BlackCat Ransomware Deploys New Signed Kernel Driver

trendmicro.com/en_us/research/23/e/blackcat-ransomware-deploys-new-signed-kernel-driver.html

May 22, 2023

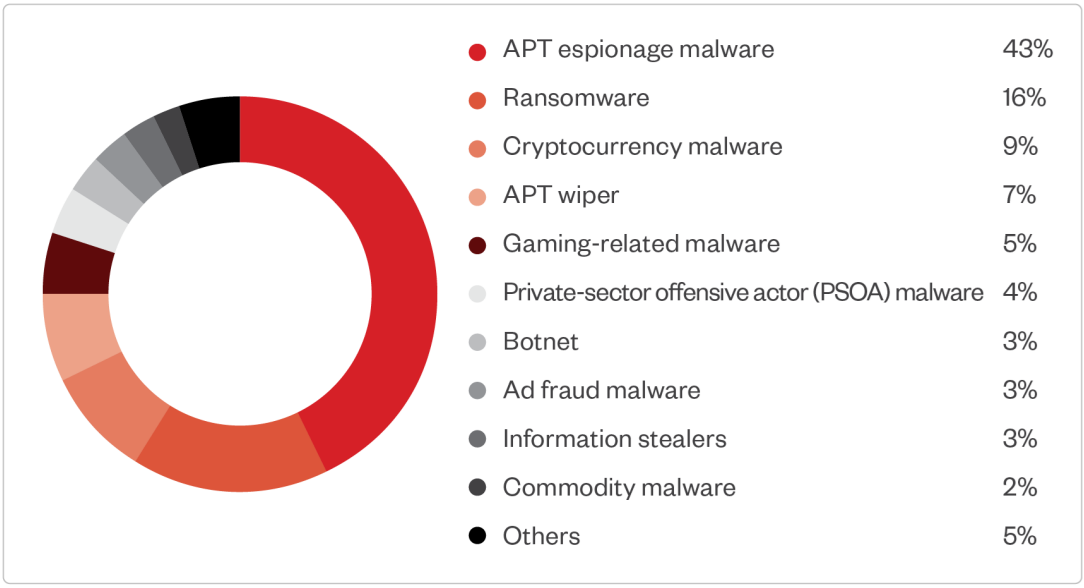

Figure 1. Distribution of kernel-level threats

Executive Summary

In late December 2022, Mandiant, Sophos and Sentinel One, via a coordinated disclosure, reported malicious kernel drivers being signed through several Microsoft hardware developer accounts (certified by Microsoft's Windows Hardware Developer Program). These profiles had been used in a number of cyberattacks that included ransomware-based incidents. Microsoft subsequently revoked several Microsoft hardware developer accounts that were abused in these attacks.

In this blog post, we will provide details on a BlackCat ransomware incident that occurred in February 2023, where we observed a new capability, mainly used for the defense evasion phase, that overlaps with the earlier malicious drivers disclosed by the three vendors. BlackCat affiliates have been known to use multiple techniques during the defense evasion phase, impairing defenses by disabling and modifying tools or using techniques as safe mode boot.

Our analysis sheds light on this new capability, which involves the use of a signed kernel driver for evasion. We believe that this new kernel driver is an updated version that inherited the main functionality from the samples disclosed in previous research. The driver was used with a separate user client executable in an attempt to control, pause, and kill various processes on the target endpoints related to the security agents deployed on the protected machines.

Malicious actors use different approaches to sign their malicious kernel drivers: Typically by abusing Microsoft signing portals, using leaked and stolen certificates, or using underground services. In our case, the attackers tried to deploy the old driver disclosed by Mandiant, which is signed through Microsoft (SHA256: b2f955b3e6107f831ebe67997f8586d4fe9f3e98). Since this driver has already been previously known and detected, the malicious actors deployed another kernel driver signed by a stolen or leaked cross-signing certificate. Trend Micro continues to monitor the abuse of any signed drivers and the related tools, tactics, and procedures (TTPs) associated with this attack surface.

The malicious signed kernel drivers

The February 2023 ransomware incident we observed proves that ransomware operators and their affiliates have a high level of interest in gaining privileged-level access for the ransomware payloads they use in their attacks. They normally use ransomware families that incorporate low-level components to avoid detection from security products once the final payloads are dropped. By mapping the kill chains of these kernel-level threats, we found that most kernel-related payloads are usually found during the defense evasion phase, as shown in Figure 1.

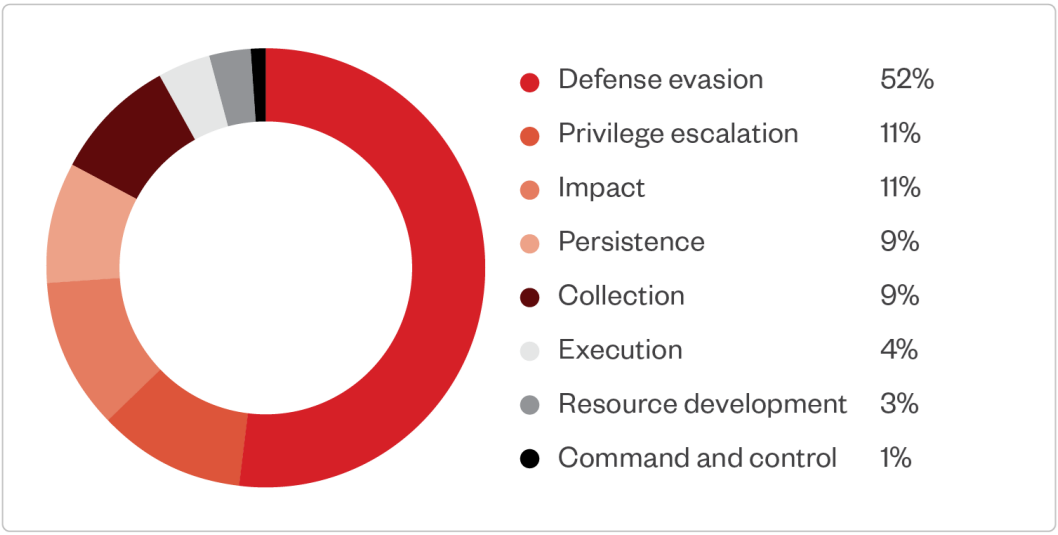

Figure 2. Most kernel-related payloads are found during the defense evasion phase
Some ransomware attacks try to comply with Microsoft code-signing requirements. This gives malicious actors the flexibility to compile kernel modules designed for very specific tasks (usually involving defense impairing and evasion) before dropping the actual payload. Ransomware operators can do one of the following approaches:

1. Use a code-signing certificate that was either leaked, stolen from a compromised environment, or purchased from an underground market.

2. Obtain a new valid code-signing certificate by impersonating a legitimate entity and following Microsoft's process for getting the cross-signing certificate (back when Microsoft still allowed cross signing for kernel-mode code), abusing Microsoft's portal for issuing signed kernel modules, and purchasing valid code-signing certificates and/or Extended Validation (EV) certificates that are tied to real identities from underground markets.

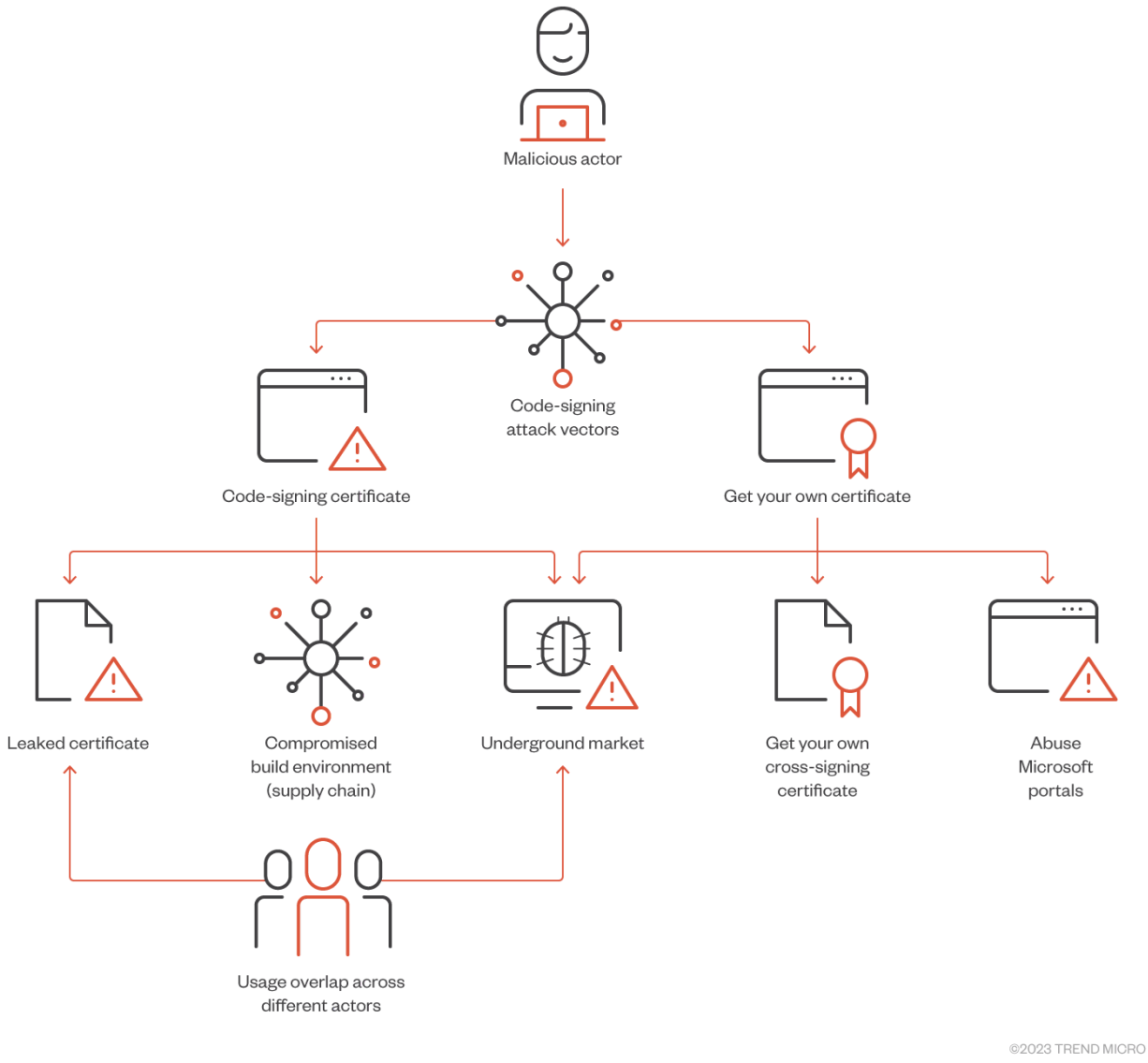

Figure 3. Diagram showing how a threat actor complies with Microsoft code-signing requirements

Analysis of a signed driver

In this section, we will examine a signed driver (*ktgn.sys*) used in the February BlackCat attacks. Figure 4 shows other examples of these new signed drivers and how they are being used as part of the BlackCat affiliate's defense evasion routine.

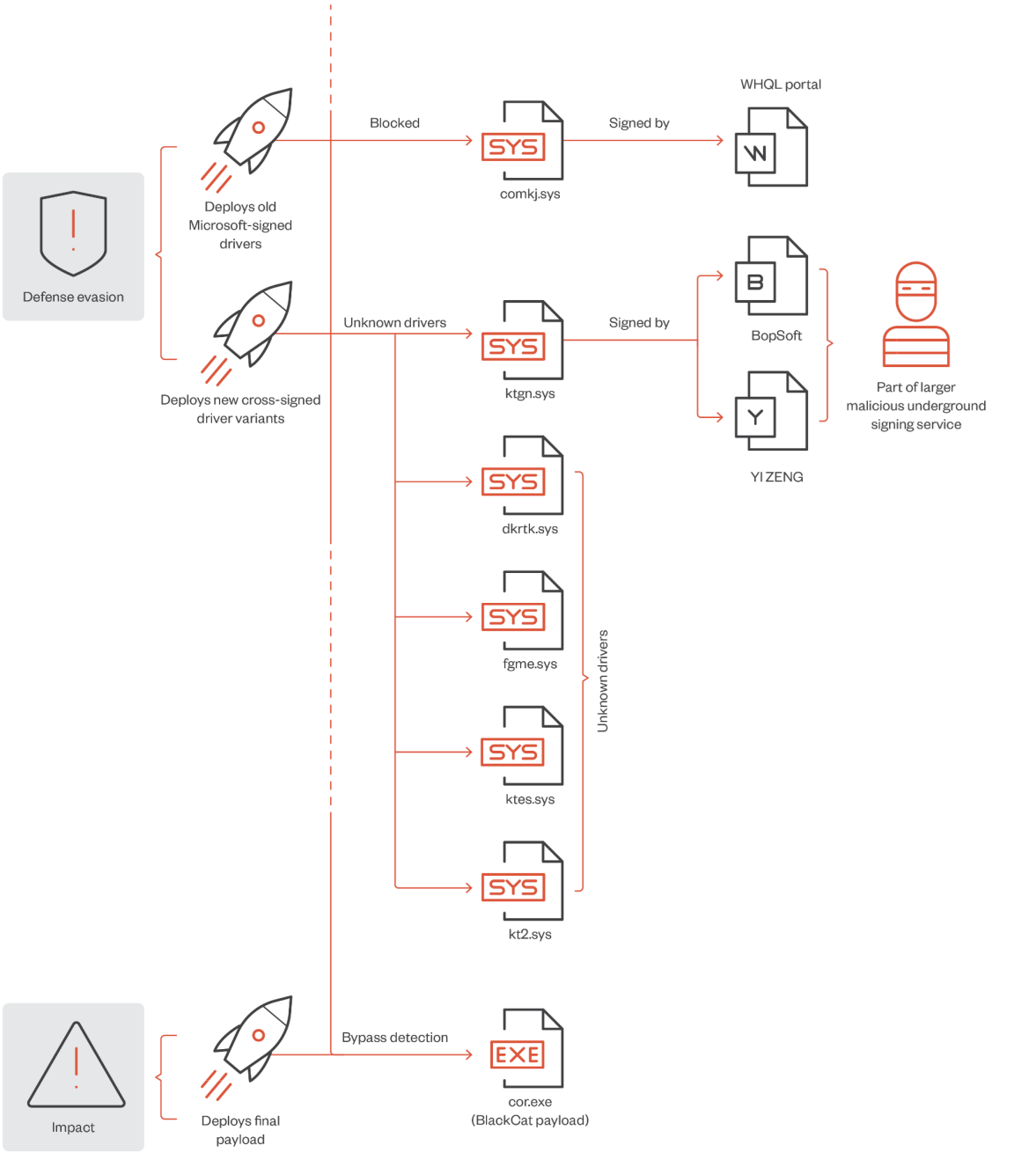

Figure 4. The dropped files by a BlackCat affiliate in the defense evasion phase

The User Agent *tjr.exe*, which is protected via a virtual machine, drops the kernel driver to the user temporary directory *C:\%User%\AppData\Local\Temp\Ktgn.sys*. It then installs the dropped driver with the name *ktgn* and the start value = System (to start when the system restarts). From our analysis of what occurs when a user interfaces with this driver, we observed that it only uses one of the exposed Device Input and Output Control (IOCTL) code — Kill Process, which is used to kill security agent processes installed on the system.

Meanwhile the driver *ktgn.sys*, which is signed using a currently revoked valid digital signature from “BopSoft” (which had also been previously used by other threat actors for code signing) can successfully be loaded on a 64-bit Windows installation where signing policies are enforced. The driver is obfuscated using *Safengine Protector v2.4.0.0* tool, which renders static analysis techniques unreliable. By loading the obfuscated driver and trying to build a user mode client to observe the exposed IOCTL interface, we can determine the function of each IOCTL code. Finally, we observed the same kernel driver being signed by different code-signing certificates.

Driver variants (SHA256)	Signer name	Valid usage	Current status	Issuer
994e3f5dd082f5d82f9cc84108a60d359910ba79	BopSoft	Code signing	Explicitly revoked by its issuer	Thawte
f6793243ad20359d8be40d3accac168a15a327fb	YI ZENG	Code signing	Explicitly revoked by its issuer	VeriSign

Table 1. The driver variants with different signers

```
.py:FFFFF801CC328EBB aAfengineShield db 'afengine Shielden v2.4.0.0',0
.py:FFFFF801CC328ED6
.py:FFFFF801CC328ED6 ; =============== S U B R O U T I N E =======================================
.py:FFFFF801CC328ED6
.py:FFFFF801CC328ED6
```

Figure 5. The packer used to obfuscate the binary

Since it does not register an unload callback function, the driver can only be unloaded if the service registry key is deleted or modified followed by a system restart.

```
C:\Windows\system32>sc start ktgn

SERVICE_NAME: ktgn
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
        PID                : 0
        FLAGS              :

C:\Windows\system32>sc stop ktgn
[SC] ControlService FAILED 1052:

The requested control is not valid for this service.


C:\Windows\system32>
```

Figure 6. The service cannot be stopped by the service control manager

```
1: kd> dt nt!_DRIVER_OBJECT ffff950a24d9e400
   +0x000 Type             : 0n4
   +0x002 Size             : 0n336
   +0x008 DeviceObject     : 0xffff950a`24bf72d0 _DEVICE_OBJECT
   +0x010 Flags            : 0x12
   +0x018 DriverStart      : 0xfffff801`d3890000 Void
   +0x020 DriverSize       : 0x9e000
   +0x028 DriverSection    : 0xffff950a`23d8d770 Void
   +0x030 DriverExtension  : 0xffff950a`24d9e550 _DRIVER_EXTENSION
   +0x038 DriverName       : _UNICODE_STRING "\Driver\ktgn"
   +0x048 HardwareDatabase : 0xfffff801`d4c6c778 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x050 FastIoDispatch   : (null)
   +0x058 DriverInit       : 0xfffff801`d3928eb5     long  +0
   +0x060 DriverStartIo    : (null)
   +0x068 DriverUnload     : (null)
   +0x070 MajorFunction    : [28] 0xfffff801`d3892b90     long  +0
```

Figure 7. The driver lacking an unload function

A symbolic link with the name \\.*keHeperDriverLink* is created that allows the user mode client to connect and communicate with it. Note that this link only allows for one connection — if more than one client tries to connect to it simultaneously, the system will crash.

```
ktgn+0x2b90:
fffff801`2c042b90 4883ec38        sub     rsp,38h
fffff801`2c042b94 803dc524000001  cmp     byte ptr [ktgn+0x5060 (fffff801`2c045060)],1   //is UserClient Connected
fffff801`2c042b9b 488bc2          mov     rax,rdx
fffff801`2c042b9e 751a            jne     ktgn+0x2bba (fffff801`2c042bba)  Branch

ktgn+0x2ba0:
fffff801`2c042ba0 488364242000    and     qword ptr [rsp+20h],0
fffff801`2c042ba6 4533c9          xor     r9d,r9d
fffff801`2c042ba9 4533c0          xor     r8d,r8d
fffff801`2c042bac 33d2            xor     edx,edx
fffff801`2c042bae b9220000c0      mov     ecx,0C0000022h
fffff801`2c042bb3 ff156f140000    call    qword ptr [ktgn+0x4028 (fffff801`2c044028)]    //nt!KeBugCheckEx
fffff801`2c042bb9 cc              int     3

ktgn+0x2bba:
fffff801`2c042bba c6059f24000001  mov     byte ptr [ktgn+0x5060 (fffff801`2c045060)],1
fffff801`2c042bc1 488bc8          mov     rcx,rax
fffff801`2c042bc4 83622000        and     dword ptr [rdx+20h],0
```

Figure 8. Checking if another user mode process is trying to connect to the driver

The exposed IOCTL Interface

This client supports ten different commands, with each command implementing a specific function that is executed from the kernel driver with the appropriate IOCTL interface exposed. Communication between the driver and the user mode client occurs using the IRP_MJ_DEVICIDE_CONROL handler via the following codes:

IOCTL Code	Description
222088h	Activate Driver
22208Ch	Deactivate Driver
222094h	Kill Process
222184h	Delete File
222188h	Force Delete File
22218Ch	Copy File
222190h	Force Copy File
2221C8h	Register Process/Thread Object notification
2221C4h	Unregister Process/Thread Object notification
222264h	Reboot the system

Table 2. Each IOCTL code and their function

Based on our analysis of the kernel driver, it seems to still be under development and testing since it is not structured well and some of its functions currently cannot be used. The following subsections provide details on the various IOCTL Interfaces.

IOCTL 222088h

IOCTL 222088h must be called first to activate the driver before any other operation can be performed. If this code is not called, the driver will not accept any operation and will return the message *STATUS_ACCESS_DENIED*. The user mode client sends this activation byte array to the driver.

The activation is a simple byte comparison against a hard coded byte array with the size 0x42 located in the driver. If the comparison passes, it will set a BOOLEAN flag, which will be checked before any operation.

```
1: kd> db fffff801`d3895000 L42
fffff801`d3895000  45 00 44 00 20 00 41 00-44 00 20 00 46 00 47 00  E.D. .A.D. .F.G.
fffff801`d3895010  20 00 48 00 47 00 20 00-47 00 46 00 20 00 54 00   .H.G. .G.F. .T.
fffff801`d3895020  52 00 20 00 53 00 59 00-20 00 55 00 54 00 20 00  R. .S.Y. .U.T. .
fffff801`d3895030  47 00 48 00 20 00 4e 00-47 00 20 00 47 00 54 00  G.H. .N.G. .G.T.
fffff801`d3895040  00 00                                            ..
```

Figure 9. Activation bytes from the running memory

```
//Activate the Driver
BYTE Activation[]{ 0x45 , 0x00 , 0x44 , 0x00 , 0x20 , 0x00 , 0x41 , 0x00 , 0x44 , 0x00 , 0x20 , 0x00 , 0x46 , 0x00 , 0x47 , 0x00
    , 0x20 , 0x00 , 0x48 , 0x00 , 0x47 , 0x00 , 0x20 , 0x00 , 0x47 , 0x00 , 0x46 , 0x00 , 0x20 , 0x00 , 0x54 , 0x00
     , 0x52 , 0x00 , 0x20 , 0x00 , 0x53 , 0x00 , 0x59 , 0x00  , 0x20 , 0x00 , 0x55 , 0x00 , 0x54 , 0x00 , 0x20 , 0x00
    , 0x47 , 0x00 , 0x48 , 0x00 , 0x20 , 0x00 , 0x4e , 0x00  , 0x47 , 0x00 , 0x20 , 0x00 , 0x47 , 0x00 , 0x54 , 0x00
      , 0x00 , 0x00 };

Status = DeviceIoControl(hDevice, (DWORD)0x222088, Activation, 0x42, NULL, NULL, &returned, nullptr);
```

Figure 10. Replicating the activation bytes to test driver operations

IOCTL 22208Ch

IOCTL 22208Ch is called after the user mode client finishes its operation to unset the flag that was previously set in IOCTL Code 222088h. This will deactivate the driver and stop it from processing any new operation.

The client will need to pass the same byte array passed in IOCTL code 222088h for the operation to be successfully completed.

IOCTL 222094h

IOCTL 222094h is used to kill any user mode process (even protected ones). Tt receives the Process ID from the user agent then creates a kernel thread in the target process context. The created kernel thread calls the **ZwTerminateProcess** API to terminate the target process.

```
ktgn+0x2cec:
fffff801`d3892cec 803d6e23000001  cmp     byte ptr [ktgn+0x5061 (fffff801`d3895061)],1    //check if process authenticated before allowing the operation
fffff801`d3892cf3 740a            je      ktgn+0x2cff (fffff801`d3892cff)  Branch

ktgn+0x2cf5:
fffff801`d3892cf5 bb220000c0      mov     ebx,0C0000022h                                  //if the Authentication flag is not set, return STATUS_ACCESS_DENIED
fffff801`d3892cfa e990020000      jmp     ktgn+0x2f8f (fffff801`d3892f8f)  Branch
```

Figure 11. Checking if the driver is activated

```
Status = DeviceIoControl(hDevice, (DWORD)0x222094, &PID, sizeof(DWORD), NULL, NULL, &returned, nullptr);
```

Figure 12. The IOCTL 222094h kill process.

IOCTL 222184h

IOCTL 222184h is used to delete specific file paths (as shown in Figure 11).

```
WCHAR FileToDelete[] = L"C:\\Users\\Test\\Desktop\\Test.exe";

Status = DeviceIoControl(hDevice, (DWORD)0x222184, FileToDelete, 0x3e, NULL, NULL, &returned, nullptr);
```

Figure 13. IOCTL 222184h deleting a file path

IOCTL 222188h

IOCTL 222188h is used to force delete files. To do this, the kernel driver does the following:

1. It tries to open all processes on the system using brute-force methods (starting from PID=0x4 to PID= 0x27FFD)
2. When it successfully opens a process, it tries to reference all handles inside the process, again using a brute-force method (starting from HANDLE=0x4 to HANDLE = 0x27FFD)
3. When it successfully references a handle, it uses the **ObQueryNameString** API to map the handle to a name. When a match is found, the kernel driver closes the handle.

This operation will ensure that all references to the file will be closed and the operation can be successfully completed without any errors stating that the file is being used by other applications.

```
ktgn+0x239b:
fffff807`d15f239b bb04000000      mov     ebx,4
fffff807`d15f23a0 bffd7f0200      mov     edi,27FFDh

ktgn+0x23a5:
fffff807`d15f23a5 488d542440      lea     rdx,[rsp+40h]
fffff807`d15f23aa 488bcb          mov     rcx,rbx
fffff807`d15f23ad ff152d1d0000    call    qword ptr [ktgn+0x40e0 (fffff807`d15f40e0)]              //PsLookupProcessByProcessId
fffff807`d15f23b3 85c0            test    eax,eax
fffff807`d15f23b5 0f888c000000    js      ktgn+0x2447 (fffff807`d15f2447)  Branch
```

Figure 14. Brute force PID

```
fffff807`d15f2161 bf04000000      mov     edi,4
fffff807`d15f2166 befd7f0200      mov     esi,27FFDh

ktgn+0x216b:
fffff807`d15f216b 488364242800    and     qword ptr [rsp+28h],0
fffff807`d15f2171 488d4530        lea     rax,[rbp+30h]
fffff807`d15f2175 4533c9          xor     r9d,r9d
fffff807`d15f2178 4889442420      mov     qword ptr [rsp+20h],rax
fffff807`d15f217d 4533c0          xor     r8d,r8d
fffff807`d15f2180 33d2            xor     edx,edx
fffff807`d15f2182 488bcf          mov     rcx,rdi
fffff807`d15f2185 ff15151f0000    call    qword ptr [ktgn+0x40a0 (fffff807`d15f40a0)]                  //ObReferenceObjectByHandle
fffff807`d15f218b 85c0            test    eax,eax
fffff807`d15f218d 787d            js      ktgn+0x220c (fffff807`d15f220c)  Branch
```

Figure 15. Brute force handles

IOCTL 22218Ch

IOCTL 22218Ch is used to copy files.

```
WCHAR Source[] = L"C:\\Users\\Test\\Desktop\\Source.exe";
WCHAR Destination[]= L"C:\\Users\\Test\\Desktop\\destination.exe";
CopyStruct CopyInfo = { 0 };
CopyInfo.Source = Source;
CopyInfo.Destination = Destination;
Status = DeviceIoControl(hDevice, (DWORD)0x22218C, &CopyInfo, sizeof(CopyStruct), NULL, NULL, &returned, nullptr);
```

Figure 16. IOCTL 22218Ch being used to copy a file

IOCTL 222190h

IOCTL 222190h is used to force copy files. The driver uses the same operation as the one used for force deletion (IOCTL Code: 222188h). It closes all references to the files from all processes using brute-force methods, then copies the file.

IOCTL 2221C4h and IOCTL 2221C8h

Both IOCTL 2221C4h and 2221C8h are used to register and unregister Process/Thread Notification callbacks. However, both paths are unreachable at the time of writing, which indicates that they are still under development or testing.

```
BOOLEAN ObjectFlag = 0;
if (ObjectFlag) {
    //Register process and thread object notification callbacks

    ObRegisterCallbacks(&ProcessReg, &ProcessHandle);
    ObRegisterCallbacks(&ThreadReg, &ThreadHandle);

    ObjectFlag = 1;
}
```

Figure 17. Pseudocode for Register Object Notification

```
BOOLEAN ObjectFlag = 0;
if (ObjectFlag) {
    //Unregister process and thread object notification callbacks
    ObUnRegisterCallbacks(&ProcessHandle);
    ObUnRegisterCallbacks(&ThreadHandle);

    ObjectFlag = 1;
}
```

Figure 18. Pseudocode for Unregister Object Notification

```
OB_PREOP_CALLBACK_STATUS OnPreOpen(PVOID /* RegistrationContext */, POB_PRE_OPERATION_INFORMATION Info) {

    Info->Parameters->CreateHandleInformation.OriginalDesiredAccess &= 0;
    Info->Parameters->CreateHandleInformation.DesiredAccess = STANDARD_RIGHTS_ALL | SPECIFIC_RIGHTS_ALL;
    Info->KernelHandle = Info->KernelHandle | 1;

    return OB_PREOP_SUCCESS;
```

Figure 19. Pseudocode for Object Notification Function

IOCTL 222264h

IOCTL 222264h Is used to reboot the system by calling the **HalReturnToFirmware** API.

Conclusion

Malicious actors that are actively seeking high-privilege access to the Windows operating system use techniques that attempt to combat the increased protection on users and processes via endpoint protection platform (EPP) and endpoint detection and response (EDR) technologies. Because of these added layers of protection, attackers tend to opt for the path of least resistance to get their malicious code running via the kernel layer (or even lower levels). This is why we believe that such threats will not disappear from threat actors' toolkits anytime soon.

Malicious actors will continue to use rootkits to hide malicious code from security tools, impair defenses, and fly under the radar for long periods. These rootkits will see heavy use from sophisticated groups that have both the skills to reverse engineer low-level system components and the required resources to develop such tools. These malicious actors also tend to possess enough financial resources to either purchase rootkits from underground sources or to buy code-signing certificates to build a rootkit. This means that the main danger involving these kinds of rootkits lie in their ability to hide complex targeted attacks that will be used early in the kill chain, allowing an attacker to impair defenses before the actual payloads are launched in victim environments.

Recommendations and solutions

Code signing certificates can often be abused by threat actors since they provide an additional layer of obfuscation in their attacks. For organizations, compromised keys present not only a security risk, but can also lead to a loss of reputation and trust in the original signed software. Businesses should aim to protect their certificates by implementing best practices such as reducing access to private keys, which reduces the risk of unauthorized access to the certificate. Employing strong passwords and other authentication methods for private keys can also help protect them from being stolen or compromised by malicious actors. Furthermore, using separate test signing certificates (for prerelease code used in test environments) minimizes the chances that the actual release signing certificates are abused in an attack.

For general ransomware attack protection, organizations can implement a systematic security framework that allocates resources towards establishing a robust defense strategy. Here are some recommended guidelines:

- Take inventory of assets and data
- Identify authorized and unauthorized devices and software
- Audit event and incident logs
- Manage hardware and software configurations
- Grant admin privileges and access only when necessary
- Monitor network ports, protocols, and services
- Establish a software allowlist for legitimate applications
- Implement data protection, backup, and recovery measures
- Enable multifactor authentication (MFA)
- Deploy the latest versions of security solutions across all layers of the system
- Watch for early signs of an attack

By adopting a multifaceted approach to securing potential entry points, such as endpoints, emails, webs, and networks, organizations can detect and protect against malicious elements and suspicious activities, thereby safeguarding themselves from ransomware attacks.

A multilayered approach can help organizations guard possible entry points into their system (endpoint, email, web, and network). Security solutions can detect malicious components and suspicious behavior, which can help protect enterprises.

Indicators of Compromise

The indicators of compromise for this entry can be found here.