How ToddyCat tried to hide behind AV software

By Andrey Gunkin

Published: 2025-04-07 · Archived: 2026-04-05 17:22:33 UTC

To hide their activity in infected systems, APT groups resort to various techniques to bypass defenses. Most of these techniques are well known and detectable by both EPP solutions and EDR threat-monitoring and response tools. For example, to hide their activity in Windows systems, cybercriminals can use kernel-level rootkits, in particular malicious drivers. However, in the latest versions of Windows, kernel-mode drivers are loaded only if digitally signed by Microsoft. Attackers get round this protection mechanism by using legitimate drivers that have the right signature, but contain vulnerable functions that allow malicious actions in the context of the kernel. Monitoring tools track the installation of such drivers and check applications that perform it. But what if a security solution performs unsafe activity? Such software enjoys the trust of monitoring tools and doesn't raise suspicions.

And that's precisely what ToddyCat attackers exploited by running their tool in the context of a security solution.

Detection

In early 2024, while investigating ToddyCat-related incidents, we detected a suspicious file named version.dll in the temp directory on multiple devices.

This 64-bit DLL, written in C++, turned out to be a complex tool called TCESB. Previously unseen in ToddyCat attacks, it is designed to stealthily execute payloads in circumvention of protection and monitoring tools installed on the device.

Kaspersky products detect this tool as Trojan.Win64.ToddyCat.a, Trojan.Win64.ToddyCat.b.

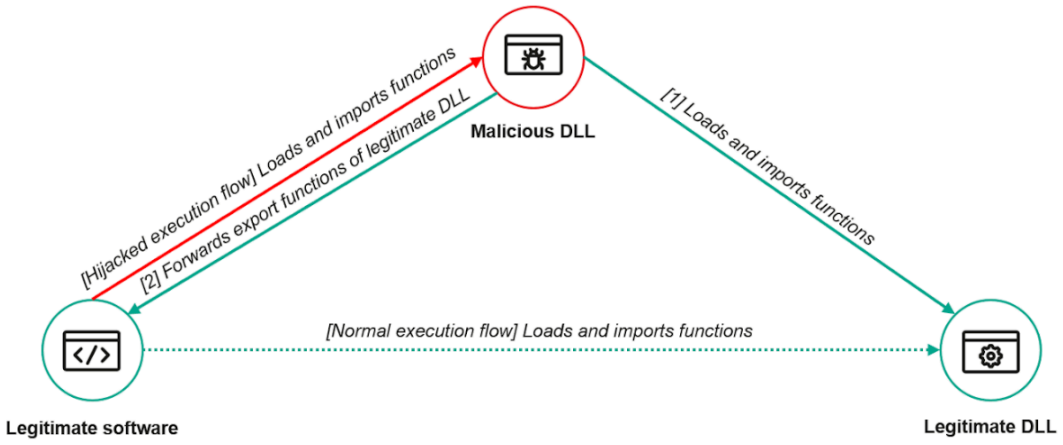
DLL proxying

Static analysis of the DLL library showed that all functions exported by it import functions with the same names from the system file version.dll (Version Checking and File Installation Libraries).

```
[redacted]\0x008F506013456EA5151DF779D3E3FF0F
00000001`8000802A: C744244C00000000          mov         d,[rsp][04C],0
00
00
00      1= 00036872 GetFileVersionInfoA = C:/Windows/System32/version.dll.GetFileVersionInfoA
00      2= 000368C1 GetFileVersionInfoByHandle = C:/Windows/System32/version.dll.GetFileVersionInfoByHandle
00      3= 00036912 GetFileVersionInfoExA = C:/Windows/System32/version.dll.GetFileVersionInfoExA
00      4= 0003695E GetFileVersionInfoExW = C:/Windows/System32/version.dll.GetFileVersionInfoExW
00      5= 000369AC GetFileVersionInfoSizeA = C:/Windows/System32/version.dll.GetFileVersionInfoSizeA
00      6= 000369FE GetFileVersionInfoSizeExA = C:/Windows/System32/version.dll.GetFileVersionInfoSizeExA
00      7= 00036A52 GetFileVersionInfoSizeExW = C:/Windows/System32/version.dll.GetFileVersionInfoSizeExW
00      8= 00036AA4 GetFileVersionInfoSizeW = C:/Windows/System32/version.dll.GetFileVersionInfoSizeW
00      9= 00036AF0 GetFileVersionInfoW = C:/Windows/System32/version.dll.GetFileVersionInfoW
00     10= 00036B31 VerFindFileA = C:/Windows/System32/version.dll.VerFindFileA
00     11= 00036B6B VerFindFileW = C:/Windows/System32/version.dll.VerFindFileW
00     12= 00036BA8 VerInstallFileA = C:/Windows/System32/version.dll.VerInstallFileA
00     13= 00036BE8 VerInstallFileW = C:/Windows/System32/version.dll.VerInstallFileW
00     14= 00036C29 VerLanguageNameA = C:/Windows/System32/version.dll.VerLanguageNameA
00     15= 00036C6B VerLanguageNameW = C:/Windows/System32/version.dll.VerLanguageNameW
00     16= 00036CAB VerQueryValueA = C:/Windows/System32/version.dll.VerQueryValueA
00     17= 00036CE9 VerQueryValueW = C:/Windows/System32/version.dll.VerQueryValueW
00
00
```

List of functions exported by TCESB

This indicates that the attackers use a DLL-proxying technique (Hijack Execution Flow, T1574) to run the malicious code. By means of this technique, a malicious DLL exports all functions of a legitimate one, but instead of implementing them, redirects calls to these functions to the original DLL. This way, an application that loads the malicious library will continue to work as normal, with the malicious code running in the context of this application in the background.



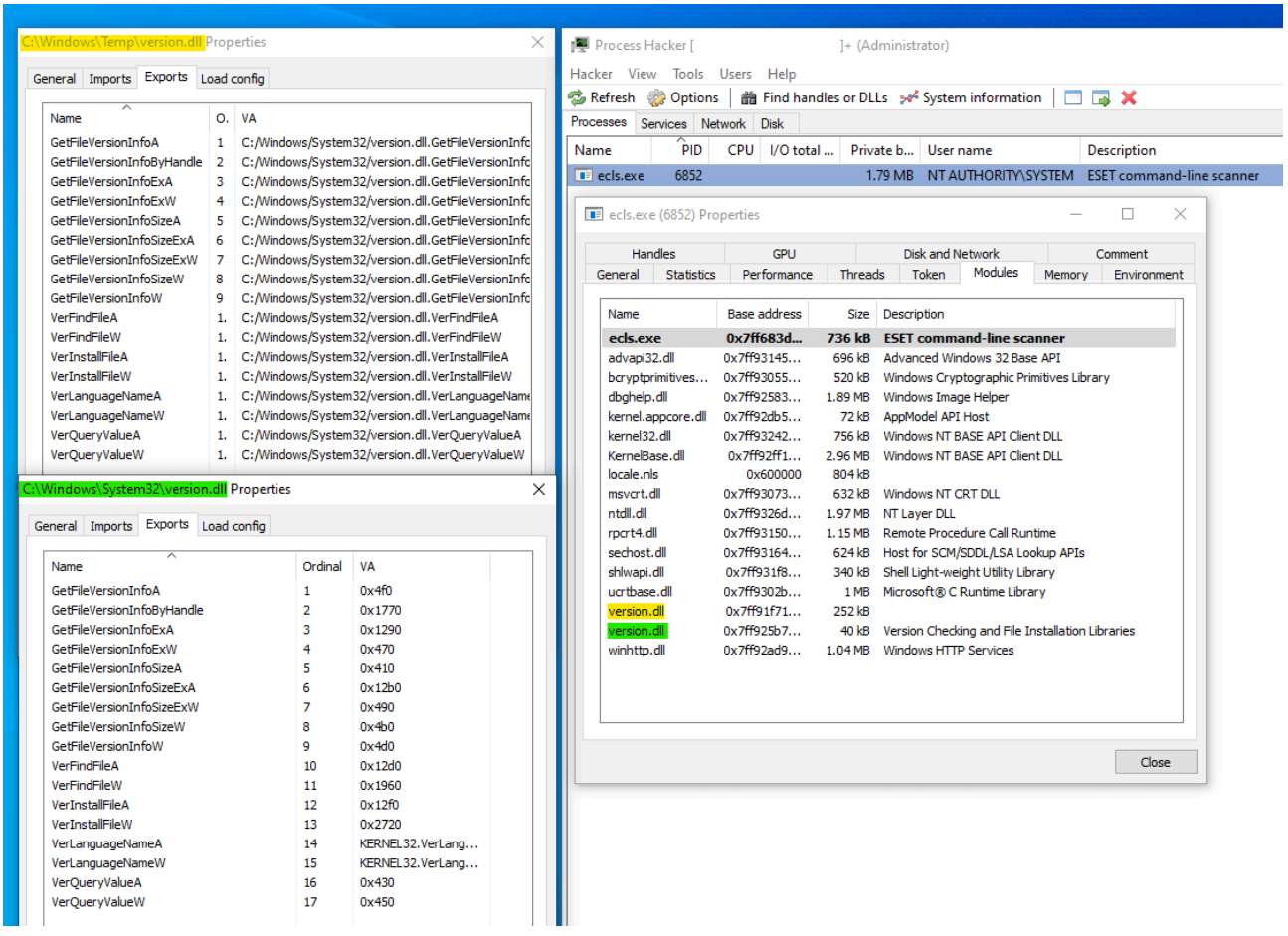
Schematic of DLL proxying

However, this is not enough to launch malware. For a malicious DLL to be able to take control, the application that loads it must contain insecure code. Such code searches for loaded dynamic library images in folders where they should not be located. If one of these folders contains a malicious library, the vulnerable application will load it instead of the legitimate one. Microsoft has an official advisory on preventing unsafe DLL loading.

CVE-2024-11859 vulnerability in ESET Command line scanner

It took us a while to find the file that loads the TCESB tool. We studied the system directories on devices where the malicious DLLs were found. On one of these, in the same folder as TCESB, there was an extensionless executable file named ecls. We believe that the operator, when transferring files to the device, made a mistake in the filename and moved two copies of it. After performing malicious activity, the file with the extension was deleted, while the other one remained in the system. This file turned out to be a component of ESET's EPP solution – a scanner launched from the command line (ESET Command line scanner). Dynamic analysis showed that the scanner insecurely loads the system library version.dll, first checking for the file in the current directory, then searching for it in the system directories. This can result in a malicious DLL library being loaded, which constitutes a vulnerability. We compiled a report with a detailed description of it, and sent it to ESET as part of the Coordinated Vulnerability Disclosure process. ESET registered the CVE-2024-11859 vulnerability, then on January 21, 2025 released an update for the ecls file patching the security issue. On April 4, information about this vulnerability appeared in an ESET security advisory.

To analyze TCESB, we ran it in a virtual environment. In the address space of the ESET Command-line scanner process, we can see two version.dll files. One is the system library, the other is the DLL of the TCESB tool.



Malicious and legitimate libraries in the memory of the ecls.exe process

Basic functionality

To determine the main functions of the malicious tool, we examined the strings located in its DLL.

```
[+] Downloading kernel offsets succeeded!
present•
not found•
[!] The kernel offsets required couldn't be retrieve using any of the methods specified
Get CallbackListHeader failed
NtOpenProcess %d error %d!
[-] Init failed: required offsets for kernel operations couldn't be loaded (error 0x%lx)!
Process %d Handle %p !
WinDefend•
[!] Couldn't allocate memory to enumerate the drivers in Kernel callbacks•
[*] Fail to GetDriverHandle..
WdNisSvc•
[+] Object callbacks have %s been found•
[+] Enabled EDR object callbacks are %s !
[*] Terminate %s  %d...
[+] Check the ETW Threat Intelligence Provider state•
[*] Check RegisterCallback ....
[*] Driver Loaded Succeed ..
taskkill /pid %d /f /t•
Delete EDR RegNotif Entry %p [%s]
[*] Clear All EDR RegisterCallback Success
[*] Check if driver available ...
```

Snippet of the list of strings that TCESB contains

The strings are not obfuscated. The search shows that most of them belong to the open-source malicious tool EDRSandBlast, designed to bypass security solutions. Kaspersky solutions detect it with the verdict HEUR:HackTool.Win64.EDRSandblast.a. ToddyCat created the TCESB DLL on its basis, modifying the original code to extend the malware's functionality. The resulting tool's capabilities include modifying operating system kernel structures to disable notification routines, for example, about a process creation event in the system or a load event.

Searching for addresses in the kernel memory

To find the structures in the kernel memory needed to disable notification routines, TCESB determines the version of the Windows kernel in the context of which it is running. To do this, it uses the GetNtoskrnlVersion() function.

```
1 LPTSTR *__fastcall GetNtoskrnlVersion()
2 {
3   LPTSTR *ntoskrnlPath; // [rsp+20h] [rbp-218h]
4   char versionBuffer[520]; // [rsp+30h] [rbp-208h] BYREF
5
6   if ( !tcslen(&g_ntoskrnlVersion) )
7   {
8     ntoskrnlPath = GetNtoskrnlPath();
9     memset(versionBuffer, 0, 0x200ui64);
10    GetFileVersion(versionBuffer, 256, ntoskrnlPath);
11    stprintf_s(&g_ntoskrnlVersion, 256, L"ntoskrnl_%s.exe", versionBuffer);
12  }
13  return &g_ntoskrnlVersion;
14 }
```

Function for getting the Windows kernel version implemented in TCESB

Next, to get information about the memory offsets of the structures corresponding to the operating system kernel version, TCESB uses one of two data sources: a CSV or PDB file.

First, the tool checks the CSV file contained in its own resources section. Stored there in table form is information about several popular kernel versions and their corresponding offsets.

TCESB searches this file line by line for a match with the previously obtained version of the current Windows kernel.

```
ntoskrnlVersion = GetNtoskrnlVersion();
hResInfo = FindResourceW(hinstDLL, lpName, lpType);
if ( hResInfo )
{
  hResData = LoadResource(hinstDLL, hResInfo);
  if ( hResData )
  {
    cbMultiByte = SizeofResource(hinstDLL, hResInfo);
    lpMultiByteStr = LockResource(hResData);
    S = sub_180015574(2i64 * (cbMultiByte + 1));
    wmemset(S, 0, 2i64 * (cbMultiByte + 1));
    MultiByteToWideChar(0, 0, lpMultiByteStr, cbMultiByte, S, cbMultiByte);
    lnBuf = fgetts(S, L"\n");
    v12 = 0i64;
    while ( lnBuf )
```

Snippet of the function for getting and reading a CSV file from TCESB resources

We studied the CSV file in the EDRSandBlast repository and its change history. The contents of the TCESB CSV fully match the CSV data in the EDRSandBlast version of August 13, 2022, while the original malware commit of October 6, 2023 adds lines that are missing in the TCESB resource. This indicates a time period during which the creators of TCESB used the EDRSandBlast code.

If the CSV file does not contain data on structures corresponding to the required kernel version, TCESB reads their addresses from the PDB file. To get it, the malware accesses the file C:\Windows\System32\ntoskrnl.exe, which contains information about the kernel file version, and inserts the data from this file into the following template, generating a URL:

```
https://msdl.microsoft.com/download/symbols/%s/%08X%04hX%04hX%016llX%X/%s
```

This is the address of Microsoft debug information server, where TCESB sends a GET request to download the PDB file. The received file is saved in the current TCESB directory, and data on the offsets of the required kernel memory structures are read from it.

Vulnerable driver

To modify the kernel structures that store callbacks used to notify applications of system events, TCESB deploys the Bring Your Own Vulnerable Driver (BYOVD) technique (Exploitation for Defense Evasion, T1211). It does

this by installing a vulnerable driver in the system through the Device Manager interface, using an INF file with installation information.

```
HANDLE installDriver()
{
  HANDLE result; // rax
  WCHAR a1[32]; // [rsp+40h] [rbp-258h] BYREF
  WCHAR Filename[268]; // [rsp+80h] [rbp-218h] BYREF

  if ( hObject )
    goto LABEL_7;
  memset(Filename, 0, 0x208ui64);
  GetModuleFileNameW(0i64, Filename, 0x104u);
  PathRemoveFileSpecW(Filename);
  strcpy(Filename, L"\\dbutildrv2.INF");
  if ( SetupDi(Filename, L"ROOT\\DBUtilDrv2", 0x26u, &DeviceInfoOut, &DeviceInfoData) )
    logInfo(L"supSetupInstallDriverFromInf Success\n");
  else
    logInfo(L"supSetupInstallDriverFromInf Failed %d\n");
```

Snippet of decompiled code for installing the TCESB driver

TCESB uses the Dell DBUtilDrv2.sys driver, which contains the CVE-2021-36276 vulnerability. This is a utility driver used to update PC drivers, BIOS and firmware.

Launching the payload

Once the vulnerable driver is installed in the system, TCESB runs a loop in which it checks every two seconds for the presence of a payload file with a specific name in the current directory – the payload may not be present at the time of launching the tool. Presumably, this is to allow the operator to verify that the tool was run without errors, so that the payload file can be moved without risk of detection. As soon as the file appears in the path being checked, it is passed to the decryption function.

```
checkDriver();
while ( !PathFileExistsW(pszPath) )
{
  logInfo(L"[+] wait...\n");
  Sleep(0x7D0u);
}
dacFileData = decryptFile(pszPath);
```

Snippet of decompiled TCESB code

The tool creates its own log file for recording all stages of execution in detail.

```
Downloading https://msdl.microsoft.com/download/symbols/ntkrnlmp.pdb/<HASH>/ntkrnlmp.pdb...
[+] Downloading offsets succeeded![+] wait...
[+] wait...
[+] wait...
[+] wait...
[+] wait...
[+] wait...
[+] wait...
```

Example of log file contents

We studied two samples of the TCESB tool. Although we were unable to obtain the payload files, our research shows that they have different names (kesp and ecore) and both are extensionless.

Our analysis of the tool code found that the data in the payload file is encrypted using AES-128.

```
if ( !CryptAcquireContextW(&phProv, 0i64, L"Microsoft Enhanced RSA and AES Cryptographic Provider", 0x18u, 0xF0000000) )
  return 0i64;
if ( !CryptCreateHash(phProv, CALG_SHA_256, 0i64, 0, &phHash) )
  return 0i64;
```

Snippet of code for determining the encryption algorithm

The decryption key is in the first 32 bytes of the payload file, followed by the encrypted data block. Below is a snippet of code for reading the key:

```
ReadFile(hFile, pbData, 32u, &NumberOfBytesRead, 0i64);
if ( !CryptHashData(phHash, pbData, 32u, 0) || !CryptDeriveKey(phProv, CALG_AES_128, phHash, 0, phKey) )
  return 0i64;
```

Snippet of code for reading the key from the payload file

The key decrypts the data block:

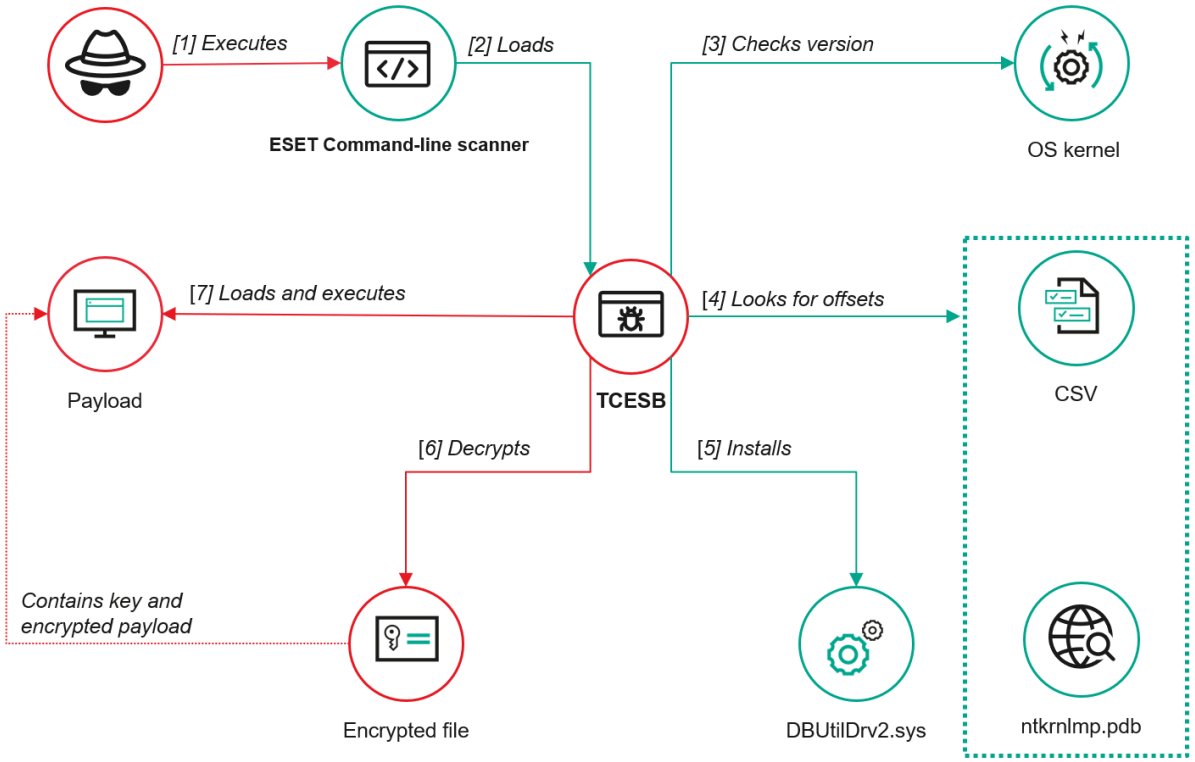
```
decryptFileSize = GetFileSize(hFile, 0i64);
if ( ReadFile(hFile, &Buffer, 0x30u, &NumberOfBytesRead, 0i64) )
{
  while ( NumberOfBytesRead )
  {
    i += NumberOfBytesRead;
    if ( i == decryptFileSize )
      Final = 1;
    if ( !CryptDecrypt(phKey[0], 0i64, Final, 0, &Buffer, &NumberOfBytesRead) )
      return 0i64;
    memcpy(&pbData[offset], &Buffer, NumberOfBytesRead);
    offset += NumberOfBytesRead;
    Buffer = 0i64;
    v10 = 0i64;
    v11 = 0i64;
    if ( !ReadFile(hFile, &Buffer, 0x30u, &NumberOfBytesRead, 0i64) )
      break;
  }
}
```

Snippet of code for reading and decrypting the payload file

The read data is placed in memory and executed.

Takeaways

We discovered a sophisticated tool that the ToddyCat APT group tried to use for stealth execution in compromised systems. This tool exploits a chain of vulnerabilities, as well as an old version of a known open-source malware that the attackers modified to extend its functionality.



Schematic of tool operation

To detect the activity of such tools, it's recommended to monitor systems for installation events involving drivers with known vulnerabilities. Lists of such drivers can be found on the loldrivers project website, for example. It's also worth monitoring events associated with loading Windows kernel debug symbols on devices where debugging of the operating system kernel is not expected. We also advise using operating system tools to check all loaded system library files for the presence of a digital signature.

Indicators of compromise

Malicious Files Hashes

D38E3830C8BA3A00794EF3077942AD96 version.dll
008F506013456EA5151DF779D3E3FF0F version.dll

Legitimate file for DLL proxying

8795271F02B30980EBD9950FCC141304	ESET Command-line scanner

Legitimate files for BYOVD

B87944DCC444E4C6CE9BB9FB8A9C0DEF	dbutildrv2.INF
DE39EE41D03C97E37849AF90E408ABBE	DBUtilDrv2.cat
DACB62578B3EA191EA37486D15F4F83C	dbutildrv2.sys
