

Auto-Start Entry Point (ASEP) Methods

Applnit_DLLs: All the DLLs that are specified in this value are loaded by each Microsoft Windows-based application that is running in the current log on session. The Applnit DLLs are loaded by using the LoadLibrary() function during the DLL_PROCESS_ATTACH process of User32.dll. Therefore, executables that do not link with User32.dll do not load the Applnit DLLs. Therefore one of the 16 imports was user32.dll (Import table (libraries: 16))

Because of their early loading, only API functions that are exported from Kernel32.dll are safe to use in the initialization of the Applnit DLLs. So murka.dat registers itself in the "Applnit_DLLs" as a load point for the beep.sys so that every time the computer starts, the RootKit driver can load itself with the kernel...

Example:

Trojan.Virantix.B

When the Trojan is executed, it creates following files:

```
%System%\user32.dat
%Windir%\medichi.exe
%Windir%\medichi2.exe
%Windir%\murka.dat
```

It then overwrites following files:

```
%System%\beep.sys          |
%System%\dllcache\beep.sys | <---- Actual driver that later on uses RootKit feature to hide the process medichi.exe
```

Next, the Trojan creates the following registry entries so that it executes whenever Windows starts:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows"Applnit_DLLs" =
"%Windir%\murka.dat"
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run"Medichi" = %Windir%\medichi.exe"
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run"Medichi2" = "%Windir%\medichi2"
```

It then hooks the following API and hides itself: <--- RootKit activity
ZwQuerySystemInformation <--- Hooks the Native subsystems function call

The Trojan then connects to the following location, which displays a fake security alert:
<http://gomyhit.com/MTC3MTY=/2/6018/852/>

It also opens the following URL, which may contain another program:
<http://81.13.38.39/aler>

The Trojan monitors the browser on the compromised computer and steals search keywords that can be used on certain search engines and submits it to following remote location:
<http://werdagoniotu.com/searc>

It attempts to download updates of itself from the following locations:
<http://globalmenu.net/1/sert>
<http://softinfoway.info/1/sert>
<http://getupdate.info/1/sert>

RootKit Activity: Rootkits work by changing API results so that a system view using APIs differs from the actual view in storage. The highest level is the Windows API and the lowest level is the raw contents of a file system volume or Registry hive (a hive file is the Registry's on-disk storage format). Thus, Rootkits, user mode or kernel mode, manipulate the Windows API or native API to remove their presence from a directory listing. A kernel-mode RootKit can control any aspect of a system's behavior so information returned by any API, including the raw reads of Registry hive and file system, can be compromised.

Little More Details:

The early Rootkits were basically *application Rootkits*. They used to overwrite Windows user-mode application binaries with trojaned binaries. To hide a given piece of Malware effectively, the RootKit might need to overwrite many applications. This sledge-hammer technique was relatively crude and easily spotted by checksum-based PC security programs.

Instead of overwriting binaries, *User Mode Rootkits* can hide objects by intercepting (*hooking*) application calls to high-level Windows API functions. Most Windows applications rely on stock functions for a wide range of basic tasks, such as displaying dialog boxes or driving devices such as hard drives and printers. The stock Microsoft functions are grouped conceptually and stored in shared libraries, referred to as Dynamic Link Libraries (DLLs). Every developer can learn the locations of functions within DLLs -- and every attacker can exploit this same knowledge. For example, many applications use the kernel32.dll FindFirstFile function to explore files. A RootKit may change the application's Import Address Table, replacing pointers to the

location of kernel32.dll with RootKit DLL pointers. When the FindFirstFile function is invoked, the RootKit uses the kernel32.dll's FindFirstFile function, but filters the result to hide filenames that provide evidence of the Rootkits presence. This Windows API hooking technique can also be used to hide processes, sockets, services, and registry keys.

More sophisticated *User Mode Rootkits* exploit the slightly lower-level Native API which invokes functions provided by the operating system's ntdll.dll library. For example, Task Manager (iexplore.exe) uses the ntdll.dll **NtQuerySystemInformation** function to get a list of active processes. A RootKit can hook Task Manager's call to ntdll.dll, and then strip Malware processes from the returned list.

Some User Mode Rootkits *patch* a few bytes of in-memory Windows API code to insert a jump to the Rootkits DLL. As described above, the RootKit filters output objects before returning control to the compromised code, which then returns data to the application that invoked the Windows API or Native API. A malicious hacker can use in-memory patching to hide files, processes, services, drivers, registry keys and values, open ports, and disk space usage.

User Mode applications and DLLs access the *kernel* by using system calls to reach device drivers. Drivers have direct access to kernel data objects, including the Master File Table, the Registry Hive, and the kernel's active process table. *Kernel Mode Rootkits* hook these system calls, patch these device drivers, or modify kernel data objects to alter results at the lowest level. For example, *system call hooking* can modify the process list returned by the kernel to ntdll.dll, instead of changing the list returned by ntdll.dll to Task Manager. Due to complexity and tight-coupling, Kernel Mode Rootkits must take care to avoid crashing the compromised PC.

An increasingly popular technique used by Kernel Mode Rootkits is *Direct Kernel Object Manipulation (DKM)*. Instead of intercepting query and enumeration API calls, DKM modifies the kernel's own data structures. For example, a DKM RootKit might remove a Malware process from the kernel's process table. It might use syntactically-invalid names to hide folders and files from the Windows NT file system (NTFS). Or it might manipulate the Registry Hive to prevent enumeration of hidden registry keys; for example, embedding a NULL character in a key's name or value, hiding what follows from Windows API RegEnumValue or Native API **NtEnumerateKey** queries.

These methods can be found in many Windows Rootkits, including **AFX** (kernel32.dll hooking and patching), **Vanquish** (kernel32.dll patching), **Hacker Defender** (ntdll.dll patching), and **FU (DKM)**. For instance, FU loads the driver **msdirectx.sys**, which can then be used to hide Malware processes and device drivers. Unlike most Rootkits, FU does not attempt to hide itself, which may explain why FU is the most oft-removed RootKit encountered by Microsoft's Malicious Software Removal Tool.

Summary

They patch the Kernel level functions and load a driver in its place. The directory listing calls are thus intercepted by the RootKit driver and then forwarded to the actual NTDLL subsystems library function. When the control is again sent back to the RootKit driver from the actual Kernel function, it then filters the output thus hiding its presence.

In the above example, **%System%\dllcache\beep.sys** is the RootKit driver that patches the **ZwQuerySystemInformation**. It thus hides the presence of the below mentioned infected files:

%System%\user32.dat
%Windir%\medichi.exe
%Windir%\medichi2.exe
%Windir%\murka.dat

At times the RootKit driver patches the **ZwQuerySystemInformation** or the **NtQuerySystemInformation** Kernel function.

The native API, referred to as the NTDLL subsystem, is a series of undocumented API function calls that handle most of the work performed by KERNEL. The NTDLL subsystem is located in **ntdll.dll**. This library contains many API function calls, and all follow a particular naming scheme. Each function has a prefix: **Ldr, Ki, Nt, Zw, Csr, dbg**, etc. All the functions that have a particular prefix follow particular a rules.

The "official" native API is usually limited only to functions whose prefix is **Nt** or **Zw**. These calls are in fact the same and are used to provide function calls in both Kernel and User space. However User applications are encouraged to use the **Nt*** calls, while Kernel callers are supposed to use the **Zw*** calls. Writing drivers for Windows is allowed to use the Kernel-mode functions in NTDLL because these drivers operate at Kernel Level.

Master Boot Record (MBR) as its Auto-Start Entry Point (ASEP)

There are several binaries in the wild which try to install this RootKit. All the known variants are detected by Microsoft antimalware products using two generic signatures: **PWS: Win32/Sinowal.gen!C** and **PWS: Win32/Sinowal.gen!D**.

This Malware attempts to modify the MBR so that it can control what gets read from the disk into memory and execute very early in the boot process. After the modified MBR is executed, it reads additional malicious code into memory which modifies the NT kernel to force it to load a malicious driver that has been stored at the end of the physical disk (The driver will not be visible while the infected OS is running.). Once the driver is loaded into the kernel, it behaves just like a standard kernel

mode RootKit, providing covert and stealth network backdoor functionality by hooking low level APIs to attempt to avoid detection. Here are some interesting things about this Malware:

First, the installer for this RootKit needs to modify the MBR in order to ensure that the RootKit can persist across reboots. It does this by using the CreateFile API attempting to open "\Device\Harddisk0\DR0" for write access. Using the CreateFile API in this way (for direct / raw disk access) requires administrative privileges. So if you are logged into Windows as a standard user or if you are using Windows Vista with UAC enabled, even if you accidentally run the Malware installer or it runs via some exploit code, it will be running with insufficient privilege to modify the hard disks MBR; thus it will not be able to persist a system restart.

Next, the perceived strength of this new RootKit, its lack of a visible footprint in the registry and file system due to the use of the MBR as the ASEP, is also a big weakness! If you suspect that you have a system that is infected with this RootKit, to prevent it from loading, all that is required is to write a known-good copy of a master boot record back to the disk to prevent the RootKit driver from being loaded on the next reboot! Fortunately, we have made that a fairly painless process with the Windows Recovery Console and the 'fixmbr' command!

Here are some instructions for using the Windows Recovery Console:

Windows XP instructions: <http://support.microsoft.com/kb/314058> (just type 'fixmbr' in the console)

Windows Vista instructions: <http://support.microsoft.com/kb/927392> (just type 'bootrec.exe /fixmbr' at the console)

After restoring a known-good MBR to the hard drive, you should be able to start Windows and perform an on-line antivirus scan to detect and remove any of the Malware components or any other Malware that may have been installed on the system and hidden by the RootKit.

The main driver makes outbound HTTP connections to a particular hard-coded IP address or domain. We presume this is so that it can receive instructions and/or register with its overseer. It may also be able to receive instructions which allow it to act as an HTTP proxy, or to download and execute further Malware. The Malware makes similar connections to a number of domains which appear to be pseudo-randomly generated.

Example:

VirTool: WinNT/Sinowal.A

When the Malware is executed, it creates following files:

VirTool: WinNT/Sinowal.A creates an executable temporary file with the prefix 'ldo'
eg: "c:\Documents and Settings\User\Local Settings\Temp\ldo1.tmp". It then executes this file.

It launches itself, using another temporary file with the prefix 'cln' as the parameter
eg: --cp "c:\Documents and Settings\User\Local Settings\Temp\cln2.tmp".
This action creates a copy of the original file and "converts" the file to a DLL.

The newly created DLL is then loaded. It creates the following service:

```
ServiceName = "{7663B344-A474-4eff-A35D-F5BE7F6531B4}"
DisplayName = ""
StartType = SERVICE_DEMAND_START
BinaryPathName = "%SystemRoot%\System32\svchost.exe -k netsvcs"
```

It sets the following registry key so the DLL can run as a service:

Adds value: ServiceDll

With data: "c:\Documents and Settings\User\Local Settings\Temp\cln2.tmp"

To subkey: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\{7663B344-A474-4eff-A35D-F5BE7F6531B4}\Parameters

It starts the service; the service drops a driver to: "<system folder>\{4C35FFDF-5669-4e96-8F6B-6CE0C16B4331}" and installs it via the following registry modifications:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\{4C35FFDF-5669-4e96-8F6B-6CE0C16B4331}
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\{4C35FFDF-5669-4e96-8F6B-6CE0C16B4331}\ImagePath=\??C:\WINDOWS\System32\{4C35FFDF-5669-4e96-8F6B-6CE0C16B4331}
```

MBR Modification

The abovementioned mentioned file 'ldo1.tmp' is responsible for modifying the hard disk's MBR (Master Boot Record) and writing the main driver and driver loaders portion directly to disk.

It attempts to access the hard disk directly via a previously installed driver. If this fails, it then reverts to trying to access the hard disk directly via \\.\PhysicalDrive0.

The original MBR is then overwritten with malicious code. Additionally, the main driver is written to the end of the physical drive, from where it is loaded directly.

Once complete the Trojan sleeps for a random period of time between 15 - 30 minutes in length, after which it initiates a system shutdown. The dialog box displaying the countdown timer is hidden from the user.

Backdoor Functionality

The main driver makes outbound connections via HTTP to the following hard coded IP address: **74.86.208.140**. Presumably this is to receive instructions and/or register with a remote attacker. Static Analysis suggests that the main component can receive instructions which allow it to act as an HTTP proxy, or to download and execute further Malware. Currently, these domains resolve to the following IP address: **72.5.175.97**.

Rootkit Installation 1 - Loads a driver in via ZwSetSystemInformation API. A very old, known and effective way to install a rootkit.

Rootkit Installation 2 - Loads driver by overwriting a standard driver (beep.sys) and starting it with service control manager (e.g. Trojan.Virantix.B).

DLL Injection 1 - Injects DLL into trusted process (svchost.exe) by injecting APC on LoadLibraryExA with "dll.dll" as a param. The string "dll.dll" is not written into process memory, it's from the ntdll.dll export table which has the same address in all processes. The APC is injected into second thread of the svchost.exe which is always in alertable state.

DLL Injection 2 - An old technique. The DLL is injected via remote thread creation in the trusted process, without using WriteProcessMemory.

BITS Hijack - Downloads a file from the internet using "Background Intelligent Transfer Service" which acts from the trusted process (svchost.exe)

Written By:

Rajdeep Chakraborty

Aka. MaliciousBrains

Site : <http://www.malwareinfo.org>

Blog : <http://blog.malwareinfo.org>

Forum: <http://forum.malwareinfo.org>

Email: maliciousbrains@malwareinfo.org

There are no patches or service packs for ignorance!