual character or key value; hooking in the kernel after the keyboard driver but before the application obtains only a "virtual key code" version of the keystroke (contextual with regard to the keyboard "layout" or language of the OS); and finally, hooking in the kernel means that the malware doesn't know which application is receiving the

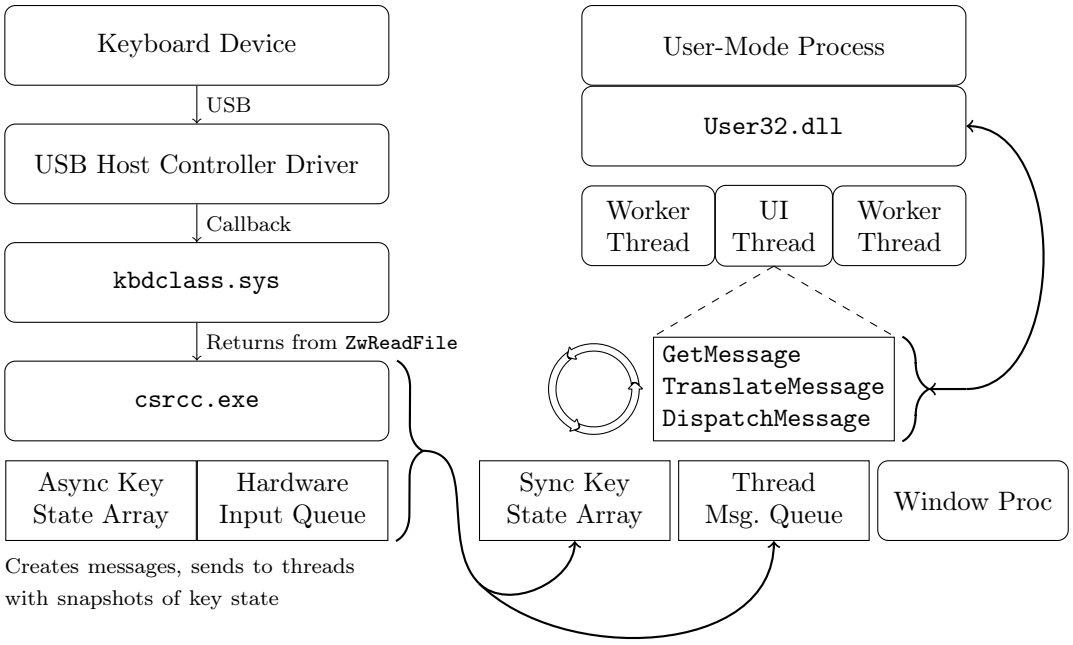

Figure 2. Data flow of keyboard input in Windows.

keyboard input, because the OS has not yet dispatched the keystrokes to the active/focused application. This is why, practically speaking, malware only has a handful of locations where it can intercept keyboard input: upon entering or leaving the system message queue, or upon entering or leaving the thread message queue.

Now that we know the hooking will likely be in user-mode, we can learn about the methods to do user-mode keystroke logging, which include:

- Hooking the Windows message functions `TranslateMessage()`, `GetMessage()`, and `PeekMessage()` to capture a copy of messages as they are retrieved from the per-thread message queue.
- Creating a Windows message hook for the `WH_KEYBOARD` message using `SetWindowsHookEx()`.
- Similarly, creating a Windows message hook for the so-called "LowLevel Hook" (`WH_KEYBOARD_LL`) message with `SetWindowsHookEx()`.
- Similarly, creating a Windows message hook for `WH_JOURNALRECORD`, in order to create a Journal Record Hook. Note: this method has been disabled since Windows Vista.
- Polling the system with `GetAsyncKeyState()`.
- Similarly, polling the system with `GetKeyboardState()` or `GetKeyState()`.
- Similarly, polling the system with `GetRawInputData()`.
- Using DirectX to capture keyboard input (somewhat lower-level method).
- Stealing clipboard contents using, *e.g.*, `GetClipboardData()`.
- Stealing screenshots or enabling a remote desktop view (multiple methods).

The following table lists some pieces of malware and which method they use.

Malware	Keylogging Technique
Zeus	Hooks `TranslateMessage()`, `GetMessage()`, `PeekMessage()`, and `GetClipboardData()`; uses `GetKeyboardState()`.[12]
Sality	`GetMessage()`, `GetKeyState()`, `PeekMessage()`, `TranslateMessage()`, `GetClipboardData()`.
SpyEye	Hooks `TranslateMessage()`, then uses `GetKeyboardState()`.
Poison Ivy	Polls `GetKeyboardLayout()`, `GetAsyncKeyState()`, `GetClipboardData()`, and uses `SetWindowsHookEx()`.
Gh0st RAT	Uses `SetWindowsHookEx()` with `WH_GETMESSAGE`, which is another way to hook `GetMessage()`.

Anti-Keylogging with Keystroke Noise

One approach to thwarting keyloggers that might seem to have potential is: Insert so many phantom keyboard devices into the system that the malware cannot reliably select the actual keyboard device for keylogging. However, based upon our new understanding of how common malware implements keylogging, it is clear that this approach will not be successful, because malware does not capture keyboard input by reading it directly from the device. Malware is designed to intercept the input at a layer high enough as to be input device agnostic. We need a different technique.

Our idea is to generate random keyboard activity "noise" emanating at a low layer and removed again in a high layer, so that it ends up polluting a malware's keylogger log, but does not actually interfere at the level of the user's experience. Our approach, shown in Figure 3, is illustrated as a modification to the previous diagram.

Technical Approach

What we have done is create a piece of dynamically loadable code (currently a DLL) which, once loaded, checks for the presence of `User32.dll` and hooks its imported `DispatchMessage()` API. From the DispatchMessage hook, our code is able to filter out keystrokes immediately before they would otherwise be dispatched to a Window Proc. In other words, keystroke noise can be filtered here, at a point after potential malware would have already logged it. The next step is to inject the keystroke noise: our code runs in a separate thread and uses the `SendInput()` API to send random keystroke input that it generates. These keystrokes are sent into the keyboard IO path at a point before the hooks typically used by keylogging malware.

In order avoid sending keystroke noise that will be delivered to a different application and therefore not filtered, our code must also use the `SetWindowsHookEx()` API to hook the WindowProc, in order to catch the messages that indicate our application is the one with keyboard focus. `WM_SETFOCUS` and `WM_KILLFOCUS` messages indicate gaining or losing keyboard input focus. We can't catch these messages in our `DispatchMessage()` hook because, unlike keyboard, mouse, paint, and timer messages, the focus messages are not posted to the message queue. Instead they are sent directly to WindowProc. By coordinating the focus gained/lost events with the sending of keystroke noise, we prevent the noise from "leaking" out to other applications.

Related Research

In researching our concept, we found some prior art in the form of a European academic paper titled NoisyKey.[13] They did not release their implementation, though, and were much more focused on a statistical analysis of the randomness of keys in the generated noise than in the noise channel technique itself. In fact, we encountered several technical obstacles never mentioned in their paper. We also discovered a commercial product called KeystrokeInterference. The trial version of KeystrokeInterference definitely defeated the keylogging methods we tested it against, but it did not appear to actually create dummy keystrokes. It seemed to simply cause keyloggers to gather incomplete data—depending on the method, they would either get nothing at all, only the Enter key, only punctuation, or they would get all of the keystroke events but only the letter "A" for all of them. Thus, KeystrokeInterference doesn't

[12] Zeus's keylogging takes place only in the browser process, and only when Zeus detects a URL of interest. It is highly contextual and configured by the attacker.

[13] *NoisyKey: Tolerating Keyloggers via Keystrokes Hiding* by Ortolani and Crispo, Usenix Hotsec 2012

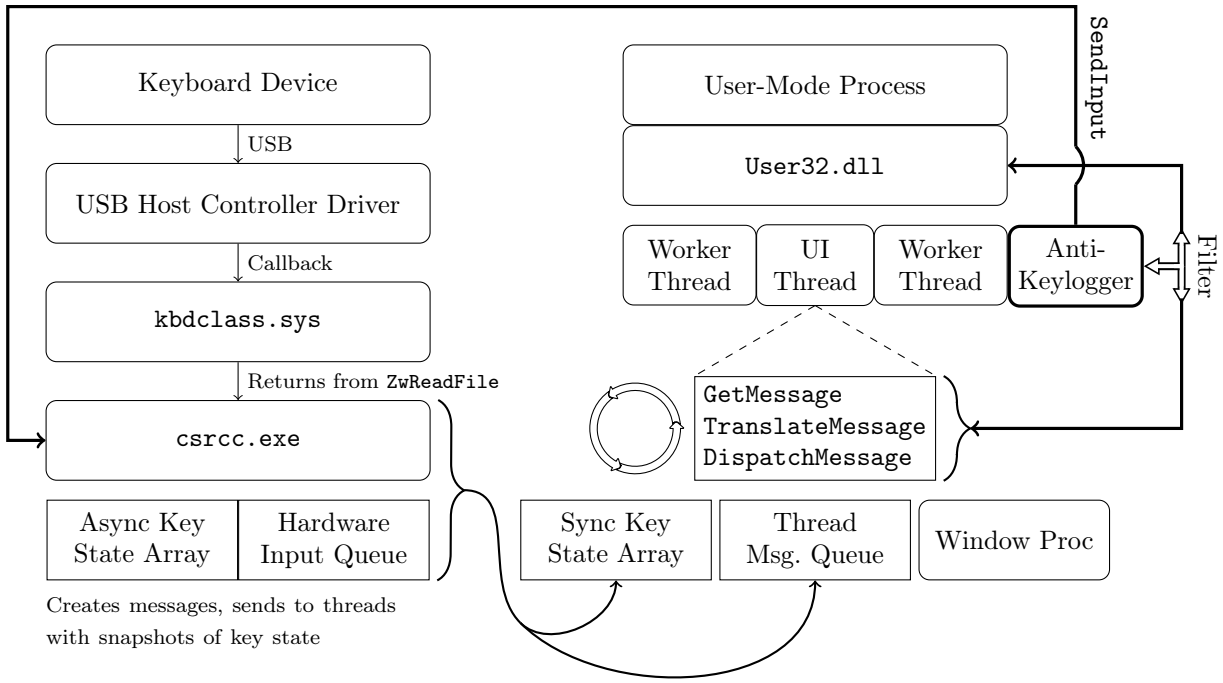

Figure 3. A noise generating anti-keylogger plugged into the Windows keyboard data flow.

obfuscate the typing dynamics, and it appears to have a fundamentally different approach than we took. (It is not documented anywhere what that method actually is.)

Challenges

For keystroke noise to be effective as interference against a keylogger, the generated noise should be indistinguishable from user input. Three considerations to make are the rate of the noise input, emulating the real user's typing dynamics, and generating the right mix of keystrokes in the noise.

Rate is fairly simple: the keystroke noise just has to be generated at a high enough rate that it well outnumbers the rate of keys actually typed by the user. Assuming an expert typist who might type at 80 WPM, a rough estimate is that our noise should be generated at a rate of at least several times that. We estimated that about 400 keystrokes per minute, or about six per second, should create a high enough noise to signal ratio that it is effectively impossible to discern which keys were typed. The goal here is to make sure that random noise keys separate all typed characters sufficiently that no strings of typed characters would appear together in a log.

Addressing the issue of keystroke dynamics is more complicated. Keystroke dynamics is a term that refers to the ability to identify a user or what they are typing based only on the rhythms of keyboard activity, without actually capturing the content of what they are typing. By flooding the input with random noise, we should break keystroke rhythm analysis of this kind, but only if the injected keystrokes have a random rhythm about them as well. If the injected keystrokes have their own rhythm that can be distinguished, then an attacker could theoretically learn to filter the noise out that way. We address this issue by inserting a random short delay before every injected keystroke. The random delay interval has an upper bound but no lower bound. The delay magnitude here is related to the rate of input described previously, but the randomness within a small range should mean that it is difficult or impossible to distinguish real from injected keystrokes based on intra-keystroke timing analysis.

Another challenge was detecting when our application had (keyboard) input focus. It is non-trivial for a Windows application to determine when its

window area has been given input focus: although there are polling-based Windows APIs that can possibly indicate which Window is in the foreground (`GetActiveWindow`, `GetForegroundWindow`), they are not efficient nor sufficient for our purposes. The best solution we have at the moment is that we installed a "Window Proc" hook to monitor for `WM_SETFOCUS` and other such messages. We also found it best to temporarily disable the keystroke noise generation while the user was click-dragging the window, because real keyboard input is not simultaneously possible with dragging movements. There are likely many other activation and focus states that we have not yet considered, and which will only be discovered through extensive testing.

Lastly, we had to address the need to generate keystroke noise that included all or most of the keys that a user would actually strike, including punctuation, some symbols, and capital letters. This is where we encountered the difficulty with the Shift key modifier. In order to create most non-alphanumeric keystrokes (and to create any capital letters, obviously), the Shift key needs to be held in concert with another key. This means that in order to generate such a character, we need to generate a Shift key down event, then the other required key down and up events, then a Shift key up event. The problem lies in the fact that the system reacts to our injected shift even if we filter it out: it will change the capitalization of the user's actual keystrokes. Conversely, the user's use of the Shift key will change the capitalization of the injected keys, and our filter routine will to fail recognize them as the ones we recently injected, allowing them through instead.

The first solution we attempted was to track every time the user hit the Shift key and every time we injected a Shift keystroke, and deconflict their states when doing our filter evaluation. Unfortunately, this approach was prone to failure. Subtle race conditions between Async Key State ("true" or "system" key state, which is the basis of the Shift key state's affect on character capitalization) and Sync Key State ("per-thread" key state, which is effectively what we tracked in our filter) were difficult to debug. We also discovered that it is not possible to directly set and clear the Shift state of the Async Key State table using an API like `SetKeyboardStateTable()`.

We considered using `BlockInput()` to ignore the user's keyboard input while we generated our own, in order to resolve a Shift state confusion. However, in practice, this API can only be called from a High Integrity Level process (as of Windows Vista), making it impractical. It would probably also cause noticeable problems with keyboard responsiveness. It would not be acceptable as a solution.

Ultimately, the solution we found was to rely on a documented feature of `SendInput()` that will guarantee non-interleaving of inputs. Instead of calling `SendInput()` four times (Shift down, key down, key up, Shift up) with random delays in between, we would instead create an array of all four key events and call `SendInput` once. `SendInput()` then ensures that there are no other user inputs that intermingle with your injected inputs, when performed this way. Additionally, we use `GetAsyncKeyState()` immediately before SendInput in order to track the actual Shift state; if Shift were being held down by the user, we would not also inject an interfering Shift key down/up sequence. Together, these precautions

solved the issue with conflicting Shift states. However, this has the downside of taking away our ability to model a user's key-down-to-up rhythms using the random delays between those events as we originally intended.

Once we had made the change to our use of `SendInput()`, we noticed that these injected noise keys were no longer being picked up by certain methods of keylogging! Either they would completely not see the keystroke noise when injected this way, or they saw some of the noise, but not enough for it to be effective anymore. What we determined was happening is that certain keylogging methods are based on polling for keyboard state changes, and if activity (both a key down and its corresponding key up) happens in between two subsequent polls, it will be missed by the keylogger. When using SendInput to instantaneously send a shifted key, all four key events (Shift key down, key down, key up, Shift key up) pass through the keyboard IO path in less time than a keylogger using a polling method can detect (at practical polling rates) even though it is fast enough to pick up input typed by a human. Clearly this will not work for our approach. Unfortunately, there is no support for managing the rate or delay used by `SendInput`; if you want a key to be "held" for a given amount of time, you have to call `SendInput` twice with a wait in between. This returns us to the problem of user input being interleaved with our use of the Shift key.

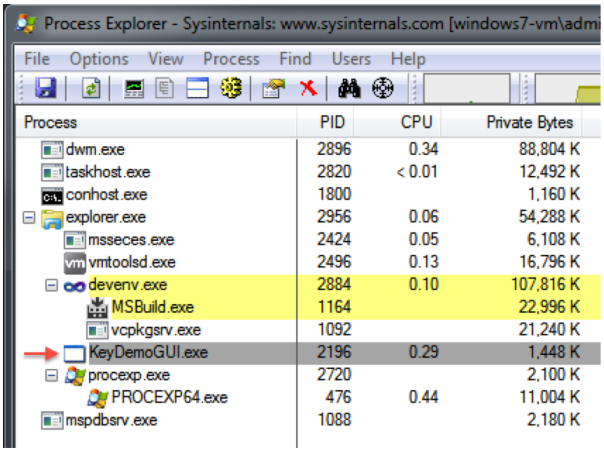

Figure 4. CPU and RAM usage of the PoC keystroke noise generator.

Our compromise solution was to put back our multiple `SendInput()` calls separated by delays, but only for keys that didn't need Shift. For keys that need Shift to be held, we use the single `SendInput()` call method that doesn't interleave the input with user input, but which also usually misses being picked up by polling-based keyloggers. To account for the fact that polling-based keyloggers would receive mostly only the slower unshifted key noise that we generate, we increased the noise amount proportionately. This hybrid approach also enables us to somewhat model keystroke dynamics, at least for the unshifted keystrokes whose timing we can control.

PoC Results

Our keystroke noise implementation produces successful results as tested against multiple user-mode keylogging methods.

Input-stealing methods that do not involve keylogging (such as screenshots and remote desktop) are not addressed by our approach. Fortunately, these are far less attractive methods to attackers: they are high-bandwidth and less effective in capturing all input. We also did not address kernel-mode keylogging techniques with our approach, but these too are uncommon in practical malware, as explained earlier.

Because the keystroke noise technique is an *active* technique (as opposed to a passive configuration change), it was important to test the CPU overhead incurred. As seen in Figure 4, the CPU overhead is incredibly minimal: it is less than 0.3% of one core of our test VM running on an early 2011 laptop with a second generation 2GHz Intel Core i7. Some of that CPU usage is due to the GUI of the demo app itself. The RAM overhead is similarly minimal; but again, what is pictured is mostly due to the demo app GUI.

Conclusions

Although real-time keyboard input is effectively masked from keyloggers by our approach, we did not address clipboard-stealing malware. If a user were to copy and paste sensitive information or credentials, our current approach would not disrupt malware's ability to capture that information. Similarly, an attacker could take a brute-force approach of capturing what the user sees, and grab keyboard input that way (screenshotting or even a live remote desktop session). For approaches like these, there are other techniques that one could use. Perhaps they would be similar to the keystroke noise concept (*e.g.*, introduce noise into the display output channel, filter it out at a point after malware tries to grab it), but that is research that remains to be done.

Console-mode applications don't rely on Windows messages, and as such, our method is not yet compatible with them. Console mode applications retrieve keyboard input differently, for example using the `kbhit()` and `getkey()` APIs. Likewise, any Windows application that checks for keyboard input without any use of Windows Messages (rare, but theoretically possible), for example by just polling `GetKeyboardState()`, is also not yet compatible with our approach. There is nothing fundamentally incompatible; we would just need to instrument a different set of locations in the input path in order to filter out injected keyboard input before it is observed by console-mode applications or "abnormal" keyboard state checking of this sort.

Another area for further development is in the behavior of `SendInput()`. If we reverse engineer the `SendInput` API, we may be able to reimplement it in a way specifically suited for our task. Specifically we would like the timing between batched input elements to be controllable, while maintaining the input interleaving protection that it provides when called using batched input.

We discovered during research that a "low-level keyboard hook" (`SetWindowsHookEx()` with `WH_KEYBOARD_LL`) can check a flag on each callback called `LLKHF_INJECTED`, and know if the keystroke was injected in software, *e.g.*, by a call to `SendInput()`. So in the future we would also seek a way to prevent `win32k.sys` from setting the `LLKHF_INJECTED` flag on our injected keystrokes. This flag is set in the kernel by `win32k.sys!XxxKeyEvent`, implying that it may require kernel-level code to alter this behavior. Although this would seem to be a clear way to defeat our approach, it may not be so. Although we have not tested it, any on-screen keyboard or remotely logged-on user's key inputs supposedly come through the system with this flag set, so a keylogger may not want to filter on this flag. Once we propose loading kernel code to change a flag, though, we may as well change our method of injecting input and just avoid this problem entirely. By so doing we could also likely address the problem of kernel-mode keyloggers.

Acknowledgments

This work was partially funded by the Halting Attacks Via Obstructing Configurations (HAVOC) project under Mudge's DARPA Cyber Fast Track program, Digital Operatives IR&D, and our famous Single Malt Gavage Funnel. With that said, all opinions and hyperbolic, metaphoric, gastronomic, trophic analogies expressed in this article are the author's own and do not necessarily reflect the views of DARPA or the United States government.