CODEDGERS

Rootkits: Attacking Personal Firewalls

Alexander Tereshkin, Senior Research Engineer, Codedgers, Inc.
the90210@codedgers.com

Introduction

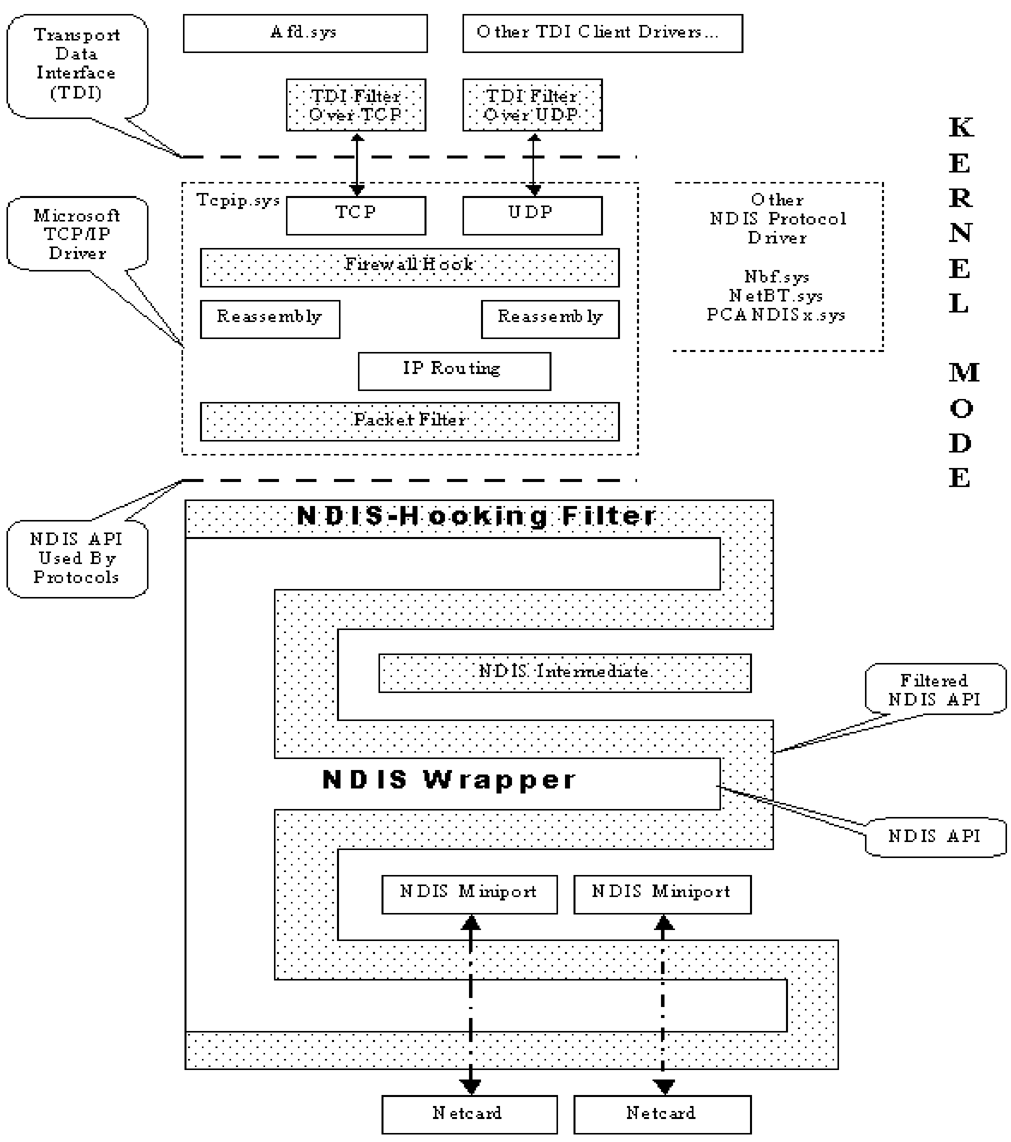
Current personal firewalls are focused on combating usermode malware

- What about protection against rootkits?

Overview

- i386 Windows NT+ network subsystem overview
- What malware authors usually do to cheat firewalls
- Common firewall techniques
  - Bypassing typical firewall hooks with no code patching
- Advanced firewall techniques
  - DKOM solutions to bypass modern firewalls
  - Live demo
- How to make firewalls resistant to the discussed attacks

Windows NT Network Subsystem Overview



Common Firewall "Bypassing" Techniques

Code injection into trusted process

- Malware finds trusted process and tries to inject code into it
  - Firewalls evolve to catch various types of code injections

Prevention of firewall drivers from loading

- Rootkit registers an image load notification callback via `PsSetLoadImageNotifyRoutine`()
- The callback checks for known driver images and counteracts their loading (e.g. by patching `XOR EAX, EAX / RET 0x08` at their entry point)

➢ **These techniques do not actually bypass firewalls – they cheat them. They are either firewall implementation specific or take advantage of incompetence of a user (i.e. weak firewall rules exploitation)**

TDI hooking

- Allows to implement per-process traffic monitoring and filtering connection attempts and packets of connectionless protocols made by upper-level socket interfaces
- High level TDI interfaces may be used by a firewall to simplify the detection and prevention of attacks against application layer protocols

NDIS hooking

- Allows to implement protection against attacks targeted from data link layer (e.g. Ethernet specific attacks) to transport layer (TCP protocol attacks). TDI hooks cannot prevent data link layer attacks
- It makes possible to hook unknown protocols' traffic (for example it may be used to switch system to "network stealth" mode)

Common Firewall Techniques – TDI hooking

Attaching to `\Device\Ip`, `\Device\RawIp`, `\Device\Tcp`, `\Device\Udp`

- Perform per-process traffic monitoring

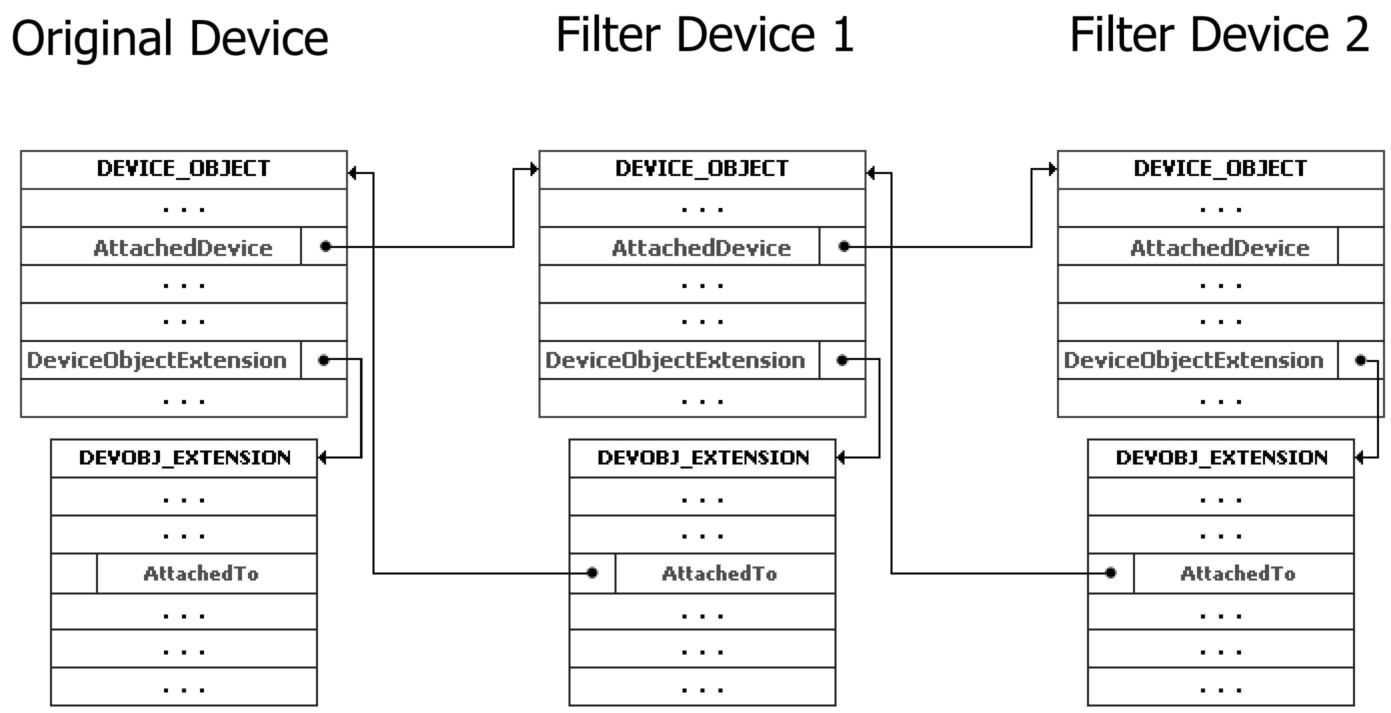
Techniques used

- Device filtering
  - Find a real device in the filter chain (lowest one)
- `DRIVER_OBJECT.MajorFunction[]` hooking
  - Perform a tunneling and find real `TCPIP.SYS` handlers

➢ It's too high level to pose obstacle for a rootkit

- It can block only rootkits that are using kernel mode sockets interface
- Greg Hoglund suggested using personal TCP/IP stack in 2001

Device Filtering – Finding Original Device



MajorFunction[ ] Hooking – Tunneling (1)

Walk relocation information of the `TCPIP.SYS`, store all absolute labels

- Drivers must have relocation info. Therefore, all specified MajorFunction elements must have DREFs in the driver image

Hook Int 01 in the IDT

- Should be done in SMP-safe way: all existing IDTs must be hooked

Catch a thread which is going to call `IoCallDriver()` to the TDI filter device, set Trace Flag in this thread

- No code patching is required: change `pIofCallDriver` pointer (it can be found easily – `IofCallDriver()` is exported) like Driver Verifier or IRP Tracker does. Edgar Barbosa used `pIofCallDriver` hooking to bypass VICE

MajorFunction[ ] Hooking – Tunneling (2)

Trace the thread until it comes to

- one of the absolute labels of the `TCPIP.SYS`
  - Original `MajorFunction[IrpStack->MajorFunction]` found
- `IopInvalidDeviceRequest()` in the `ntoskrnl.exe`
  - This MajorFunction was not specified by the `TCPIP.SYS`
- the caller (`IoCallDriver()` returns)
  - TDI filter has denied something, or current IRP is pending. Wait for another `IoCallDriver()`

Remember original MajorFunction[ ] value if it was found, clear TF

- Now rootkit is able to call original MajorFunction directly, without `IoCallDriver()`. It should adjust IRP stack locations manually

Unhook IDT if all required MajorFunction[ ] entries have been found

Common Firewall Techniques – NDIS Hooking

Hooking `NdisRegisterProtocol(),NdisOpenAdapter(), NdisDeregisterProtocol(),NdisCloseAdapter()`

- Catch all known (read, TCP/IP) protocol registrations
  - Patch `NDIS_PROTOCOL_CHARACTERISTICS`
- Catch all protocol bindings
  - Patch returned `NDIS_OPEN_BLOCK`

Techniques used

- `NDIS.SYS` export table patching
- `NDIS.SYS` code patching

➢ **Still not a problem for a rootkit**

NDIS Hooking – Typical Scenario

Driver calls `NdisRegisterProtocol()`

- Firewall checks for known protocol name (usually TCPIP, RASARP and TCPIP_WANARP) and then patches `NDIS_PROTOCOL_CHARACTERISTICS` with its own handlers
  - Some internal NDIS macros call functions by pointers from `NDIS_PROTOCOL_CHARACTERISTICS` and not from `NDIS_OPEN_BLOCK`

Driver calls `NdisOpenAdapter()`

- Firewall calls original `NdisOpenAdapter()`, and if it succeeds, patches `(PNDIS_OPEN_BLOCK)*NdisBindingHandle` code pointers
  - SendHandler, SendPacketsHandler
  - RequestHandler
  - TransferDataHandler
  - …

NDIS AntiHooking – Bypassing Firewall Hooks

Rootkit may patch its own handlers over the firewall hooks in the `NDIS_OPEN_BLOCK` of a certain protocol binding

- Used in "DeepDoor" by Joanna Rutkowska and "Peligroso" by Greg Hoglund
- This may work for simple firewalls, but more advanced ones will check their hooks for presence (subsequent `NDIS_OPEN_BLOCK`s checks) and integrity (i.e. splices/detours of their handlers)

How to register a protocol which will not be noticed by a NDIS-hooking firewall?

Bypass firewall hooks!

- It's a good idea to leave hooks intact, so that firewall will notice nothing. Active antihooking may trigger the defense subsystem of the firewall

These hooks may be either EAT-based or direct code patches in the `NDIS.SYS`

- EAT hooks may be defeated by finding original API addresses
- Direct code hooks may be defeated with the code pullout technique

NDIS EAT AntiHooking – Using Original Functions

Load `NDIS.SYS` file image from the disk

- Assume that disk IO is not hooked. Bypassing disk IO hooks is beyond the scope of this presentation ☺

Map image sections to appropriate virtual addresses

- This step may be skipped if we're going to translate Relative Virtual Addresses to Relative Physical Addresses using virtual section table each time we encounter a RVA. Reason: saving of memory

Walk export table and find needed RVAs

- There's no `GetProcAddress()` equivalent in the kernel (`MmGetSystemRoutineAddress()` can be used only for ntoskrnl and hal exports)

Apply found RVAs to the original `NDIS.SYS` image, don't rely on the import table anymore

- Make sure that API code is not hooked

NDIS Code AntiHooking – Using New NDIS Image

It's possible to load `NDIS.SYS` image from the disk with our own PE loader and make calls into this image

- Map `NDIS.SYS` file image sections to appropriate virtual addresses in the nonpaged memory
- All absolute pointers must be rebased to the existing `NDIS.SYS` image: we want our new hook-free code to use existing NDIS data

Advantages

- Initialization speed: we should perform a few fairly simple operations to make things up and running
- The loaded code will always be 100% correct – it is a clone of the running NDIS
- The technique is portable: there's no need to implement different PE loader for every processor which OS supports

Disadvantages

- Code size: we're going to use only few functions, but load the whole NDIS
- New code is identical to the original `NDIS.SYS`: a memory scanner could detect a copy

NDIS Code AntiHooking – Code Pullout Overview

More intelligent solution: build a sufficient NDIS code subtree

- Again, absolute pointers should be fixed to the existing NDIS image

Advantages

- Generated code size is much smaller than the full NDIS image
- NDIS code may be mutated with any polymorphic algorithm, signatures will be broken
- If we have to perform a search for a not-exported symbol based on code XREFs or other dependencies, the searching process may be combined with code walking to improve the performance

Disadvantages

- Initialization speed: there is a number of time-consuming operations
- It is theoretically possible that we encounter instructions that our disassembler will not be able to decode: the disassembler engine must understand as many instructions subsets as possible
- It's architecture-dependent: one has to implement code coverage and rebuilding tools for every supported processor

NDIS Code AntiHooking – Code Pullout (1)

PE loader maps new virtual image of the NDIS.SYS from the disk

- Don't care about relocs – they will be fixed later
- Do not use `MmCreateSection()` with a `SEC_IMAGE` allocation attribute: original section mapper (`MiCreateImageFileMap()`) may be hooked

Entry points for the subtree are defined as RVAs of the needed APIs

- All subtrees will intersect with each other over the shared code – generated code should not be redundant

Engine builds a code coverage map: each queued branch is being statically walked with a disassembler

- We stop on `RET`, `IRETD`, unpredictable control transfer (like `JMP reg32`) or when we come to the code that has been already analyzed. Calls and conditional jumps "fork" execution flow – they add branches to later analysis. Subtree coverage map is complete when there is no more branches left in the analysis queue

NDIS Code AntiHooking – Code Pullout (2)

All contiguous regions of the covered code are copied to one chunk of memory one after another without gaps

- Here we recalculate entry points for the addresses which were specified as the top of the original code subtrees (NDIS APIs in our case)
- This is where polymorphic methods may be applied to get rid of any static code signatures

Engine relinks all relative jumps and calls in the generated code

- All relative instructions that connect non-adjacent code blocks were damaged while merging a coverage

Relocations are fixed to the original NDIS image

Modern Firewall Techniques

Registration of a dummy protocol for walking protocols list

- Will spot new protocols without hooking, thus leaving antihooking methods useless

Periodical checks of the `NDIS_OPEN_BLOCK`s code pointers integrity

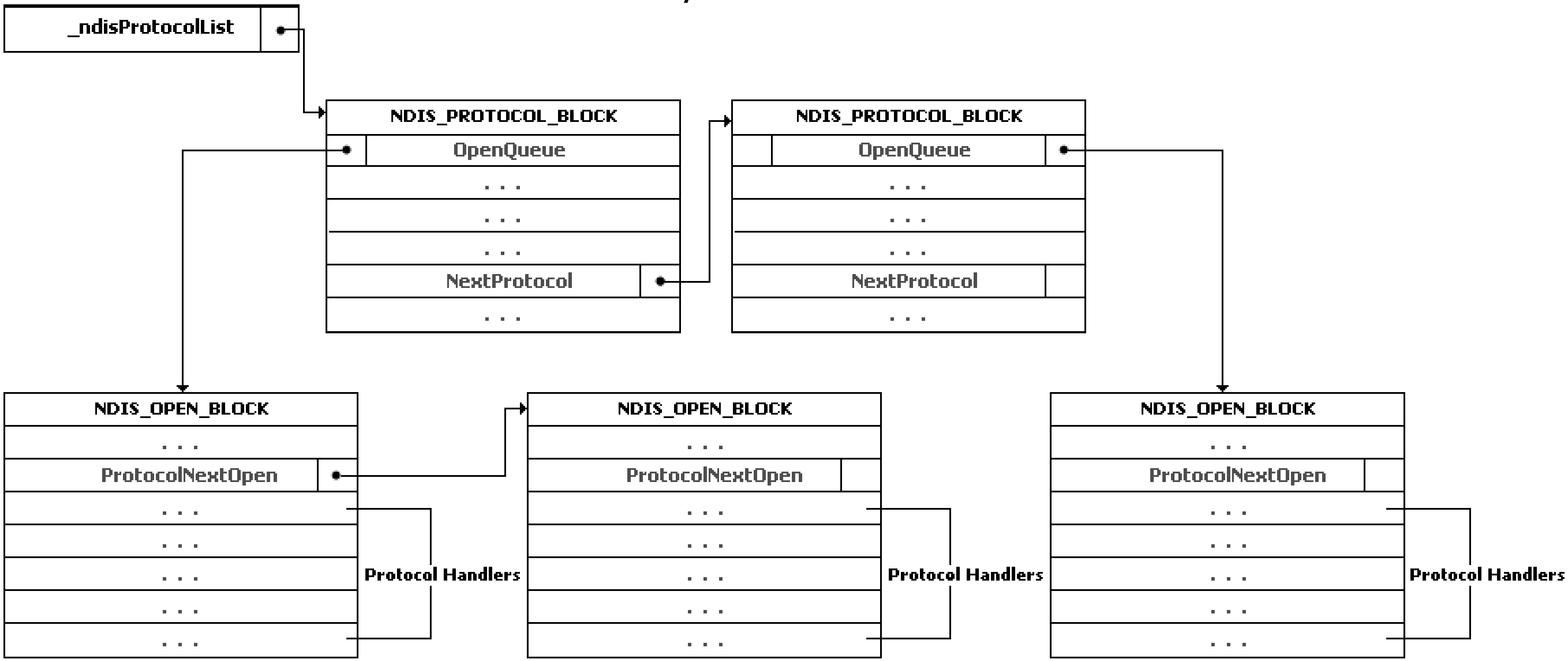
- “DeepDoor” and “Peligroso” rootkits will lose their hooks

Anti-splice and anti-detours tricks

- Various control data is addressed in trampolines via PIC code with the help of EIP-based deltas: direct detours will change the logic of the firewall trampoline, which may lead to BSOD or rootkit compromise

`NdisRegisterProtocol()` returns valid `NDIS_PROTOCOL_BLOCK` pointer which is first in the protocols list

- Walking list this way is dangerous! `ndisProtocolListLock` must be acquired, otherwise a race condition may occur



Walking Protocols – Stable Way

The right solution will be to use real `ndisProtocolList` and `ndisProtocolListLock`

- The problem: they are not exported
- `ndisProtocolList` is singly linked, so we can't walk it backwards to find a head
- To be sure that firewall is not cheating us, we again will use static analysis of the `NDIS.SYS` file
- Here's a fact: both these global variables are used by the `NdisRegisterProtocol()`

First, enumerate all absolute pointers in the `NdisRegisterProtocol()` execution tree

- Eliminate all IAT pointers from this list

Now check, which global variable from the list is ever used as a `PKSPIN_LOCK` by examining calls to `KfAcquireSpinLock()` and such

- From NT4 till 2003 Server there will be just one spin lock – the `ndisProtocolListLock`

Acquire found spin lock and check other global variables – do they look like a head of a `NDIS_PROTOCOL_BLOCK` singly linked list

- Some memory forensics required!

NDIS Internals – ndisProtocolList Usage

NdisRegisterProtocol()

- Places new NDIS_PROTOCOL_BLOCK at the head of the ndisProtocolList

NdisDeregisterProtocol()

- Removes protocol from the list

ndisReferenceProtocolByName()

- ndisCheckAdapterBindings()
- ndisHandleProtocolReconfigNotification()
- ndisHandleProtocolUnloadNotification()
- ndisHandleProtocolBindNotification()
- ndisHandleProtocolUnbindNotification()

ndisDereferenceProtocol()

- Decrements reference counter and frees NDIS_PROTOCOL_BLOCK if it reaches zero
- Does not walk ndisProtocolList if the protocol remains referenced

ndisPnPDispatch()

- Checks for empty ndisProtocolList before calling ndisQueueBindWorkitem()

ndisCheckAdapterBindings()

➢ **ndisProtocolList is not used by the packet indication code**

NDIS Internals – Registering a Filter (1)

`NdisOpenAdapter()` updates a corresponding miniport filter database (`ETH_FILTER` for ethernet in NT4/2000, `X_FILTER` structure in XP+)

- Database is selected using `Miniport->MediaType` value
  - `Miniport->EthDB` for ethernet
  - `Miniport->TrDB` for token ring
  - `Miniport->FddiDB` for fiber optic
- For ARCnet miniports `ARC_FILTER` is used instead of `X_FILTER`; the filter database is at `Miniport->ArcDB`

```
struct _X_FILTER {       // XP SP2
   /*<+0x0>*/ /*|0x4|*/ struct _X_BINDING_INFO* OpenList;
   /*<+0x4>*/ /*|0x210|*/ struct _NDIS_RW_LOCK BindListLock;
   /*<+0x214>*/ /*|0x4|*/ struct _NDIS_MINIPORT_BLOCK* Miniport;
   /*<+0x218>*/ /*|0x4|*/ unsigned int CombinedPacketFilter;
   /*<+0x21c>*/ /*|0x4|*/ unsigned int OldCombinedPacketFilter;
   /*<+0x220>*/ /*|0x4|*/ unsigned int NumOpens;
   /*<+0x224>*/ /*|0x4|*/ struct _X_BINDING_INFO* MCastSet;
   /*<+0x228>*/ /*|0x4|*/ struct _X_BINDING_INFO* SingleActiveOpen;
   /*<+0x22c>*/ /*|0x6|*/ unsigned char AdapterAddress[6];
   …
```

NDIS Internals – Registering a Filter (2)

- **XNoteFilterOpenAdapter()/EthNoteFilterAdapter() attaches new ETH_BINDING_INFO/X_BINDING_INFO to the selected filter database**
  - Current `NDIS_OPEN_BLOCK` pointer is stored there
- **This way NDIS saves information about NDIS_OPEN_BLOCK bindings to the particular NDIS_MINIPORT_BLOCK**
  - NDIS does not use `ndisProtocolList` to find an open binding on any network event, but firewalls do (indirectly): they get information about bindings by walking the protocol list

```
struct _X_BINDING_INFO {          // XP SP2
   /*<+0x0>*/ /*|0x4|*/ struct _X_BINDING_INFO* NextOpen;
   /*<+0x4>*/ /*|0x4|*/ struct _NDIS_OPEN_BLOCK* NdisBindingHandle;
   /*<+0x8>*/ /*|0x4|*/ void* Reserved;
   /*<+0xc>*/ /*|0x4|*/ unsigned int PacketFilters;
   /*<+0x10>*/ /*|0x4|*/ unsigned int OldPacketFilters;
   /*<+0x14>*/ /*|0x4|*/ unsigned long References;
   …
```

NDIS Internals – Managing Received Packets

Ethernet packet managers

- `ethFilterDprIndicateReceivePacket()`
  - Gets `X_FILTER` pointer by looking into `PNDIS_MINIPORT_BLOCK` (`Miniport->EthDB`), which is its first parameter
- `EthFilterDprIndicateReceive()`
  - For legacy miniports only
  - Gets `ETH_FILTER/X_FILTER` pointer as the first parameter

Managers walk `ETH_BINDING_INFO / X_BINDING_INFO` lists and indicate packets to the appropriate protocols

- NDIS doesn't care about `NDIS_PROTOCOL_BLOCK`s here; only `NDIS_OPEN_BLOCK`s matter (`X_BINDING_INFO.NdisBindingHandle`)

➢ **Therefore, NDIS may indicate the packets to the protocol which is not present in the `ndisProtocolList`**

Bypassing Protocol Walking – Approaches

New protocol registration: Code Pullout + DKOM methods

- We should exclude our protocol from the `ndisProtocolList`: it will remain functional, but a firewall won't be able to find it using list walking
- **Approach I**: call hook-free versions of `NdisRegisterProtocol()`, `NdisOpenAdapter()` and then unlink `NDIS_PROTOCOL_BLOCK` from the list. Very similar to process hiding via `PsActiveProcessHead` elements unlinking
- **Approach II**: modify copied `NdisRegisterProtocol()` and `NdisOpenAdapter()` code trees

Without new protocol: nothing to hide

- We should establish hooks over the existing protocols; hooking `NDIS_OPEN_BLOCK`s is too high level
- **Approach III**: hook existing `ETH_BINDING_INFO` / `X_BINDING_INFO`s
- **Approach IV**: register new `ETH_BINDING_INFO` / `X_BINDING_INFO` manually

Bypassing Protocol Walking I – Protocol Unlinking

Unhook `NdisRegisterProtocol()` and `NdisOpenAdapter()`

Call these hook-free APIs to register and bind a rootkit protocol

- It will be linked in the `ndisProtocolList`, but a firewall will not detect its registration and binding

Unlink returned `NDIS_PROTOCOL_BLOCK` from the list

- We have already found `ndisProtocolList`

Major shortcoming

- Firewall may detect and hook newly registered protocol before we unlink it

➢ **Easy to implement, but very impractical: rootkit may be compromised if a firewall has a timer which is frequent enough**

Bypassing Protocol Walking II – Pullout Update

Modify our copies of `NdisRegisterProtocol()` and `NdisOpenAdapter()` code trees

- Substitute all references to the original `ndisProtocolList` with references to the fake one in the generated code: both APIs will remain coherent
- This may be done on the final step of the code generating – relocations linking
- Fake `ndisProtocolList` may be NULL

➢ **Uses disassembler engine (i.e. not easily portable), requires to hook `Receive*` handlers of all other protocols bound to same adapter in order to block packets designated to our TCP/IP stack – only rootkit protocol should receive them**

Bypassing Protocol Walking III – X_FILTER Hooks

It has been shown that NDIS packet receive managers use `X_FILTER.OpenList` as a head of all open bindings

- Choose random protocol binding to the specific adapter by walking its `X_FILTER.OpenList`
- Make a copy of its `NDIS_OPEN_BLOCK` (accessed via `X_BINDING_INFO.NdisBindingHandle`)
- Patch `Receive*` handlers in the copied open block
- Substitute pointer to the original `NDIS_OPEN_BLOCK` for pointer to the patched copy

➢ **Very stealthy: this approach introduces only one pointer modification to a system**

Bypassing Protocol Walking IV – NDIS API Emulation

Register new `ETH_BINDING_INFO` / `X_BINDING_INFO` manually

- Create correct `NDIS_OPEN_BLOCK` without `NdisOpenAdapter()`
- Properly add new `X_BINDING_INFO` which points to faked `NDIS_OPEN_BLOCK` to the `X_FILTER.OpenList`

Major shortcoming

- Very NDIS version dependent

➢ **Very hard to implement properly; different code for every supported NDIS version**

Bypassing Protocol Walking I, II – Stealth Packet Sends

What about sending packets?

- It's almost trivial: only `NDIS_OPEN_BLOCK` and `NDIS_MINIPORT_BLOCK` are required, and they are not hooked by a firewall

Hook-free `NdisOpenAdapter()` may set `NDIS_OPEN_BLOCK.SendHandler` to

- `ndisMSendX()`
- `ndisMSend()`
- `ndisMWanSend()`
- `ndisMSendSG()`

These APIs may be hooked in the `NDIS.SYS` image

- Use code pullout again – this time without any relocations updates

We didn't register our own protocol – we hooked X_BINDING_INFO of an existing one

- So, its `NDIS_OPEN_BLOCK.SendHandler` and `SendPacketsHandler` may be hooked by a firewall

In order to send packets stealthy, we should find original `ndisMSend*` functions

- By tunneling the firewall with an innocent packet: sooner or later it should be sent via call to one of NDIS packet send functions
- By searching NDIS image for not-exported symbols using XREFs or code signatures analysis
- By temporarily registering and binding a dummy protocol with aid of previously discussed methods to get original NDIS send functions pointers
  - Protocol registration and binding must not be caught by a firewall

Demonstration

CODEDGERS

FireWalk rootkit:
kernel mode FTP server over the rootkit's TCP/IP stack VS popular personal firewalls

Hardening Firewalls

CODEDGERS

A firewall should operate at a more privileged level than a rootkit, otherwise it can always be bypassed

Since in i386 NT they both run in kernel mode, the only solution for firewall vendors is to complicate rootkits' (and their authors') life as much as possible

- Maybe full rewrite of NDIS (with a lot of obfuscations) is a good idea – at least, there will be no symbols ☺

Hardening Firewalls – Monitoring Packet Receives

Find unlinked protocols

- Walk filter databases for each miniport, get a list of `NDIS_OPEN_BLOCK`s bound to them
- Hook all found `NDIS_OPEN_BLOCK`s
- Save `NDIS_PROTOCOL_BLOCK`s associated with each `NDIS_OPEN_BLOCK`
- Walk `ndisProtocolList` and alert user about unlinked protocols

KLISTER by Joanna Rutkowska did similar things to find processes unlinked from `PsActiveProcessHead` list

- It was bypassed too ☺

Hardening Firewalls – Monitoring Packet Sends

Firewall has to take into account that rootkit may not use its `NDIS_OPEN_BLOCK.SendHandler()` or `SendPacketsHandler()` to send packets to the network

Rootkit may call `ndisMSend*` directly

- However, it should find these functions first

- **Firewall should at least hook code of `ndisMSend*` and `ndisMWanSend*`. The nature of packet send interface does not require any special system object registration (you should register and bind a protocol in order to receive packets), and until this behavior doesn't change, firewalls will be having hard times catching sent packets**

References

Joanna Rutkowska, *KLISTER*
http://invisiblethings.org/tools/klister-0.4.zip

Joanna Rutkowska, *Rootkits vs. Stealth by Design Malware*
http://invisiblethings.org/papers/rutkowska_bhfederal2006.ppt

Greg Hoglund, *NT Rootkit*
http://www.rootkit.com/vault/hoglund/rk_044.zip

Opc0de, *Bypassing VICE 2*
http://rootkit.com/newsread.php?newsid=197

PCAUSA, *Windows Network Data and Packet Filtering*
http://www.ndis.com/papers/winpktfilter.htm

90210, *Bypassing Klister 0.4 With No Hooks or Running a Controlled Thread Scheduler*
http://www.rootkit.com/vault/90210/phide2.zip

Questions?

Thank you for your time!