# RegPhantom Backdoor Threat Analysis

By Pierre-Henri Pezier

Published: 2026-03-30 · Archived: 2026-04-05 22:19:53 UTC

## Executive Summary

This report analyzes RegPhantom, a stealthy Windows kernel rootkit designed to give attackers code execution in kernel mode while leaving very little visible evidence behind. The malware abuses the Windows registry as a covert trigger mechanism: a usermode process can send an encrypted command through a registry write, which the driver intercepts and turns into arbitrary kernel-mode code execution.

What makes this threat notable is the combination of stealth, privilege, and trust abuse. The driver runs as a signed kernel component, allowing it to operate at the highest privilege level on Windows systems. It does not rely on normal driver loading behavior for its payloads and instead reflectively maps code into kernel memory, making the loaded module invisible to standard tools that enumerate drivers. It also blocks the triggering registry write, wipes executed payload memory, and stores hook pointers in encoded form, which significantly reduces forensic visibility.

Our analysis further shows that this is not an isolated sample. Multiple related binaries were identified across several months, including versions signed with valid certificates issued to Chinese companies. Combined with sample timeline patterns, submission history, and shared development traits, this supports an assessment of active maintenance by a China-nexus threat actor with moderate confidence.

For defenders, RegPhantom is dangerous because it enables kernel-level execution from an unprivileged usermode context and is built specifically to avoid common detection and triage methods. Traditional artifact-based investigation is unlikely to be sufficient. Detection should focus primarily on the driver binary itself, related signed samples, and the underlying code patterns shared across this malware family.

## Technical Overview

This report presents the analysis of a Windows kernel driver ( `.sys` ) signed with a Microsoft-trusted code-signing certificate, operating as a rootkit, which we track as **RegPhantom**. The driver establishes a covert usermode-to-kernel code execution pipeline by leveraging the Windows registry as a communication channel: an unprivileged usermode process writes an XOR-encrypted command to any registry key, the driver intercepts it via `CmRegisterCallback` , decrypts the

payload, and reflectively loads an arbitrary PE into kernel memory. This allows an attacker to execute code at the highest privilege level while blending into legitimate system activity.

The loaded payload can further hijack the driver's kernel callbacks, enabling persistent kernel-mode hooks without additional driver loads. Because the PE is reflectively loaded into a raw pool allocation rather than through the standard module loader, it does not appear in `PsLoadedModuleList` and is invisible to tools that enumerate loaded drivers.

The driver binary was protected with **Control Flow Guard (CFG) obfuscation**, incorporating opaque predicates and duplicated basic blocks designed to impede static analysis. All findings in this report are based on the deobfuscated binary.

## Threat Landscape

The currently observed RegPhantom sample set spans at least 18 June 2025 through 6 August 2025, based on compilation timestamps and VirusTotal first-submission dates. This window should be treated as the visible activity period rather than the full lifetime of the toolchain, as earlier samples may not have been submitted publicly and later activity may not yet have been identified. Within that timeframe, the presence of multiple samples compiled across different dates, filenames, and environments indicates active development and ongoing adaptation by the threat actors.

Two samples bear valid code-signing certificates issued to Chinese companies: **Guangzhou Xuanfeng Technology Co., Ltd.** and **Autel Intelligent Technology Corp., Ltd.** The majority of first submissions originate from China, and compilation timestamps are broadly consistent with a UTC+8 working schedule. While submission metadata alone is unreliable for attribution — as it may reflect third-party researchers or proxy infrastructure — the use of two distinct Chinese code-signing certificates represents a stronger indicator, as obtaining such certificates requires direct access to the issuing entities or their infrastructure. Taken together, these indicators support a **China-nexus** assessment at **moderate confidence**.

| SHA-256 | Size | Filename | Creation time | Submit from |
|---|---|---|---|---|
| `703dfb12edc6da592e3dfb951ca2d84bf349e6a16ad3a2ab32b275349956e7c4` | 41.00 KB | `MapDriver.sys` | 2025-06-18 12:43:05 UTC | 🇨🇳 CHINA |
| `006e08f1b8cad821f7849c282dc11d317e76ce66a5bcd84053dd5e7752e0606f` | 52.21 KB | `MyDriver-signed-20250619.sys` | 2025-06-18 13:02:00 UTC | 🇨🇳 CHINA |
| `5599ec1f3e1eb52a7e0f3b9dbd0c9849cf494c32ed1e50e76c43d2200daa283a` | 58.00 KB | `TestDriver.sys` | 2025-06-28 17:43:27 UTC | 🇨🇳 CHINA |
| `5650f8e0904433247a0cdc68c7b73c68291b52523dad1edb93a9bd7439273698` | 58.50 KB | `TestDriver.sys` | 2025-06-28 17:47:33 UTC | 🇨🇳 CHINA |
| `6606a963beb709da2d87d685d998e126f2a52efaad64eab8bbb5ba70c7ca5194` | 58.50 KB | `0629.sys` | 2025-06-28 17:54:04 UTC | 🇨🇳 CHINA |
| `97876c085318d8606e8478976d98dab77a7e905a87a4b0a27e20d794af25cd4c` | 58.00 KB | `MyDriver.sys` | 2025-06-28 17:54:26 UTC | 🇨🇳 CHINA |
| `cb2ed2ece12a675e19f2b537840a2b5d8bcdd1d508ec5c386178e60161d2cfe8` | 58.50 KB | `TestDriver.sys` | 2025-06-29 | 🇨🇳 CHINA |

| SHA-256 | Size | Filename | Creation time | Submit from |
| --- | --- | --- | --- | --- |
| | | | 03:01:11 UTC | |
| `b2bbdeb48f60e591d78ddc98fffc9504128e9b948fd58a54c2cfa927ff9db105` | 58.50 KB | `MyDriver_0629.sys` | 2025-06-29 03:01:32 UTC | 🇨🇳 CHINA |
| `218ab4cb7bf3622b4b8d5fa9196d817b91046e1eca84c26091f3f703ab214707` | 58.00 KB | `TestDriver.sys` | 2025-07-15 13:24:20 UTC | 🇨🇳 CHINA |
| `91860b4d03b32a4ca6e8e92856272d953999934e6316f65677a615cbfb8d31d0` | 80.14 KB | – | 2025-07-17 03:40:47 UTC | 🇨🇳 CHINA |
| `c55f5339abaf48b9392df67d5b6f6e011d878d7ee848724ad5dbe8c4d898ef23` | 58.50 KB | `TestDriver.sys` | 2025-07-30 06:25:04 UTC | 🇸🇬 SINGAPORE, 🇺🇸 USA |
| `7c9312ebe2afc299a0835a32700cdd2c5099c228799414c48058c0fb6095df9b` | 58.50 KB | `TestDriver.sys` | 2025-07-30 09:48:56 UTC | 🇸🇬 SINGAPORE |
| `01c3d2a947c56e16718f1f54c0820996dce1d44da25d38b2a9992eb16e6b11e6` | 58.50 KB | `TestDriver.sys` | 2025-07-30 09:49:32 UTC | 🇨🇳 CHINA, 🇸🇬 SINGAPORE, 🇺🇸 USA |
| `f6683adcb8a152d31ef1132ee3f4cb818dcf0b5e361f991286f9fb5d2d747afd` | 63.00 KB | `TestDriver.sys` | 2025-08-05 19:26:27 UTC | 🇯🇵 JAPAN |
| `7606a3b69488795fe2d71558caab7877ea313425e55a63aebb932d0d92b38aee` | 63.00 KB | `TestDriver.sys` | 2025-08-05 19:28:27 UTC | 🇯🇵 JAPAN |
| `32addf18477324f478bf93ac22be65550bc71450c9bc4fe49aa3be22219aae65` | 63.00 KB | `FsFilter.sys` | 2025-08-06 08:26:23 UTC | 🇨🇳 CHINA, 🇯🇵 JAPAN, 🇷🇺 RUSSIA |
| `0956ec57c3ddcd24c4d61bd6a4dd16b5f1468f701a286e46b761f5be4fc478ac` | 63.00 KB | `DevDriver.sys` | 2025-08-06 05:16:15 UTC | 🇨🇳 CHINA, 🇷🇺 RUSSIA, 🇺🇸 USA |
| `b10d8bb537ab05e51f08d0b942ee9f92f3226d118fcac794d1a7396bbc0b531f` | 58.50 KB | `FsFilter.sys` | 2025-08-01 05:39:36 UTC | 🇨🇳 CHINA, 🇺🇸 USA |

| SHA-256 | Size | Filename | Creation time | Submit from |
|---|---|---|---|---|
| `77646afc50ac65756999441ff5879049c51309745fc9eb86d343174ad5601f2c` | 58.50 KB | `FsFilter.sys` | 2025-07-17 03:40:47 UTC | 🇨🇳 CHINA, 🇺🇸 USA |
| `a0c291e8942c8c7feccff3fbdb65f65c76312d384a73d3748042a319209c91c` | 58.50 KB | `TestDriver.sys` | 2025-07-17 03:47:20 UTC | 🇨🇳 CHINA |
| `6e1254e478d5b7e60a7a6c6c23943884eca59b214d5a8ecdbdea1a0bbd08df58` | 58.50 KB | `TestDriver.sys` | 2025-07-15 13:30:35 UTC | 🇨🇳 CHINA |
| `aaed39996db0c5f9b7ebbda773e67aced72100af701bf2cd933c3aae6b31f9ce` | 58.50 KB | `TestDriver.sys` | 2025-07-15 13:29:31 UTC | 🇨🇳 CHINA |
| `2ece92c1b221338b0f37cc033b2a160bb03cd4d3c228f0924fcb7be6c9bbea10` | 58.50 KB | `TestDriver.sys` | 2025-07-15 13:27:17 UTC | 🇨🇳 CHINA |
| `f25784a7577f2e4fa254e93458f6c92de66c623a3029c284a39f4076bb8d7046` | 58.50 KB | `TestDriver.sys` | 2025-07-14 06:44:42 UTC | 🇨🇳 CHINA |
| `39eabd51174ae57bcaa05fc50ff7bb704464b97e315f6e03a6a447000463b261` | 58.00 KB | `TestDriver.sys` | 2025-06-28 17:43:59 UTC | 🇨🇳 CHINA |
| `836259c4475e372277b5115f8f4542c4210fd2817aaacd00f0a350b067fde165` | 43.00 KB | `MapDriver.sys` | 2025-06-26 20:00:44 UTC | 🇨🇳 CHINA |
| `8f24be8d38df0d2cec0abf78873b83d2a633b650324e99505993604909a13805` | 43.00 KB | `MapDriver.sys` | 2025-06-26 19:09:54 UTC | 🇨🇳 CHINA |
| `f06dacf7f7152c632ed435ab60bb1a8e9e9a7eb5d416eb6419eb4446f7fa821f` | 42.00 KB | `MyDriver.sys` | 2025-06-18 13:02:00 UTC | 🇨🇳 CHINA, 🇷🇺 RUSSIA |
| `1f3d90ed62bf1b4fd501cbd435d2519486b60ad91704b6e38b93da00960cd22d` | 42.00 KB | `MapDriver.sys` | 2025-06-18 12:43:58 UTC | 🇨🇳 CHINA |

Submission origin heatmap

## Related Samples

The following samples are signed by **Guangzhou Xuanfeng Technology Co., Ltd.** and exhibit the same CFG obfuscation with opaque predicates and duplicated basic blocks observed in RegPhantom. These samples have not been fully reverse-engineered at the time of writing.

| SHA-256 | Filename |
| --- | --- |
| `cc123e35363aeace09900bf3de76080eb46f7e04edede742dbdf2d80be129cc0` | `MapDriver.sys` |
| `a0eee7cd05ca3dbddb57414df99768c05ade18f9c13fb31e686558e636badf26` | `0627.sys` |
| `9721430672e361eff1f92dd4cc81686635730bc9656f1542411ed2df93dea831` | `MapDriver.sys` |

The shared obfuscation technique and signing certificate reinforce the connection to the same development pipeline.

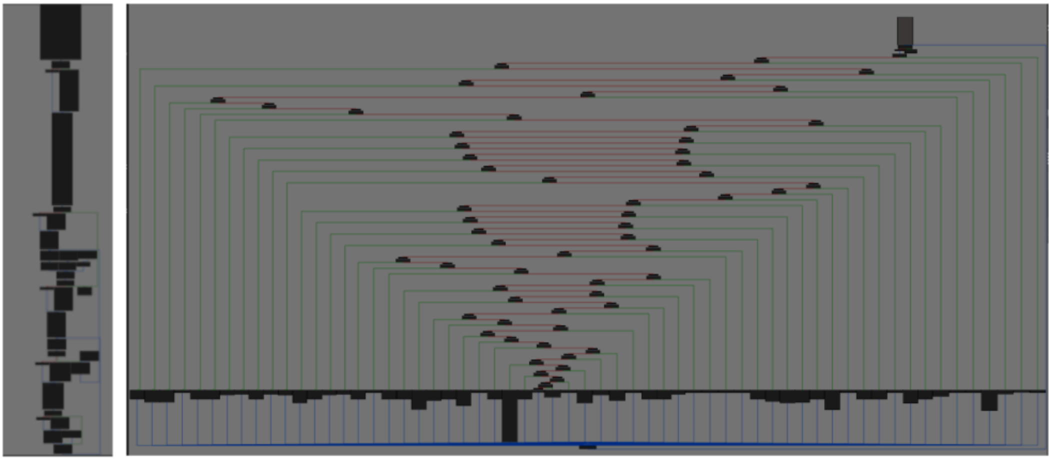
## Technical details

The technical analysis below is based on sample `006e08f1b8cad821f7849c282dc11d317e76ce66a5bcd84053dd5e7752e0606f`.

### Obfuscation

RegPhantom employs multiple layers of obfuscation to resist both static and dynamic analysis. The most prominent technique is **Control Flow Guard (CFG) obfuscation**: the binary's control flow graph is inflated with opaque predicates — conditional branches whose outcome is statically determined but difficult for a disassembler to resolve — and duplicated basic blocks. This effectively breaks decompiler output and forces the analyst to reason about each branch manually.

However, the variety of opaque predicates used is low — the same patterns are reused across the binary — making the obfuscation vulnerable to pattern-based CFG simplification. The following graph shows the same function before and after deobfuscation. The left side displays the original obfuscated CFG, inflated with opaque predicates and duplicated blocks; the right side shows the cleaned version produced by our deobfuscation script ( `CFG_deobf.py` ).



CFG obfuscation — obfuscated vs. deobfuscated

Beyond control flow, the driver obfuscates all **function calls** — both to imported APIs and to its own internal functions. The APIs are present in the import table, but the driver never calls them directly. Instead, each call site computes the target address by adding a global variable to a hardcoded offset, producing a pointer to the target function. The same technique is applied to internal function calls within the binary. This indirection prevents disassemblers and decompilers from resolving any call targets, making cross-references unreliable across the entire binary. The screenshot below shows an obfuscated call site; the original API target can be recovered using our deobfuscation script ( `deobfuscate_calls.py` ).

```
sub     rsp, rax
mov     rax, rsp
mov     [rbp+var_58], rax
mov     [r8], r9
mov     [rcx], rdx
mov     dword ptr [rax], 0
mov     rax, 8239B5897FFF77FBh
add     rax, cs:off_14000C090
lea     rcx, sub_140007D10
xor     edx, edx
lea     r8, qword_14000C100
sub     rsp, 20h
call    rax
add     rsp, 20h
mov     ecx, eax
mov     rax, [rbp+var_58]
mov     [rax], ecx
mov     [rbp+var_28], 25CAEA15h
jmp     loc_14000A485
```

API call obfuscation

**Command payloads** written to the registry are XOR-encrypted, preventing signature-based detection of the data in transit. After a command is read and processed, the driver **re-encrypts** the buffer, ensuring no plaintext is ever left in memory. Similarly, **hook function pointers** stored in the driver's data structures are XOR-encoded at rest and only decoded immediately before use, complicating memory forensics.

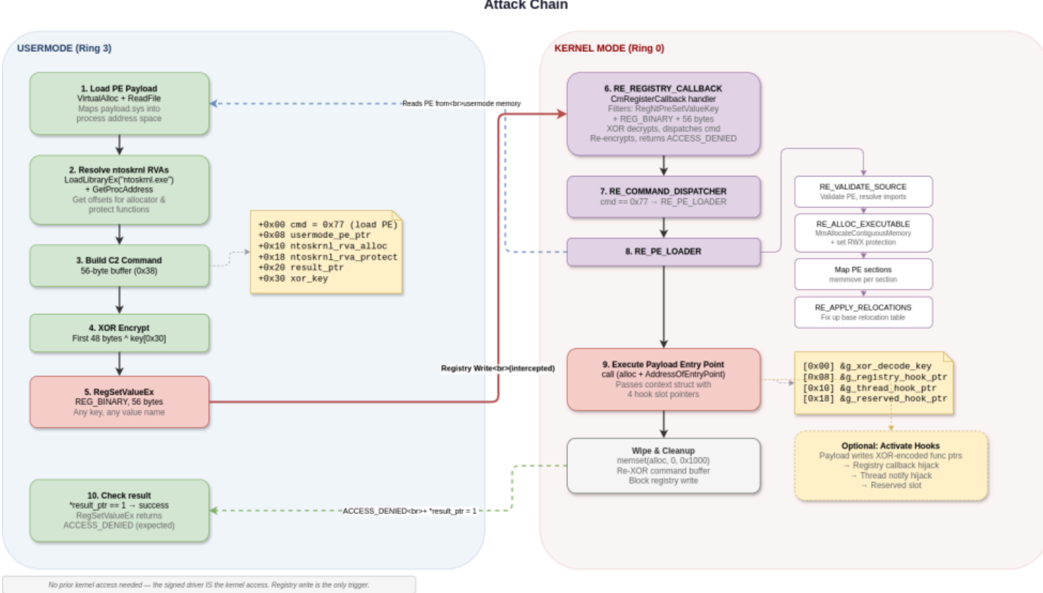
The driver also takes active steps to **deny registry writes** to the keys it monitors, preventing other processes from overwriting or tampering with its communication channel while also eliminating forensic artifacts. After executing a loaded PE payload, the driver **wipes the memory region** where the payload resided, leaving no trace of the executed code.

Finally, the use of a **valid code-signing certificate** allows the driver to bypass Windows Driver Signature Enforcement (DSE), enabling it to load on systems with default security policies.

| Technique | Purpose |
| --- | --- |
| CFG obfuscation with opaque predicates | Hinders static analysis and decompilation |
| API call obfuscation | Hides calls to sensitive kernel APIs from static analysis |
| XOR-encrypted command payloads | Prevents signature-based detection of commands |
| Re-encryption after processing | No plaintext commands left in memory |
| Registry write denial | No artifacts in the registry |
| XOR-encoded hook pointers | Obfuscates function pointers in memory dumps |
| Post-execution memory wipe | Loaded PE is erased after running |
| Signed driver | Bypasses Driver Signature Enforcement |

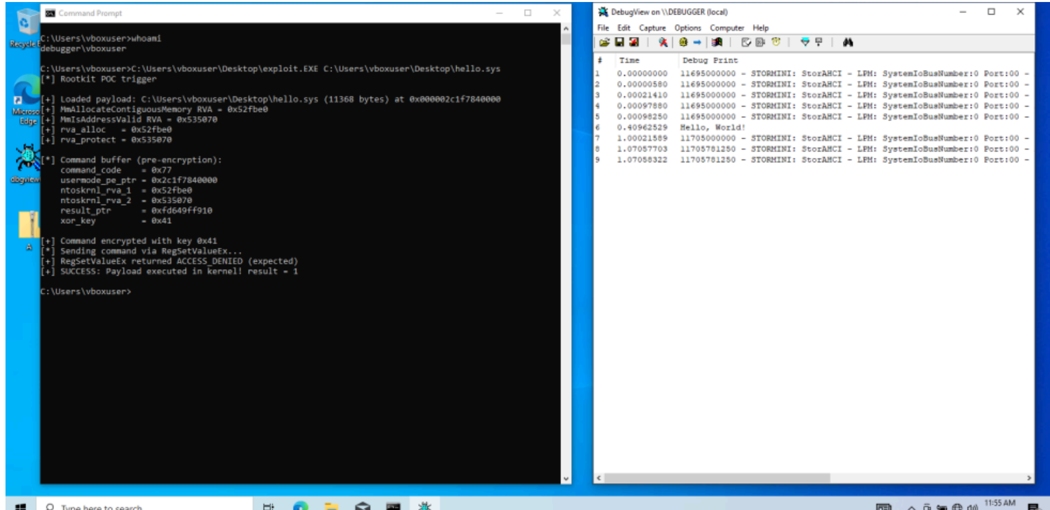
### Execution Flow

The following diagram illustrates the end-to-end attack chain:



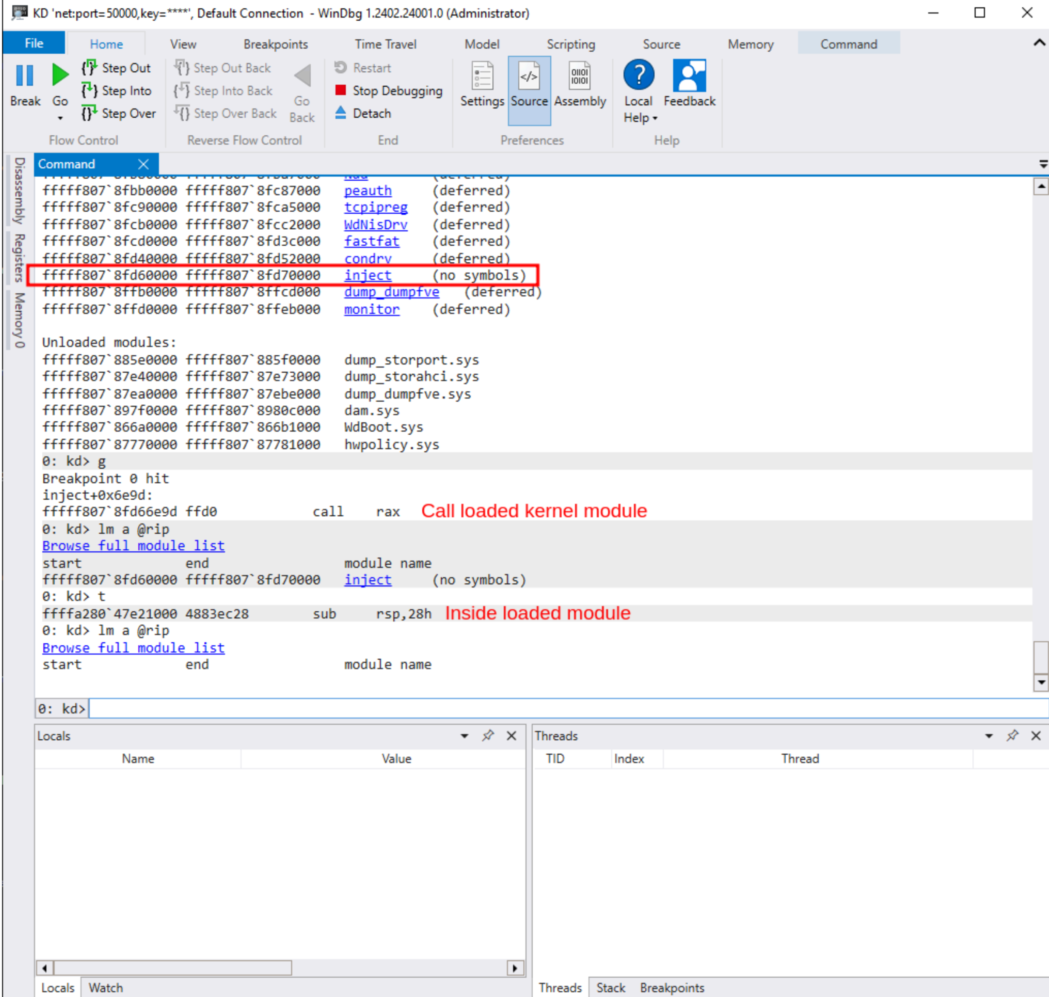

Attack chain

Only the kernel driver was recovered during analysis — neither the accompanying userland executable nor the stage 2 kernel module loaded by the driver were found. To validate the attack chain, we developed our own proof-of-concept: a userland trigger ( `poc_trigger.c` ) named “exploit.EXE” that crafts the 56-byte XOR-encrypted command buffer and issues the registry write, paired with a minimal “hello world” kernel module named “hello.sys” as the payload. The screenshot below confirms that an unprivileged usermode process can successfully load an arbitrary unsigned module into kernel space through RegPhantom:



Exploitation POC

The WinDbg trace below confirms that the injected module executes in kernel mode. Because the PE is reflectively loaded into a raw pool allocation rather than through the standard module loader, it does not appear in the kernel module list ( `lm` / `PsLoadedModuleList` ), making it invisible to tools that enumerate loaded drivers.




Module injection in kernel land

*Note: the module name visible in the trace was chosen for demonstration purposes; in a real deployment, the payload would use a less conspicuous name.*

**1. DriverEntry**

Registers two kernel callbacks and sets the unload routine:

```
CmRegisterCallback(RE_REGISTRY_CALLBACK, NULL, &g_cm_callback_cookie)
PsSetCreateThreadNotifyRoutine(RE_THREAD_NOTIFY)
DriverObject->DriverUnload = RE_CLEANUP
```

If `PsSetCreateThreadNotifyRoutine` fails, it cleans up by calling `CmUnRegisterCallback`. Both callbacks are registered at load time to avoid suspicion — registering kernel callbacks in `DriverEntry` is normal driver behavior.

**2. Registry Communication Channel (RE_REGISTRY_CALLBACK)**

The driver intercepts **all** registry operations system-wide via `CmRegisterCallback`. It filters for a specific write pattern:

- Operation = `RegNtPreSetValueKey` (pre-write)
- Value type = `REG_BINARY`
- Data size = exactly **56 bytes** (0x38)

Any registry key works — the communication channel is path-agnostic. When a matching write is intercepted:

1. Copies the 56-byte data to a local buffer
2. XOR-decrypts the first 48 bytes using the single-byte key at offset 0x30
3. Passes the decrypted command to `RE_COMMAND_DISPATCHER`
4. Re-encrypts the buffer (no plaintext left in memory)
5. Returns `STATUS_ACCESS_DENIED` (0xC0000022) to **block the write**

The registry value is never actually written. Monitoring tools (Procmon, ETW) only see a failed `RegSetValueEx` with an access denied error — indistinguishable from a permission issue.

**3. Command Protocol (RE_COMMAND_DISPATCHER)**

The decrypted 48-byte payload is parsed as:

```
+0x00  QWORD  command_code      0x77 = load and execute PE
+0x08  QWORD  usermode_pe_ptr   Usermode VA of raw PE image
+0x10  QWORD  ntoskrnl_rva_1    ntoskrnl offset for memory allocator
+0x18  QWORD  ntoskrnl_rva_2    ntoskrnl offset for memory protect function
+0x20  QWORD* result_ptr        Usermode pointer, written to 1 on success
+0x28  QWORD  reserved
+0x30  BYTE   xor_key           Single-byte XOR key (not encrypted)
+0x31  7 bytes padding
```

The callback runs in the calling process's context, so all usermode pointers are directly accessible from kernel mode. Only command `0x77` is implemented.

**4. Reflective PE Loader (RE_PE_LOADER)**

When command `0x77` is dispatched:

1. **RE_VALIDATE_SOURCE** — Validates the PE at the usermode address, walks the import directory, resolves imports using `RtlFindExportedRoutineByName`
2. **RE_ALLOC_EXECUTABLE** — Loads `ntoskrnl.exe` base, resolves allocator/protect functions from the provided RVAs, allocates `SizeOfImage` bytes, zeroes it, sets `PAGE_EXECUTE_READWRITE` (0x40)
3. **Section mapping** — Copies each PE section to its virtual address in the allocated buffer via `memmove`
4. **RE_APPLY_RELOCATIONS** — Processes the PE relocation table to fix up addresses for the new base
5. **Entry point execution** — Calls `allocated_base + AddressOfEntryPoint` with a context struct:

```
Context struct (0x20 bytes):
  [0x00]  → &g_xor_decode_key
```

```
  [0x08]  → &g_registry_callback_hook_ptr
  [0x10]  → &g_thread_notify_hook_ptr
  [0x18]  → &g_reserved_hook_ptr
```

1. **Cleanup** — `memset(allocated_base, 0, 0x1000)` wipes the mapped image after execution

#### 5. Hook Activation

The loaded PE receives pointers to 4 driver globals. By writing XOR-encoded function pointers to these slots, the payload can:

- **Hijack the registry callback** (`g_registry_callback_hook_ptr`) — all future registry operations are forwarded to the payload's handler, enabling a more sophisticated C2 protocol
- **Hijack the thread notify callback** (`g_thread_notify_hook_ptr`) — the payload receives all thread creation/destruction events without calling `PsSetCreateThreadNotifyRoutine` itself
- **Use the reserved slot** (`g_reserved_hook_ptr`) — available for future expansion

All hook pointers are XOR-encoded with `g_xor_decode_key` before being decoded at call time, adding a layer of obfuscation to memory forensics.

#### 6. Thread Notify Stub (RE_THREAD_NOTIFY)

Registered at driver load but completely dormant until a payload activates it. Once `g_thread_notify_hook_ptr` is set, it decodes the handler (`g_xor_decode_key ^ g_thread_notify_hook_ptr`) and forwards `(ProcessId, ThreadId, Create)` to the payload.

This depends entirely on `RE_REGISTRY_CALLBACK` — the only activation path is through the registry C2 → PE loader chain.

## Conclusion

RegPhantom is a purpose-built kernel rootkit that turns the Windows registry into a stealthy execution channel. By intercepting registry writes at the kernel level, it avoids leaving any persistent artifacts — no registry values are written, no files are dropped, and the injected code never appears in the kernel module list. The combination of a valid code-signing certificate, CFG obfuscation, and reflective PE loading makes it difficult to detect through conventional static or forensic analysis.

The presence of multiple samples compiled over several months, signed with two distinct Chinese certificates, and submitted from multiple countries points to an actively maintained tool by a China-nexus threat actor at moderate confidence. The related samples sharing the same obfuscation and signing infrastructure suggest a broader toolkit that extends beyond RegPhantom itself.

Detection should focus on the driver binary on disk rather than runtime artifacts, as the rootkit is specifically designed to leave none. The YARA rule provided below targets the driver's unique byte-level patterns — the XOR decryption loop and the command selector — which have remained stable across all observed samples.

## Detection

### Artifacts

RegPhantom leaves no persistent artifacts on disk or in the registry. Because the driver returns `STATUS_ACCESS_DENIED` before the registry write completes, the command payload is never committed — no registry value is created or modified. The loaded PE is wiped from kernel memory after execution. As a result, traditional forensic approaches based on registry or filesystem analysis will not surface evidence of RegPhantom activity.

### yara

```
rule MAL_Kernel_RegPhantom_Mar26 {
   meta:
      description = "Detects RegPhantom, a kernel-mode rootkit that allows an attacker to inject arbitrary coc
      author = "Pezier Pierre-Henri (Nextron Systems)"
      date = "2026-03-19"
```

```
        reference = "Internal Research"
        hash = "006e08f1b8cad821f7849c282dc11d317e76ce66a5bcd84053dd5e7752e0606f"
        score = 80
    strings:
        $s1 = "CmRegisterCallback" fullword
        $s2 = "PsSetCreateThreadNotifyRoutine" fullword

        $o1 = {
            // xor decrypt
            48 8b 09     // mov     rcx, [rcx]
            0f b6 14 08  // movzx   edx, byte ptr [rax+rcx]
            4c 31 c2     // xor     rdx, r8
            88 14 08     // mov     [rax+rcx], dl
        }
        $o2 = {
            // Command selector
            c6 01 01     // mov     byte ptr [rcx], 1
            48 83 38 77  // cmp     qword ptr [rax], 77h ; 'w'; Check if command_code (offset 0x00) == 0x77 ('w'
            0f 94 c0     // setz    al
            24 01        // and     al, 1
        }
    condition:
        uint16(0) == 0x5a4d
        and all of them
}
```

### About the author:

#### Pierre-Henri Pezier

Pierre-Henri Pezier is an IT Security Engineer and Threat Researcher with over a decade of experience in offensive security, reverse engineering, malware analysis and secure software development. He began reverse-engineering software in the early 2010s, a passion that expanded into analyzing advanced threats, developing decryptors, and writing detection rules. With a background in both offensive and defensive security, Pierre-Henri has worked on malware classification engines, sandbox environments, and EDR evasion techniques.

---

Source: https://www.nextron-systems.com/2026/03/20/regphantom-backdoor-threat-analysis/