

Forum:[Hack Forums](#)

Thread: [\[DISCUSSION\] High-Level Hooks vs Polling: Input Monitoring Architecture](#)

psycheuser - August 23, 2026, 5:49 am

When developing input monitoring tools or accessibility frameworks on Windows, developers generally choose between two primary methods for capturing keyboard states: High-Level Hooks or direct API polling.

Using Windows Hooks via the `SetWindowsHookEx` API with a `WH_KEYBOARD_LL` parameter allows the application to intercept input events globally before they reach the target window. This event-driven approach is highly efficient because it consumes minimal CPU cycles. However, because it hooks directly into the OS user subsystem, it can easily be detected by behavioral monitoring software that watches for global hook installations.

The alternative method is polling using `GetAsyncKeyState` inside a continuous execution loop. This checks the physical state of the keys directly. While it completely bypasses the need to install a conspicuous global hook, it is highly resource-intensive, often spiking CPU utilization if the polling interval isn't properly optimized.

For the application developers here, do you find event-driven hooks more reliable for input logging, or do you prefer optimized polling methods to maintain a lower stealth profile?

closedeyeknight - August 29, 2026, 6:14 am

`SetWindowsHookEx` installs a callback. The callback will be triggered when `WM_KEYDOWN`/`WM_KEYUP`/`WM_SYSKEYDOWN`/`WM_SYSKEYUP` comes.

Check MSDN's introduction of `WH_KEYBOARD_LL`, u will see this:

Code `LRESULT CALLBACK LowLevelKeyboardProc(`

`_In_ int nCode,`

`_In_ WPARAM wParam,`

`_In_ LPARAM lParam`

`);` This is the prototype of callback. And the core param: `LPARAM` is defined as:

Code `typedef struct tagKBDLLHOOKSTRUCT {`

`DWORD vkCode;`

`DWORD scanCode;`

`DWORD flags;`

`DWORD time;`

`ULONG_PTR dwExtraInfo;`

`} KBDLLHOOKSTRUCT, *LPKBDLLHOOKSTRUCT, *PKBDLLHOOKSTRUCT;` Most important segment is:

Code `DWORD vkCode;`

And `GetAsyncKeyState` uses for checking the key pressed. `GetKeyState` for toggled key checking (Like CapsLocks etc).

This flow chart is like:

Key pressed -> Hook captured -> callback called -> virtual key code checking (`GetAsyncKeyState + GetKeyState`) + ASCII

psycheuser - August 29, 2026, 8:30 am

(Aug 29, 2026 06:14 AM) closedeyeknight Wrote: [\[CITE\]](#) `SetWindowsHookEx` installs a callback.

The callback will be triggered when

WM_KEYDOWN/WM_KEYUP/WM_SYSKEYDOWN/WM_SYSKEYUP comes.

Check MSDN's introduction of WH_KEYBOARD_LL, u will see this:

Code LRESULT CALLBACK LowLevelKeyboardProc(

In int nCode,

In WPARAM wParam,

In LPARAM lParam

); This is the prototype of callback. And the core param: LPARAM is defined as:

Code typedef struct tagKBDLLHOOKSTRUCT {

DWORD vkCode;

DWORD scanCode;

DWORD flags;

DWORD time;

ULONG_PTR dwExtraInfo;

} KBDLLHOOKSTRUCT, *LPKBDLLHOOKSTRUCT, *PKBDLLHOOKSTRUCT; Most important

segment is:

Code DWORD vkCode;

And GetAsyncKeyState uses for checking the key pressed. GetKeyState for toggled key checking (Like CapsLocks etc).

This flow chart is like:

Key pressed -> Hook captured -> callback called -> virtual key code checking (GetAsyncKeyState + GetKeyState) + ASCII

Appreciate the technical breakdown. Breaking down LowLevelKeyboardProc and pointing out the KBDLLHOOKSTRUCT layout hits the core of how the message pump processes those low-level events. That distinction between handling the raw vkCode via the callback versus evaluating modifier and toggle states down the line with GetKeyState is spot on. If you are chaining hooks like this in a live environment, how do you usually handle timeout mitigation or prevent the OS from dropping your hook when the message queue gets heavily congested during high-throughput injection or interception?

Clifftonn - August 29, 2026, 4:04 pm

for reliability, hooks win — event-driven, no missed keystrokes, lower CPU. for stealth, polling wins — no global hook installation, no behavioral signature.

the real answer is context. if you're on a machine with CrowdStrike or SentinelOne, SetWindowsHookEx gets flagged in seconds. polling from a legitimate parent process with a sane interval (50-100ms) flies under the radar longer.

hybrid approach: start with hooks for the initial capture window, then switch to polling once you detect an EDR scan or behavioral trigger. best of both, adaptive to the environment.

to your question directly — event-driven hooks are more reliable. but optimized polling (not a tight loop, throttled, from a trusted process) is more survivable. depends what you're optimizing for.

psycheuser - August 29, 2026, 4:06 pm

(Aug 29, 2026 04:04 PM)Clifftonn Wrote: [\[CITE\]](#) for reliability, hooks win — event-driven, no missed keystrokes, lower CPU. for stealth, polling wins — no global hook installation, no behavioral signature.

the real answer is context. if you're on a machine with CrowdStrike or SentinelOne, SetWindowsHookEx gets flagged in seconds. polling from a legitimate parent process with a sane interval (50-100ms) flies under the radar longer.

hybrid approach: start with hooks for the initial capture window, then switch to polling once you detect an EDR scan or behavioral trigger. best of both, adaptive to the environment.

to your question directly — event-driven hooks are more reliable. but optimized polling (not a tight loop, throttled, from a trusted process) is more survivable. depends what you're optimizing for.

For reliability, hooks win completely because they are event-driven and avoid the dropped inputs that happen with polling intervals. But the point about EDR visibility is spot on—dropping `SetWindowsHookEx` on a machine running CrowdStrike or SentinelOne will light up alerts immediately.

That hybrid idea of adapting based on the environment or falling back when behavioral triggers show up is a solid approach to balancing reliability with survivability against modern endpoint protection.