GhostSys: Stealthy Post-CET Syscall Evasion for Windows 11

Samuel ‘tlsbollei” Varga
Independent Security Researcher
tlsbollei@gmail.com

Version 2.0 — July 2025

1 Abstract

Abstract

Windows 11 now pairs hardware-enforced *Control-flow Enforcement Technology* (CET) with aggressive endpoint monitoring, reducing the effectiveness of classic direct-syscall tricks. We introduce *GhostSys*, a suite of five techniques that jointly achieve $> 98\%$ *syscall coverage* while triggering *zero* detections across three commercial EDRs. Our methods require no `.text` mutation, no RWX pages and remain fully CET-compliant. We supply empirical evidence—complete with confidence intervals and Intel shadow-stack traces—demonstrating their practicality, and we propose concrete mitigations to help defenders close the newly exposed gaps.

Contents

2 Introduction

Direct system-call (*syscall*) invocation is a linchpin of modern red-team tradecraft. By bypassing high-level Win32 APIs, attackers evade user-mode hooks that security products place in `ntdll.dll`. Historically, tools like *SysWhispers* [1] automated the generation of syscall stubs, and frameworks such as *D/Invoke* [2] loaded fresh copies of DLLs to call unhooked native APIs. Two recent shifts now threaten this strategy:

1. **Hardware Control-Flow Integrity**. Intel CET (Shadow Stacks and Indirect Branch Tracking) and AMD Shadow Stack enforce a parallel call/return stack, breaking conventional inline stubs and ROP chains.

2. **Advanced EDR Telemetry**. Products such as Defender for Endpoint, CrowdStrike Falcon and SentinelOne now combine inline hooking, *hook shadowing* and rapid memory snapshots for heuristic analysis.

We ask: *can an attacker still execute arbitrary syscalls, at scale, without violating CET or tripping modern EDRs?* GhostSys answers "*yes*" via five synergistic techniques.

Contributions. Our work:

1. Formulates a post-CET threat model (§3) capturing realistic defender and attacker capabilities.

2. Presents five novel evasion techniques (§§4–8) and releases open-source PoCs for each.

3. Supplies a rigorous evaluation with statistical variance analysis and CET MSR traces (§9).

4. Outlines deployable mitigations for defenders (§10), informed by hardware performance counters and integrity checks.

5. Provides an *artifact package* meeting USENIX repeatability guidelines (§13), including code, a VM environment, and reproduction instructions.

3 Threat Model and Assumptions

3.1 Defender Capabilities

We assume the defender:

- Hooks all documented syscalls in `ntdll.dll` and employs *hook shadowing*.
- Enforces CET (shadow stack ON) on Intel Tiger Lake or AMD Zen 4 CPUs.
- Captures user-mode memory for offline forensic YARA scanning.
- Logs syscalls via ETW, kernel shims or proprietary drivers.
- Enables Virtualisation-Based Security (VBS) and HyperGuard.

3.2 Attacker Goals & Constraints

The attacker possesses arbitrary user-mode code execution and seeks to:

- Invoke any native syscall while bypassing user-mode API hooks.
- Remain CET-compliant (no shadow-stack violations).
- Avoid writable-executable (RWX) pages and long ROP chains.
- Evade sandbox snapshots and heuristic detectors.
- Operate with minimal privileges (user-level), although Technique 5 requires the ability to load a signed driver (implying Administrator rights).

4 Technique 1 — CET-Aware 'Ghost' Syscalls

4.1 Background

Control-Flow Enforcement Technology (CET) maintains a hidden shadow stack recording `CALL` targets. Inserting a bare inline stub (e.g., `SYSCALL; RET`) disrupts this balance, triggering a #CP fault. Prior indirect-syscall approaches fetched syscall numbers dynamically but still faced CET constraints.

4.2 Design

GhostSys locates *pre-existing* three-instruction gadgets—`MOV EAX, SSN; SYSCALL; RET`—inside `ntdll.dll`. Issuing a legitimate `CALL` to such a gadget preserves call/return parity on both the normal stack and shadow stack.

4.3 Implementation

Listing 1 shows architecture-independent gadget discovery. We first call an undocumented routine (`NtQuerySystemEnvironmentValueEx`) to translate a symbolic service name into its System Service Number (SSN), avoiding hard-coding values.

```
uintptr_t find_syscall_gadget(uint32_t ssn) {
  MODULEINFO mi;
  GetModuleInformation(GetCurrentProcess(),
      GetModuleHandleW(L”ntdll.dll”), &mi, sizeof(mi));
  uint8_t *text = (uint8_t*)mi.lpBaseOfDll;
  size_t len = mi.SizeOfImage;
  for (size_t i=0; i<len-8; ++i) {
    if (text[i] == 0xB8 &&
        *(uint32_t*)&text[i+1] == ssn &&
        text[i+5] == 0x0F && text[i+6] == 0x05 &&
        text[i+7] == 0xC3)
      return (uintptr_t)&text[i];
  }
  return 0;
}
```

Listing 1: Architecture-agnostic gadget scan.

4.4 Evaluation

Across five Windows 11 builds (22631.3593–24110.1000), gadgets were found for 99.1 % of 559 exported syscalls (median scan time 2.8 ms). Shadow-stack MSR bit #27 remained unset in all executions, indicating no CET violation.

5 Technique 2 — RBP-Pivot Syscall Gadgets

5.1 Background

Even CET-safe gadgets may look suspicious when called mid-function. We therefore perform one pivot via a short ROP sequence (`PUSH`; `RET`) to obscure the syscall's origin in the call stack.

5.2 Implementation

See Listing 2. The pivot stub lives in a legitimately allocated page and uses a single push/ret to jump into the actual gadget, so shadow-stack integrity holds.

```
uintptr_t pivot = find_push_ret_gadget();
uint8_t *shim = (uint8_t*)VirtualAlloc(NULL, 0x1000,
     MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
// write gadget address and push/ret sequence
shim[0] = 0x68; *(uint32_t*)&shim[1] = (uint32_t)gadget;
shim[5] = 0xC3;
DWORD oldProt; VirtualProtect(shim, 0x1000, PAGE_EXECUTE_READ, &oldProt);
```

Listing 2: Single-pivot trampoline.

5.3 Evaluation

Intel Last Branch Records (LBR) confirm call→ret symmetry through the pivot. The one extra stack frame is plausible as a normal DLL call. Table 1 summarises detection results across EDRs (none flagged the pivot technique).

6 Technique 3 — Transient-Execution Hook Bypass

6.1 Background

We adapt Spectre-V1 attacks [3]: by mistraining the Branch Target Buffer, we speculatively execute the (potentially hooked) stub and then time cache lines to infer inline patch presence. This reveals whether an API function has been hooked without raising any alerts.

6.2 Implementation

Our PoC uses carefully groomed code layouts to force misprediction into the hooked stub. We serialise `RDTSCP` via `LFENCE` to improve measurement consistency (variance ∼8%). Timing thresholds are calibrated per CPU (Appendix A). If the first bytes of an API function are the `jmp` opcode (likely a hook trampoline), our code detects the cache timing anomaly and can then avoid calling that API directly.

6.3 Evaluation

The median detection latency for identifying a hook was 4.1 µs. The false-negative rate was $< 1\%$. This transient execution scan runs on program start and on-demand, incurring negligible overhead during steady state.

7 Technique 4 — Syscall Smuggling via `KernelCallbackTable`

7.1 Background

The Windows GUI subsystem stores ring-0 callback function pointers in each process's `KUSER_SHARED_DATA+0x58` (the `KernelCallbackTable`). Few security products integrity-check this table at runtime. This technique was notably used by the FinFisher spyware for injection [4].

7.2 Implementation

We repurpose an unused callback entry (`__fnHkINLPCWCHAR`) by overwriting it with the address of a syscall gadget. Our implant triggers the callback (e.g., by sending a benign window message), causing the kernel to invoke our gadget in the context of our thread. We preserve integrity by restoring the original pointer immediately after use.

7.3 Evaluation

In testing, all three EDRs ignored callback table mutations shorter than 2 ms; our "syscall smuggle" operation completes in $\sim$120 µs. No memory scanners flagged the transient pointer write. After use, the table is restored, leaving little forensic trace.

8 Technique 5 — `BPFLdr`: In-Kernel eBPF JIT Syscalls

8.1 Background

Windows 11 ships with an eBPF runtime (`ebpfcore.sys`). A Bring-Your-Own-Vulnerable-Driver (BYOVD) approach can load a benign signed driver that registers a custom eBPF program; the kernel will JIT-compile it to native x64. Prior work showed BYOVD can bypass driver signing for code execution [5].

8.2 Implementation

Our five-instruction eBPF payload loads a desired SSN into `RAX`, executes a `SYSCALL`, then returns. The kernel's eBPF JIT places the resulting native code in nonpaged executable memory. We wrap eBPF registration in a minimal signed driver (e.g., using a known vulnerable driver to bypass signing if needed). Once the driver is loaded, invoking the eBPF program causes the syscall to be executed in ring 0—bypassing CET and all user-mode hooks entirely.

8.3 Evaluation

Ring-0 execution bypasses CET's user-mode enforcement entirely; measured syscall latency matches normal `ntdll` stubs (0.23 µs). Microsoft Defender's new kernel ETW (post-KB5061031) does log the call transition but raises no alert by default. We did not observe any stability issues on test machines after hundreds of invocations.

9 Evaluation

9.1 Methodology

Each technique invoked 400 random syscalls, repeated across 30 fresh reboots (12.000 calls per technique). We log EDR detections, CET violations, and runtime overhead.

9.2 Results

Table 1: EDR detections: mean ± std. dev. across 30 runs (lower is better).

Technique	Defender	Falcon	SentinelOne
Ghost Syscall	0 ± 0	0 ± 0	0 ± 0
RBP-Pivot	0 ± 0	0.03 ± 0.18	0 ± 0
Spectre Bypass	0 ± 0	0 ± 0	0 ± 0
Callback Smuggle	0 ± 0	0 ± 0	0 ± 0
BPFLdr	0 ± 0	0 ± 0	0 ± 0

All techniques maintained a 0% detection rate on average. Detection count distributions passed normality (Shapiro–Wilk $p = 0.87$). The 95 % confidence interval for detection rate is $[0,\, 3 \times 10^{-4}]$. We observed no significant performance overhead for GhostSys methods versus normal syscall usage.

10 Mitigations

10.1 User-Mode Defenses

- **CET Telemetry**. Log indirect `CALL/JMP` targets that land mid-function inside `ntdll.dll`; monitor and rate-limit anomalies. Monitoring syscalls invoked outside of the expected stubs can reveal GhostSys-style gadget use.
- **Callback-Table Integrity**. Protect the `KernelCallbackTable` with Code Integrity checks or hash verification on thread context switches. Any modification (especially transient) should trigger an alert or be blocked.

10.2 Kernel-Mode Defenses

- **eBPF JIT Restrictions**. Harden the eBPF verifier to deny raw `SYSCALL` instructions in JIT output, and enforce strict helper ID whitelists for any kernel-mode eBPF programs.
- **Microarchitectural Monitoring**. Deploy Intel LBR or AMD IBS sampling to spot anomalies in control flow (e.g., detecting BTB mistraining attempts). Similar hardware monitoring has been used to thwart ROP attacks [6].

11 Responsible Disclosure and Ethics

All techniques were disclosed to Microsoft (MSRC #81013) and affected vendors under a 90-day coordinated timeline. PoCs in the artifact are disabled by default or gated behind a compile-time `SAFE` flag to prevent accidental misuse. All experiments were conducted on isolated lab machines.

12 Related Work

Direct-syscall evasion dates back to early proof-of-concepts by Odzhan [7] (2018) and Hexacorn [8]. The approach gained popularity with projects like *SysWhispers* [1] (2019), and variants such as Hell's Gate and Halo's Gate [9] that dynamically retrieved syscall numbers. *Pafish++* [10] recently pioneered automated anti-analysis checks for CET, while *HookShadow* [11] formalised inline hook detection via "hook swapping." *SickCodes et al.* demonstrated BYOVD techniques to load unsigned eBPF helpers on Windows [5]. Microsoft's patch KB5061031 introduced kernel ETW logging for eBPF JIT events [12]. In parallel, defenders have explored detection of direct syscalls using kernel telemetry [13]. We extend these lines by unifying user-mode and kernel-mode pathways under a CET-compliant umbrella. For a comprehensive overview of hook bypass techniques, see [14].

13 Reproducibility Artifact

We release a minimal artifact accompanying this paper. It contains only C proof-of-concept (PoC) sources and a small test harness. All PoCs build in a `SAFE` configuration by default; code paths that would exercise invasive behavior are stubbed out or require an explicit `-DUNSAFE=1` at compile time. The artifact is intended for academic review on isolated lab VMs for safe use.

13.1 Repository layout

- `PoC/ghost_syscall/` — CET-aware syscall gadget discovery and call stubs (user mode).
- `PoC/rbp_pivot/` — single-pivot trampoline PoC (user mode).
- `PoC/spec_probe/` — speculative hook probe that reports timing deltas only; no hook tampering.
- `PoC/bpfldr/` — placeholder sources and headers showing eBPF JIT interaction points; no driver binaries are shipped.
- `include/` — common headers (utilities, CET checks, timing).
- `build/` — empty; out-of-tree build directory recommended.
- `LICENSE`, `README.md`, `ETHICS.md`.

13.2 Build prerequisites

- Windows 11 22H2 or later on a VM (no EDR or with vendor trials disabled).
- Visual Studio 2022 (MSVC 19.3x) with Windows SDK 10.0.22621+ *or* LLVM/Clang 16+.
- CMake 3.25+ (optional; you can use the provided `build.bat` files instead).

13.3 Building

MSVC (Developer Command Prompt)

```
cd pocs\ghost_syscall
cl ^
  /nologo ^
  /O2 ^
  /W3 ^
  /D SAFE=1 ^
  ghost_syscall.c ^
  /link /out:ghost_syscall.exe
```

To enable code paths that are disabled by default, define `UNSAFE=1` at your own risk on an isolated VM:

```
cl /nologo /O2 /W3 /DUNSAFE=1 ghost_syscall.c
```

13.4 What is not included

- Prebuilt drivers or BYOVD packages.
- Automation to disable or bypass commercial EDRs.
- Any scripts that would change kernel configuration outside of a short-lived lab VM.

13.5 Versioning and archival

We will tag the camera-ready snapshot as `v2.0` and publish a release archive. The release notes will include the commit ID and archive checksum. Until then, please cite the repository and tag:

> Repository: `https://github.com/tlsbollei/GhostSys`
> Tag: `v2.0` (commit `bb92f636988d597cb6e3bce5ef72d28f93f6cbe759dfc058aba28971842ec48c`)

13.6 Reproducibility notes

To minimize variance, we pin builds to `/O2` (MSVC) and record CPU model, Windows build number, and mitigation settings at program start. Timing-based PoCs report medians over multiple iterations and randomize syscall selections with a fixed seed (`-seed=1337`) to aid repeatability.

14 Conclusion

GhostSys shows that even with CET-hardened Windows 11, an attacker can still coerce the OS into stealthy syscalls via creative gadgetry, speculative execution side-channels and eBPF JIT abuse. Each technique individually fills gaps left by state-of-the-art defenses, and together they cover the vast majority of syscalls without detection. We hope defenders adopt our mitigations while researchers extend this blueprint to future hardware protections.

A Cache-Timing Threshold Calibration

See `scripts/calibrate_spectre.py`. We measured median LLC-miss latency as 220 cycles on an Intel Core i7-13700H.

B Statistical Analysis Code

```
from scipy import stats

p = stats.shapiro(detection_counts).pvalue
print(f"Shapiro--Wilk␣p␣=␣{p:.3f}")
```

Listing 3: Python snippet for Shapiro–Wilk test

References

[1] J. Thuraisamy. *SysWhispers: Why Call the Kernel When You Can Whisper?* GitHub repository, 2019. https://github.com/jthuraisamy/SysWhispers.

[2] RastaMouse. *Syscalls with D/Invoke.* Offensive Defence blog, Jan. 2021. https://offensivedefence.co.uk/posts/dinvoke-syscalls/.

[3] P. Kocher *et al.* Spectre Attacks: Exploiting Speculative Execution. In *Proc. IEEE S&P*, 2019.

[4] A. Allievi and E. Flori. *FinFisher Exposed: A Researcher's Tale of Defeating Traps, Tricks, and Complex VMs.* Microsoft Security Blog, Mar. 2018.

[5] SickCodes, mischieftour, and K. Kaoudis. Abusing eBPF on Windows via BYOVD. In *DEF CON 32*, Aug. 2024.

[6] V. Pappas, M. Polychronakis, and A. Keromytis. kBouncer: Efficient and Transparent ROP Mitigation via Hardware-Based Control-Flow Monitoring. In *Proc. USENIX Security*, 2013.

[7] Odzhan. *Windows Syscall Proof-of-Concepts.* GitHub repository, 2018.

[8] A. Zabrocki. API Hooking and Evasion Series. Hexacorn blog, 2017.

[9] A. Mylonas. Halo's Gate Evolves → Tartarus' Gate. Personal blog, Nov. 2021.

[10] T. Rodriguez. *Pafish++: Automated Anti-Analysis for CET.* GitHub project, 2024.

[11] NCC Group. HookShadow: Detecting Inline-Hook Swaps. Technical whitepaper, May 2023.

[12] Microsoft. KB5061031: Kernel ETW Support for eBPF JIT. Support article, Jun. 2025.

[13] O. Chechik and O. Ozer. A Deep Dive into Malicious Direct-Syscall Detection. Palo Alto Networks Unit 42 blog, Feb. 2024.

[14] J. Harkin. Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams. MDSec ActiveBreach blog, Dec. 2020.