Andrea Allievi
Senior Security Researcher 06/02/2012

Personal Firewall: We really live in a secure environment?

These weeks in our labs we have analyzed some personal firewall software to discover if they really do their job. I've developed a simple network driver that simulate a well-designed rootkit. The driver uses technologies like Windows Socket Kernel, old Transport Driver Interface (TDI), and complex Network Driver Interface Specification (NDIS) to communicate to external world using a network card (NIC). But before digging in Firewall analysis let's talk about Windows Network Architecture. This is necessary to understand how firewall operates…

Windows Networking Infrastructure (a brief introduction)

All Nt Operation systems adhere to the network OSI reference model. This model is composed of different network layer. Talking about OSI model is behind the scope of this article, the reader can find all information about OSI model everywhere in internet (Wikipedia for example). Windows OS is composed of several networking component:

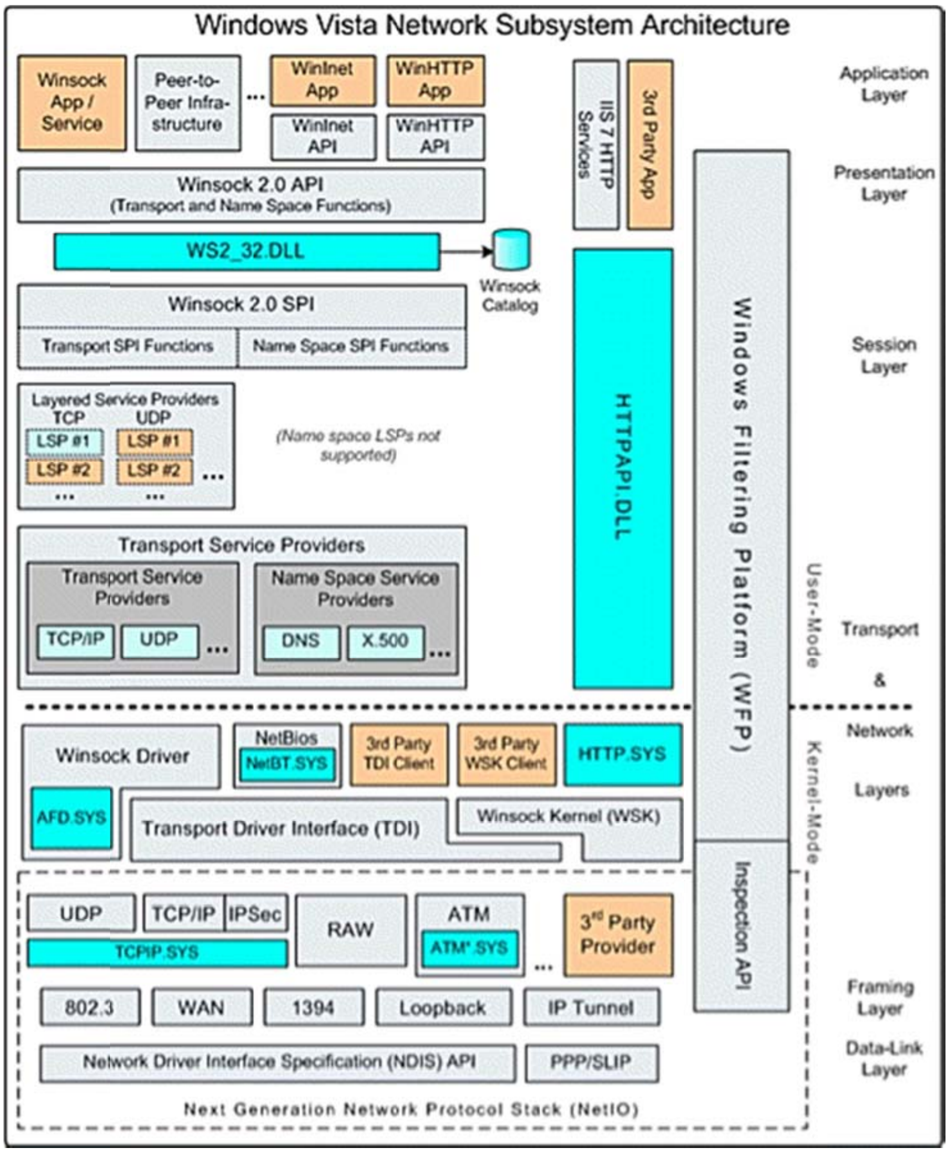
- Networking API – provide a protocol-independent way for applications to communicate across a network. It is implemented usually in user mode.
- Transport Driver Interface (TDI) clients – Are legacy kernel-mode device drivers that usually implement the kernel-mode portion of a networking API's implementation. They operate at network level of OSI Model and they are not used in Windows 7.
- Network Driver Interface Specification (NDIS) **protocol drivers** are kernel-mode protocol drivers that accept IRPs from TDI clients and process the requests these IRPs represent. This processing might require network communications with a peer, prompting the NDIS protocol driver to add protocol-specific headers (for example, TCP, UDP, and/or IP) to data passed in the IRP and to communicate with adapter drivers using NDIS functions.
- The NDIS library (Ndis.sys) provides encapsulation for adapter drivers, hiding from them specifics of the Windows kernel-mode environment. The NDIS library exports functions for use by NDIS protocol drivers.
- NDIS miniport drivers are kernel-mode drivers that are responsible for interfacing protocol drivers to particular network adapters.

Recent versions of Nt Kernel, like those of Windows Vista, Seven and future Windows 8, had replaced TDI Interface with 2 clever components:

- Winsock Kernel (WSK) is a transport-independent, kernel-mode networking API that replaces the legacy TDI mechanism. WSK provides network communication by using socket-like programming semantics similar to user-mode Winsock
- The Windows Filtering Platform (WFP) is a set of APIs and system services that provide the ability to create network filtering applications.
- NDSI Filter Driver, introduced in NDIS 6.0 specifications, provide filtering services for NDIS miniport drivers. This kind of filtering is more powerful than WFP because it operates at lower level (a level that correspond to Link OSI layer) but it has the big drawback in its complexity (like all NDIS Specs).

For example a packet sent from a TCP socket, start its life in an user mode application, go through Winsock API, it is filtered and processed by Transport service providers (and WFP) and then pass the

boundaries to Kernel Mode. Here it's managed by Winsock driver (AFD.sys) transformed in an IRP and sent directly to Winsock Kernel (WSK). WSK uses TCP/UDP driver (*tcpip.sys*, Microsoft Tcp/Ip NDIS protocol driver) to add TCP and IP headers, and sends entire packet to network adapter NDIS miniport driver (obviously before arriving to NDIS miniport, a packet is filtered by all installed NDIS Filter drivers in recent versions of Windows). NDIS at the end talk with hardware of network card and send Ethernet frame (that contains original packet) out from system.
The reader probably has understood that for a really well designed Firewall software, it's best to deploy a NDIS Filter driver, because WFP offer the ability to filter TCP, UDP connections, but not RAW links… NDIS however, as stated before, is really complex and it's really difficult (and furthermore a long work) to develop a serious NDIS filter driver.



Test Driver: How a possible rootkit can bypass firewall software

Our test is simple, we simulate a kernel mode rootkit that was installed without user consent in System. How a rootkit network component should do to bypass firewall software? We know that theoretically firewall filters TCP/IP traffic. After having installed our NDIS protocol driver we have set firewall to **block** all network traffic.
Our test NDIS protocol driver every 3 seconds send an Ethernet packet to a remote host and monitor replies from extern. If firewall was good, it should block either sent packet, and received replies.

Firewall Name	Default Configuration		"Block all" configuration	
	SEND	RECEIVE	SEND	RECEIVE
Windows XP Firewall	✗	✗	✗	✗
Windows Vista & 7 Firewall	✗	✗	✗	✗
Norton Internet Security 2012	✗	✗	✗	✗
Kaspersky Internet Security (Vista SP2)	✗	✗	✓	✗
Dr. Web Security Space 7	✗	✓	✓	✓
MCafee Total Protection	✗	✓	✓	✓
ZoneAlarm Firewall	✗	✓	✗	✓
ESET Smart Security	✗	✗	✓	✓
AVG Internet Security 2012	✗	✗	✗	✗
Trend Micro Internet Security 2012	✗	✗	✗	✗
F-Secure Internet Security	✗	✗	✗	✗

Keys:
✗ - Firewall doesn't block connection
✓ - Firewall blocks connection

Table 1 – Firewall test with NDIS Test driver.

Tests are made with Windows Xp, Vista and Seven. Results are impressive: Only 3 security suites completely do their job. A first-glance analysis show that others suites implements WFP drivers (or similar). **This is not completely enough**!

By the way this is not a full alarm. Implementing a fully functional NDIS protocol driver is very complex, and requires that rootkit developer implements a full Tcp/Ip Stack. This can be a big hurdle for the rootkit dimension and stealth. Only very few rootkits corrently uses NDIS to communicate with external world…

NDIS Protocol Driver – Development guidelines

With NDIS library you can develop 3 types of driver: Miniport driver - links network card with NDIS Library; Protocol driver – uses any network card (cabled and WiFi NICs) to communicate, and optional implement a particular network protocol; Filter driver – can filter and monitor all traffic directed to a network card. Each NDIS driver type require different developing modality. We are interested in protocol drivers.

Implementing an NDIS Protocol driver job starts in *DriverEntry* routine. You have to choose which version of NDIS use defining relative NDISxx symbol (see *ndis.h* WDK header file) in the driver main header file. First of all you have to allocate and compile an `NDIS_PROTOCOL_CHARACTERISTICS` structure. This structure has to be passed to *NdisRegisterProtocol(Ex)* routine that initializes NDIS Library and binds NDIS Protocol driver to network adapters.
Binding is the process that associate an NDIS protocol driver with a particular Network card. You cannot send and receive data from an unbinded network card. If the reader has disassembled NdisRegisterProtocol function, he can understand many things:

- *NdisRegisterProtocol* add to an internal Queue an object that contains protocol name that the developer has specified in `NDIS_PROTOCOL_CHARACTERISTICS` structure and then return to caller.
- *ndisWorkerThread* internal function, for each element in the queue, call internal *ndisCheckProtocolBindings*.
- *ndisCheckProtocolBindings* has the task to traverse internal miniport driver list, and for each miniport driver, enumerates and checks every network device that belong to miniport driver. It checks network card's "Upperbind" reg value. This value is the **key** for the entire binding process. If its data (of REG_MULTI_SZ type) contains developed protocol name, it means that the protocol has to be binded to that network card. It finally calls *ndisInitializeBinding* to start real binding process…

In summary, you can bind protocol driver to each network card adding protocol name to each network card registry value "HKLM\SYSTEM\CurrentControlSet\Control\Class\{4D36E972-E325-11CE-BFC1-08002BE10318}\00xx\Linkage\UpperBind" (where xx letters are network card number) and then you can start your driver. To enumerate all physical network cards (those that are listed on Control Panel) you can enumerate registry subkeys in "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkCards" and get *ServiceName* reg value data of each subkey.

Before call *NdisRegisterProtocol* you obviously have to initialize driver as usual, setting its dispatch functions, creating all devices and symbolic links, etc…

If the bind process has finished correctly and developed protocol driver has bound to at least one adapter card, NDIS Library calls *ProtocolBindAdapter* functions specified in `NDIS_PROTOCOL_CHARACTERISTIC`. In this driver function you have to construct all Binding data structure needed by driver, packets pool and buffers pool, and finally call *NdisOpenAdapter(Ex)* function. At this stage you are ready to send and receive Ethernet frames…

SENDING AND RECEIVING ETHERNET FRAMES

Sending a Ethernet frame (RAW Packet) with NDIS is quite simple. You have to start with a network binding handle returned by *NdisOpenAdapter(Ex)* function, allocate an Ndis buffer with *NdisAllocateBuffer* (or *NdisAllocateMdl*) function, allocate an Ndis Packet (*NdisAllocatePacket* or *NdisAllocateNetBuffer* API), link buffer and packet together, and then send packet with *NdisSendPackets* (or *NdisSendNetBufferLists*). That's all!

Receiving Ndis frames is a bit more complex. First of all, with *NdisRequest* function (and `OID_GEN_CURRENT_PACKET_FILTER` set in Ndis request structure) you have to tell network adapter miniport driver that your protocol driver would like to receive incoming frames from a miniport driver. After do this, Protocol driver *ProtocolReceive* and *ProtocolReceivePacket* functions will be called every time an incoming frame reaches network adapter.

These 2 functions (superseded by *ProtocolReceiveNetBufferLists* in NDIS 6) have to be implemented in this way: check Frame header (length and source MAC address), allocate enough buffer to copy data, and copy received frame data. If received frame is entire in `LookaheadBuffer` pointer, you don't have to call *NdisTransferData,* otherwise call that function to recover entire packet content.

Windows Socket Kernel Test

In WSK test we have worked with a Windows Socket Kernel driver (TDI for Windows Xp). This kind of driver uses kernel socket structures to communicate with a network remote host. WSK and TDI operates at Network OSI layer, indeed can communicate using TCP/IP protocol. Our driver connects to a dedicated webserver and downloads a test web page.
WSK is a brand new technology and is very easy to program and utilize (furthermore is also well documented by Microsoft).
Theoretically a good firewall software should be able to block every WSK communication because WSK operate at a quite high kernel layer.

Firewall Name	**Default Configuration**		**"Block all" configuration**	
	SEND	**RECEIVE**	**SEND**	**RECEIVE**
Windows Vista & 7 Firewall	✗	✗	✗	✗
Norton Internet Security 2012	✗	✗	✓	✓
Kaspersky Internet Security (Vista SP2)	✓	✓	✓	✓
Dr. Web Security Space 7	✓	✓	✓	✓
MCafee Total Protection	✗	✗	✓	✓
ZoneAlarm Firewall	✓	✓	✓	✓
ESET Smart Security	✗	✗	✓	✓
AVG Internet Security 2012	✓	✓	✓	✓
Trend Micro Internet Security 2012	✓	✓	✓	✓
F-Secure Internet Security	✗	✗	✗	✗

Keys:
✗ - Firewall doesn't block connection
✓ - Firewall blocks connection

Table 1 – Firewall test with WSK Test driver.

Tests with many security suite available on market are quite impressive: results demonstrate that our perception about WSK is true, 80% of firewalls can really block every WSK Kernel connections. The main problem is that not **ALL** firewalls block our simple connections and, most important, majority of them used with default configuration doesn't block our test download. They should be work in this manner because a connection started from kernel mode isn't badly in theory. Kernel software is part of trusted computing base and runs in "SYSTEM" account, the most powerful one and without any restrictions applied to it. The main fact is that any malware software shouldn't be able to load code in kernel mode.

Even if I personally hate technology like Kernel Patch Protection or Code Signing Enforcement, that is a nightmare for power users, I unfortunately confirm: technologies like those enforces very much the integrity of the System preventing malware to be able to load kernel code.

WSK Driver – Development guidelines

It's too much easy to develop a Windows WSK driver. I personal not write here how to develop this kind of driver because it's very easy and I don't have much time. The reader can take a glance at Microsoft official documentation available on MSDN (http://msdn.microsoft.com/en-us/library/windows/hardware/ff556958%28v=vs.85%29.aspx) or read "The Rootkit Arsenal - Escape and Evasion in the Dark Corners of the System" book for example, that provide a good introduction on WSK (like many others security related books do).

Conclusions

In my brief analysis we have seen that **we don't live** completely in a safe environment with a lot of software firewalls. This would be a big issue if all rootkits implemented "NDIS kind" convert channel to communicate with C&C… but fortunately this is far away for reality. In our days the major of rootkits uses only TDI and WSK technology to connect to external world, that majority of firewalls filter and intercept.
By the way a very well designed malware can leverage the NDIS firewalls flaw and act undisturbed in system (keep in mind however that malware has to overcome limitations of loading digitally unsigned code in Kernel mode, a big stake produced by technologies like Patchguard and Driver Signing Enforcement).
As conclusion we can say that to fully protect a company for network attacks and malware threats, the good and well-formed IT security specialist should use Hardware Firewall systems…

Last revision: 06/02/2012
Andrea Allievi