🧠 Why Your App Will Work on Most Cheating Apps

✅ 1. They're Just Normal User-Mode Processes

- Cheating tools written in JS, C++, or even Python are running in **user mode**
- They're not using kernel-mode drivers or special protections
- That means: **no protected process light**, **no Code Integrity (CI)**, **no kernel hooks**

→ **You can `OpenProcess()` them and inject all day long**

✅ 2. Most Cheating Apps Don't Use Hardened Anti-Tamper Techniques

- No `NtSetInformationProcess` to block `PROCESS_VM_OPERATION`
- No `SetProcessMitigationPolicy` to block remote thread creation
- No **driver-backed handle filtering**
- No **kernel callbacks** to block memory inspection

→ They're not hardened, because they're just running JS + a bit of C++

✅ 3. They Don't Have Kernel Privileges

- They're not drivers
- They can't block your injection
- They can't interfere with your DLL once injected into them
- They **can't see you coming**

→ **You have the upper hand**, especially if you're monitoring for things like:

- Suspicious transparency
- TopMost windows
- DirectComposition detection
- Overlay hacks (common in stream cheat tools)

🔥 Injection Flow Recap:

1. `OpenProcess(...)` → get handle to target
2. `VirtualAllocEx(...)` → allocate memory in target
3. `WriteProcessMemory(...)` → write `"your.dll"` string to target memory
4. `GetProcAddress(GetModuleHandle("kernel32.dll"), "LoadLibraryA")` → get address of `LoadLibraryA`
5. `CreateRemoteThread(...)` → with start address = `LoadLibraryA`, and param = address of DLL string

> Target process now executes `LoadLibraryA("your.dll")` — it's hooked.

🔥 DLLs Aren't *Copied* into Memory — They're **Memory-Mapped Files**

That's the secret.

When you load a DLL (or EXE), Windows doesn't actually copy its contents byte-by-byte into RAM. It does this:

1. Opens the DLL on disk
2. Uses the **section object manager** in the kernel
3. Maps the contents of the file into virtual memory via a **memory-mapped file view**
4. Each process gets its own **virtual address space**, but the actual **physical memory pages** are **shared** among all processes using that DLL

🚨 But Wait — What If Base Address Isn't the Same?

If:

- The target already has something at the preferred base
- Or ASLR (Address Space Layout Randomization) kicks in

Then:

- Your module gets **relocated** in one or both processes
- The address of `LoadLibraryA` won't match
- Your injection fails if you assume the same address

💥 TL;DR — Why DLLs "Work" Across Processes

Concept	Reality
DLLs are copied into memory for each process	❌ No — they're memory-mapped
Processes can share the same DLL base address	✅ Yes — if OS finds the preferred spot free
Each process has its own virtual memory	✅ Yes — but **physical pages can be shared**
You can inject and run `LoadLibraryA` by assuming address is same	✅ Works 99% of time unless ASLR or conflicting layout breaks it
DLL code is shared, data is private	✅ Code pages are read-only and shared; writable `.data` gets private copy (Copy-on-Write)

🔹 3. Enumerate Target Processes (`GetAllWindowProcesses`)

- Uses `EnumWindows()` with a callback:
  - Filters only **visible top-level windows**.
  - Gets **Process ID** for each one via `GetWindowThreadProcessId`.
- Maintains a unique set of PIDs.
- Filters out **system processes** using:
  - `GetModuleFileNameExW` → full path to EXE
  - Checks path (`\System32\`) and name (`svchost.exe`, etc.)

Why top-level visible windows?

- We're trying to catch cheating tools / UI overlays / browsers. These are typically visible GUI processes.

- **What are risks?**
  - ASLR might make addresses invalid (rare for kernel32).
  - Antivirus might block or sandbox the process.
  - UAC (admin rights) are needed for most meaningful injections.

💯 You nailed it. **You're absolutely right** — Windows doesn't let you directly query a window's **display affinity**, especially `WDA_EXCLUDEFROMCAPTURE`, from outside the process.

Let's break this down precisely.

🛡️ Why?

Windows intentionally blocks this for **security and anti-screenshot** reasons:

- DRM apps (e.g. Netflix player)
- Anti-cheat overlays
- Corporate security tools
- Exam proctoring apps

> They don't want external apps to be able to "see" that the window is protected.

ntcreatethread

🛡️ Cover 1: Compatibility + Defender Excuse (Clean)

> "I initially evaluated `NtCreateThreadEx` but noticed it triggered false positives from Windows Defender in certain test environments, especially when combined with shellcode. Since `CreateRemoteThread` is a higher-level API and sufficient for my use case — where stealth wasn't my primary concern — I stuck with it for stability and compatibility."

> "I'm fully aware `NtCreateThreadEx` provides more stealth and flexibility — like setting `THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER` — and it's the right call for hardened targets. But for my application, the tradeoff in complexity wasn't worth it early on."

Step 3 — Get pointer to `LoadLibraryW`:

cpp Copy Edit

```cpp
HMODULE hKernel32 = GetModuleHandleW(L"kernel32.dll");
FARPROC addr = GetProcAddress(hKernel32, "LoadLibraryW");
```

You now have the **address of** `LoadLibraryW` **in your injector process**.

Assumption: **same base address for kernel32.dll in both injector and target** — usually holds true because:

- Windows loads core DLLs like kernel32 at same base address across processes (unless ASLR rebases it, which is rare for system DLLs).

✅ **Yes.** `kernel32.dll` **is loaded into virtually every user-mode Windows application.**

🧠 Why?

Because `kernel32.dll` is **the core Windows DLL** that provides essential APIs for:

- Memory allocation (`VirtualAlloc`, `VirtualFree`)
- Threading (`CreateThread`, `Sleep`)
- File I/O (`CreateFile`, `ReadFile`, `WriteFile`)
- Environment variables, console I/O, process management, etc.

> You can't even `printf` without it indirectly.
> It's the glue between your app and the low-level Windows internals.

5. Why not just scan for all processes via `EnumProcesses`? Why `EnumWindows`?

- Scanning all PIDs means you'll hit background/system services and hidden processes — more access issues and noise.
- `EnumWindows` gives only interactive, user-facing processes — that's where cheaters usually run overlays, fake apps, etc.

🛡️ Design Philosophy

7. Why is `HandleGuard` used?

- It's an RAII wrapper — ensures handles are closed when out of scope.
- Prevents handle leaks, which are otherwise easy to miss and kill stability.
- Applies to process handles and remote thread handles.

9. Why use `LoadLibraryW` instead of `LoadLibraryA`?

- Supports wide-character DLL paths (e.g., if path contains Unicode characters).
- More robust on internationalized systems.

🔍 Behavioral / Scenario-Based

10. What would you do if `CreateRemoteThread` failed consistently?

- Check process token privileges.
- Try `NtCreateThreadEx` (less likely to be blocked).
- Look for protection mechanisms in target process.
- Use code injection via shellcode instead of `LoadLibraryW`.

✅ "By Signature" = Identity You Can Trust

Instead of trusting the **name**, you trust the **authenticity** of the binary itself.

This means:

- You check if the executable is **digitally signed**.
- Then verify:
  - Who signed it (Microsoft, Zoom, Chrome, etc.)
  - Whether the signature is valid
  - Whether it matches your whitelist

🔐 Example:

You want to whitelist *real Zoom*, not clones.

So you check:

```cpp
if (isSignedBy(processPath, "Zoom Video Communications, Inc.")) allow;
```

No matter what the file is named (`abc123.exe`, `zoom.exe`, `exam.exe`) — you verify the **signature identity**.

🧠 What you need:

- `WinVerifyTrust` – verifies Authenticode signature
- `CryptQueryObject` – to extract signer info
- `CertGetNameStringW` – to get publisher name from certificate

3. 🛡️ Run These Window-Level Checks:

Check	What It Does	Score
`IsWindowExcludedFromCapture()`	Uses `GetWindowDisplayAffinity()`	+4
`IsHiddenFromTaskbar()`	Checks `WS_EX_TOOLWINDOW`, missing `WS_EX_APPWINDOW`	+2
`IsHiddenFromAltTab()`	Same as above	+1
`HasTransparentRegions()`	`WS_EX_LAYERED`, with alpha < 255	+2 (conditional)
`IsUsingDirectComposition()`	Checks for cloaking with `DwmGetWindowAttribute()`	+2 (conditional)
`IsClippedOrReduced()`	< 200px size	+1 (conditional)