

Explicit Compromise Detection

Joanna Rutkowska

`http://invisiblethings.org`

Redmond, WA, December 4th, 2005.

Problem definition

- ⊕ System is compromised when it looks innocent but in fact is *not clean*.
- ⊕ What does it mean to be *not clean*?
 - ⊕ Backdoors, keyloggers, etc..
 - ⊕ Spyware installed by user who doesn't read EULA carefully?
- ⊕ In general: system is compromised when there are *things* installed (running) which cannot be easily spotted using standard administration/AV tools.

Different approaches to Compromise Detection...

- ⊕ Cross view based approaches
 - ⊕ Detect hidden files, registry keys, processes
- ⊕ Check Integrity of Important OS elements
 - ⊕ System files integrity (verify digital signatures, AV heuristics, etc...)
 - ⊕ Enumerate autorun programs (lots of tools)
 - ⊕ Code sections integrity (SVV)
 - ⊕ IAT/EAT, SDT, IRP tables (VICE2 checks some of them)
 - ⊕ NDIS pointers
 - ⊕ ...
- ⊕ Signature based approaches
 - ⊕ Scan for *known* rootkit/backdoor engines

X-VIEW based approaches

- ⊕ Detect *hidden* objects (files, reg keys, processes)
- ⊕ Assumes that there actually are hidden objects
- ⊕ Also doesn't tell how the system was compromised (e.g. what technique was used to hide object, just that we had hidden object)
- ⊕ Easy bypass algorithm:
 - ⊕ Detect that someone performs object scan
 - ⊕ If yes, temporarily unhide our hidden object
 - ⊕ Will not work if X-VIEW scanner performs also other heuristics on found objects (AV-like scanner)
- ⊕ But, we can have compromised system not having any hidden objects!

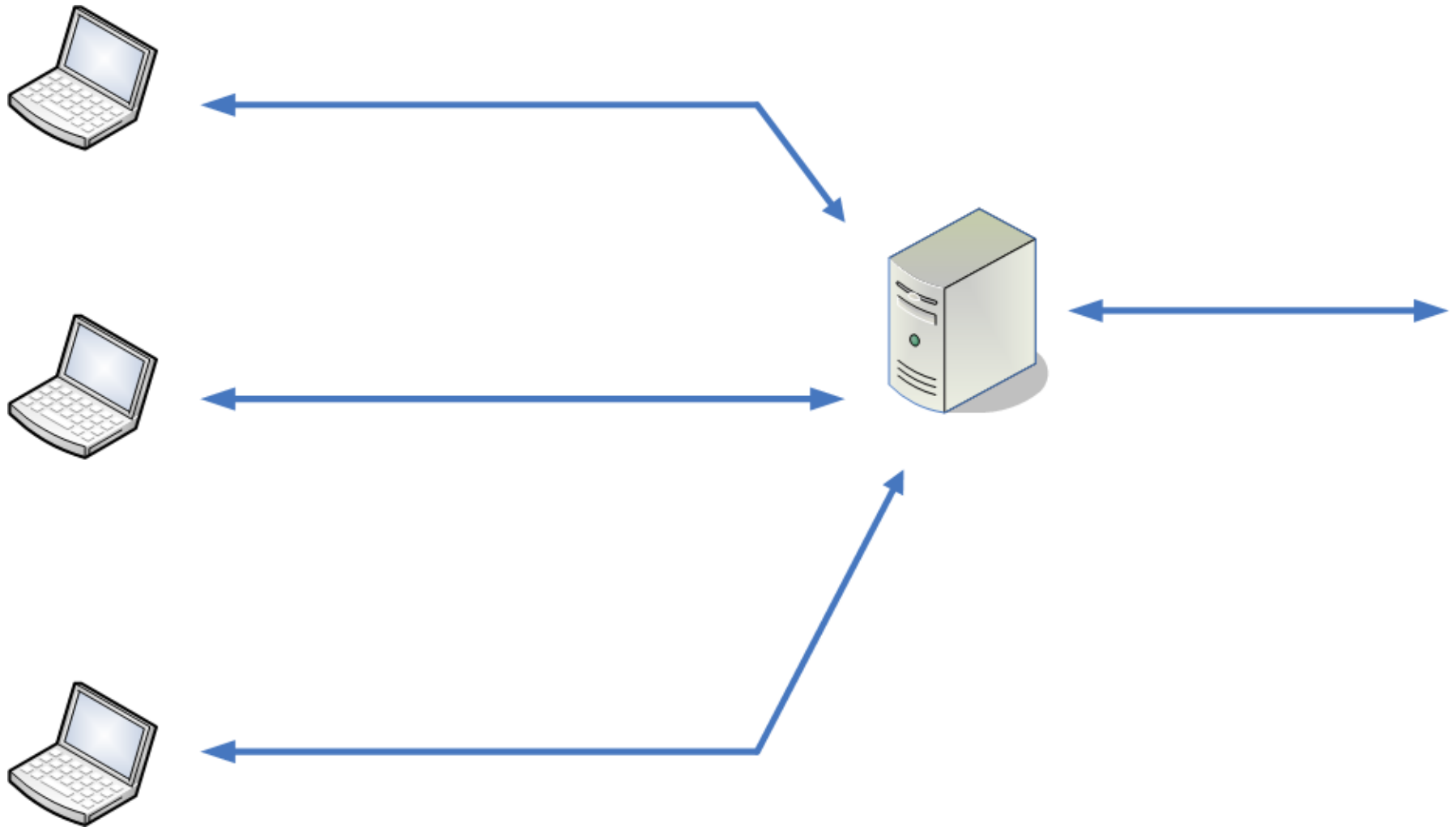
Stealth by design malware

- ⊕ Malware can be written without usage of extra processes → no need for process hiding
- ⊕ Malware can use covert channels for communication → no need for socket hiding
- ⊕ Malware can use advanced file infection for surviving the reboot → no need for registry key hiding (watch out for system files digital signatures)
- ⊕ Malware can use advanced metamorphic engines → no need for files hiding
- ⊕ Many tasks (e.g. keylogging, network backdoor) can be implemented without code modifications → no need for implementation specific attacks (ala commercial hxdef)

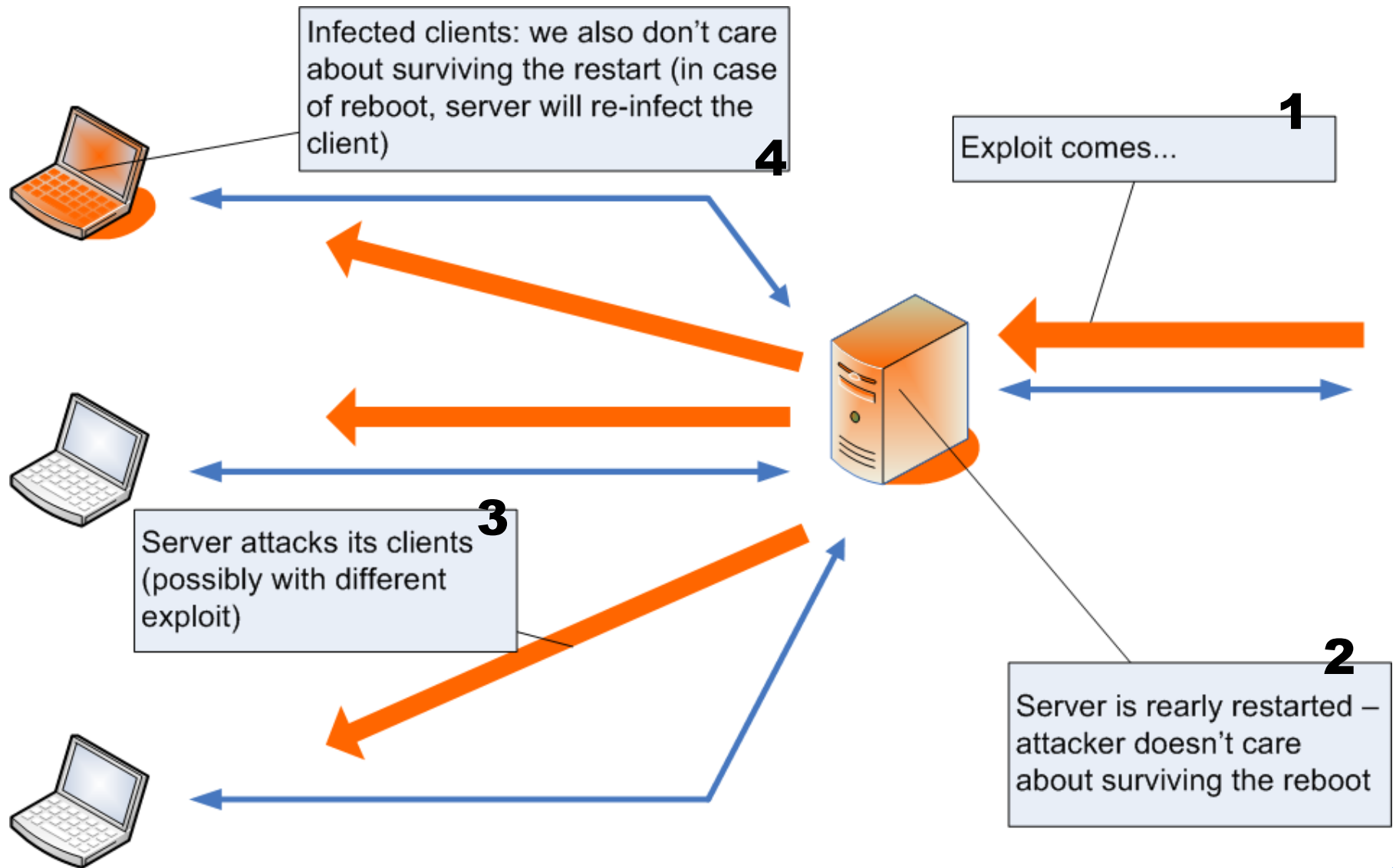
Stealth by design approach

- ⊕ Currently the biggest problem in creating “stealth by design” malware is how to survive system reboot (bypass detailed offline analysis AKA forensic analysis)
- ⊕ Despite using advanced file infection techniques (ala Z0mbie's *Zmist*), malware can... decide that it will not be interested in surviving system reboot...

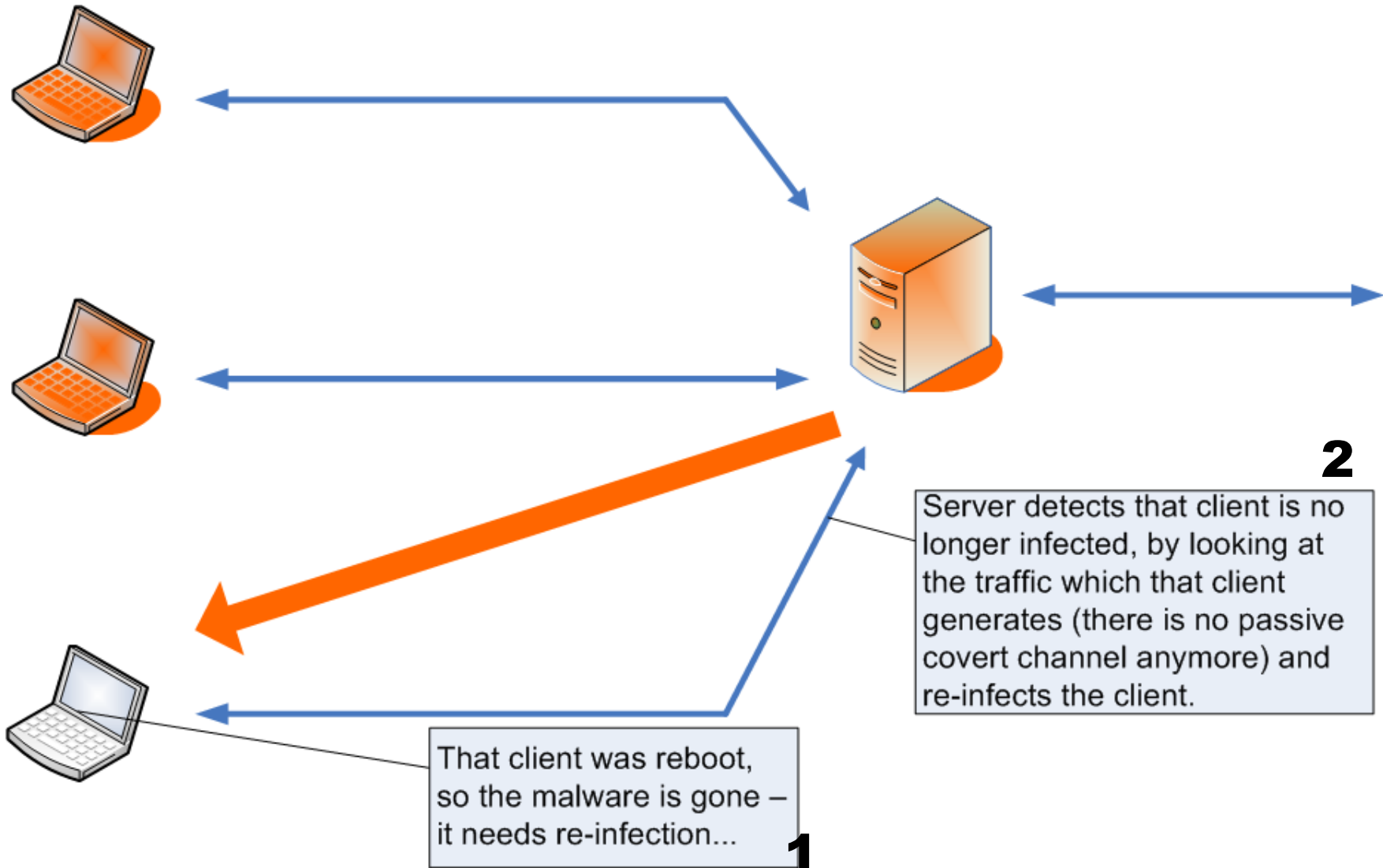
Example



Network infected



Client re-infection



Passive covert channels

- ⊕ My talk at CCC 2004:
 - ⊕ <http://invisiblethings.org/papers/passive-covert-channels-linux.pdf>
 - ⊕ http://invisiblethings.org/papers/joanna-passive_covert_channels-CCC04.ppt
- ⊕ NUSHU (passive covert channel POC in TCP ISNs for Linux 2.4 kernels):
 - ⊕ <http://invisiblethings.org/tools/nushu.tar.gz>
- ⊕ How to detect NUSHU (and how to improve it so it will not be detectable) by Steven Murdoch et al:
 - ⊕ <http://www.cl.cam.ac.uk/users/sjm217/papers/ih05coverttcp.pdf>

Explicit Compromise Detection (ECD)

- ⊕ Verify important OS components (like code sections in memory, SSDT, IDT, IRP dispatch tables, etc...)
- ⊕ We don't try to find malware side effects (hidden objects) but malware itself – possible to detect “stealth by design” malware as well.

Implementing ECD...

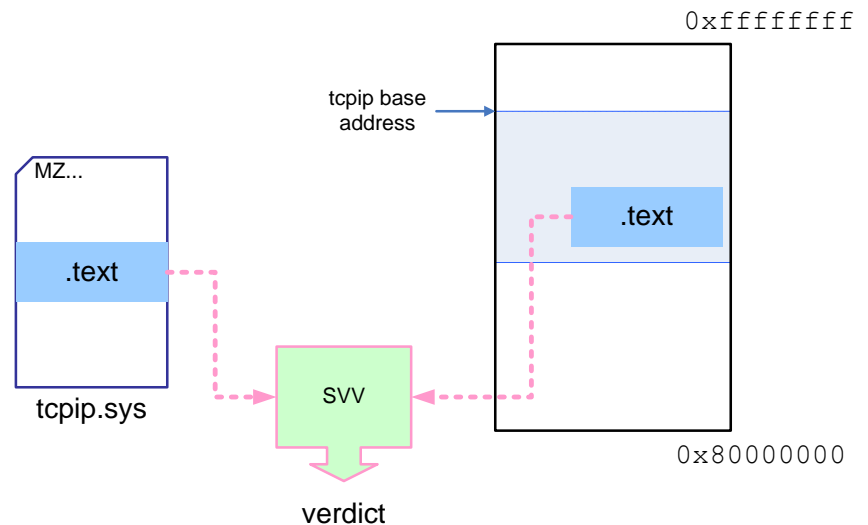
- ⊕ Step 0: verify file system and registry integrity
 - ⊕ Digital signatures of all system files (`sigverif`, be aware to use proper certificate)
 - ⊕ AV heuristics for other file infection detection (EPO viruses)
 - ⊕ Auto-starting programs enumeration
 - ⊕ Off-line analysis

Implementing ECD...

- ⊕ Step 1: in-memory code sections integrity
- ⊕ Step 2: important dispatch tables integrity
 - ⊕ IAT/EAT, SDT, IDT, IRP dispatch tables
- ⊕ Step 3: network subsystem integrity
 - ⊕ NDIS
 - ⊕ Winsockets
- ⊕ Step 4 ...

System Virginty Verifier Idea

- ⊕ Code sections are read-only in all modern OSes
- ⊕ Program *should not* modify their code!
- ⊕ Idea: check if code sections of important system DLLs and system drivers (kernel modules) are the same in memory and in the corresponding PE files on disk
 - ⊕ Don't forget about relocations!
 - ⊕ Skip .idata
 - ⊕ etc...



False Positives

- ⊕ Important in commercial implementations (AV products)
- ⊕ “Rootkit technology” is widely used in many innocent tools ;)
 - ⊕ Personal Firewalls/Behavior blocking systems
 - ⊕ Monitoring tools like Virus Monitors
 - ⊕ System “tracing” tools (like Filemon, DbgView, etc...)
 - ⊕ ...
- ⊕ How to deal with those false positives?
- ⊕ But first... the self –modifying kernel problem...

Self-modifying kernel code

- ⊕ `.text` section of the two core kernel modules does not match the corresponding files on the disk!
- ⊕ Reason: code self modifications to replace some hardware related functions so they match the actual hardware
- ⊕ Those two components are:
 - ⊕ `ntoskrnl.exe` ← kernel core
 - ⊕ `HAL.DLL` ← Hardware access functions
- ⊕ How to deal with those discrepancies?


Filtering kernel self-modifications

- ⊕ Observation: there are only a bunch of functions which are altered (10-20 in `ntoskrnl.exe` and `hal.dll`).
- ⊕ In HAL: all modifications are `00` → `XX` (i.e. original function body is NULL in the file)
- ⊕ In NTOSKRNL:
 - ⊕ Most are one-byte modifications (`NOP` → `XX`)
 - ⊕ The rest (5 or so) are pretty stable among different kernels (from 2000, through XP, to 2003)

False Positives

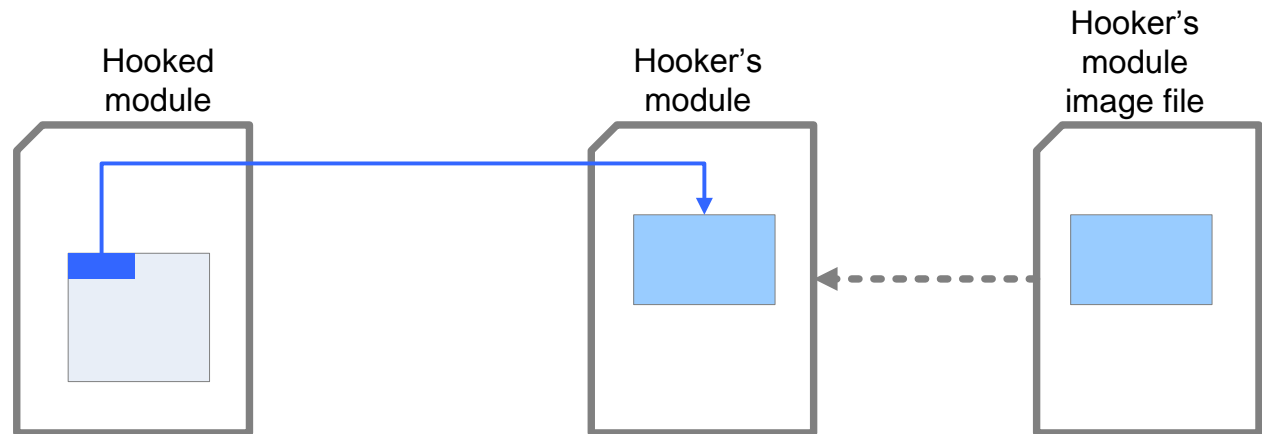
- ⊕ Now how to deal with “innocent” hooking?
- ⊕ To solve this problem we should describe the several types of hooking which we can observe in the nature...

Missing vs. hidden files

- ⊕ Some definitions first:
- ⊕ File is *missing* when `CreateFile()` fails,
- ⊕ File or directory is *hidden* when `FindFirstFile()`/
`FindNextFile()` do not return that file in directory listing,
- ⊕ If a directory is *hidden* all its files and subdirectories are also *hidden*.
- ⊕ Note:
 - ⊕ missing files don't need to be hidden (e.g. `REGSYS.SYS`)
 - ⊕ hidden files usually are *not* missing
(e.g. `c:\root3d\hxdef.sys`)
 Dir is hidden, but present

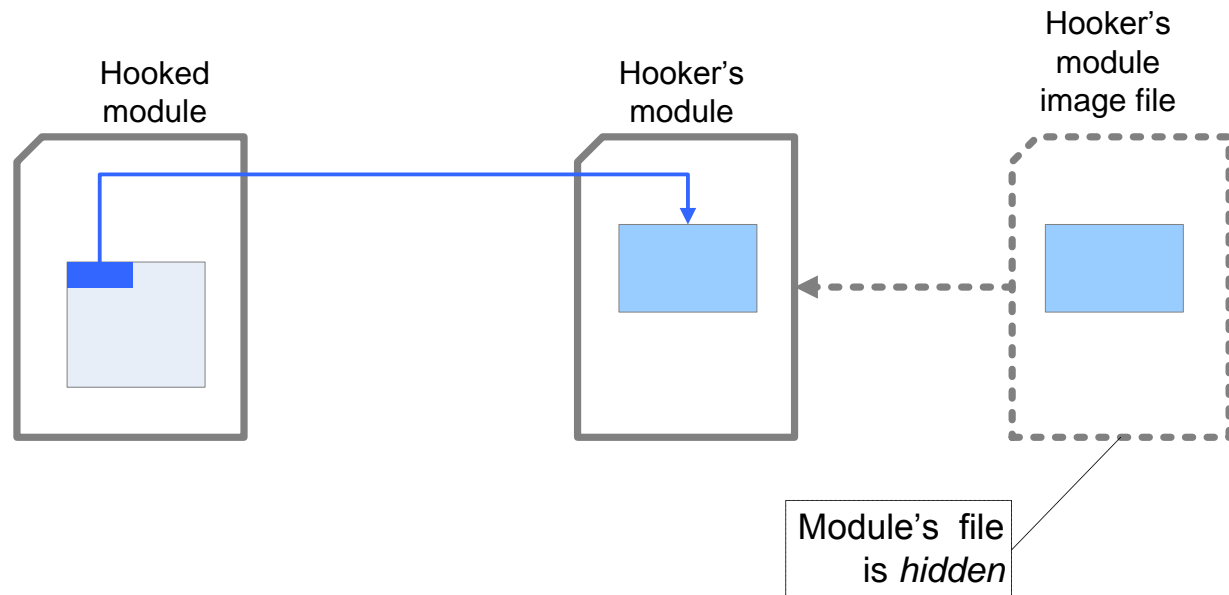
Typical Innocent hooking

- ⊕ New code looks like a “JMP” (not obfuscated)
- ⊕ Redirection to code section inside valid module
- ⊕ Hooker's module file present and not hidden (rootkit/backdoor needs to hide its module)
- ⊕ Verdict: probably personal firewall or other innocent program.



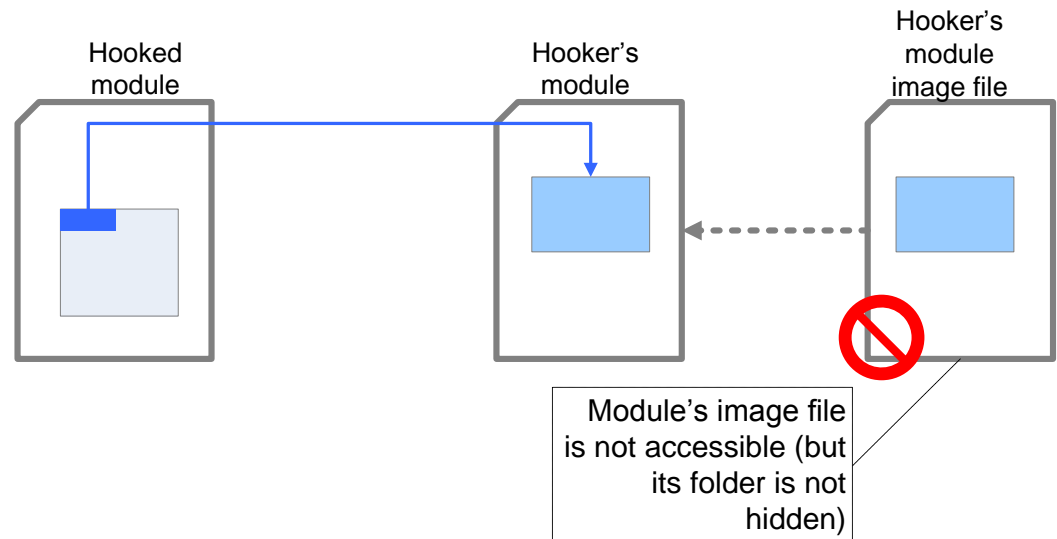
Malicious hooking I

- ✚ Hooker's module file is *hidden*. It may be open-able by *CreateFile()* but still it is probably a rootkit. Innocent programs do not hide their module files.



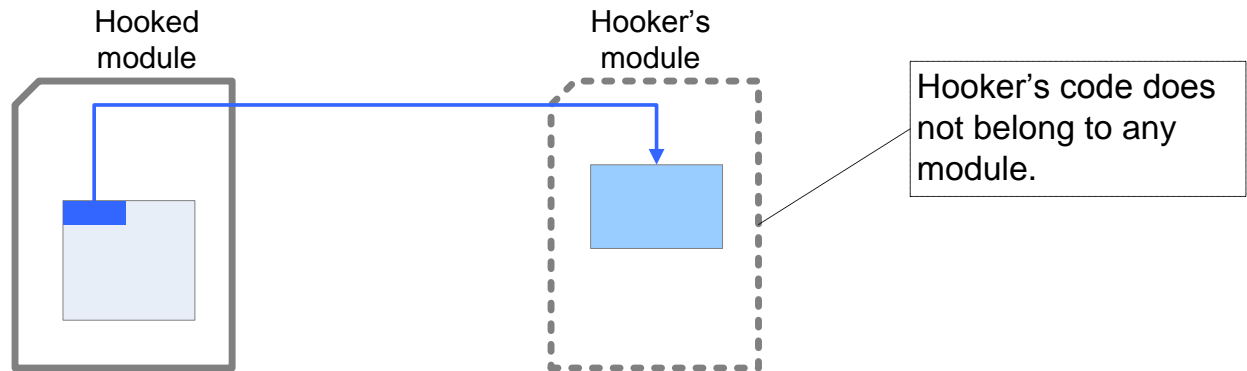
“Debugger” – like hooking

- ⊕ Module image file is missing (but is not located in a hidden directory)
- ⊕ Typical for tools which extract their driver from resource section and place it *temporarily* in the system driver directory (DbgView, RegMon, etc...)



Malicious hooking II

- ⊕ Redirection to code which does not belong to any module.
- ⊕ Typically this means that either:
 - ⊕ Code was placed in a manually allocated block of memory
 - ⊕ Owning module's descriptor was intentionally tampered
- ⊕ This is very suspicious. Innocent program should not do this.



Malicious hooking III

- ⊕ Rootkit uses obfuscated code to redirect execution → SVV cannot determine the target address.
- ⊕ It's not a problem, because we expect such behavior (i.e. using obfuscated JMP-ing code) in malicious software only
- ⊕ So, if altered code cannot be decoded into a JMP then report it as an infection.

Detecting SDT modifications

- ⊕ `KiServiceTable` is part of `.text`
- ⊕ If modifications in inside `KiServiceTable` then it is the target address itself (not JMP-ing code)
- ⊕ The actual SDT can be relocated (like when Kaspersky AV is used)
- ⊕ Thus `KiServiceTable` is not always pointed by `[KeServiceDescriptorTbl[0]+0]`
- ⊕ Thus a robust method for finding the address of `KiServiceTable` is needed – see 90210's article on rootkit.com for the smart solution

SVV Verdicts



100% virgin (not expected to occur in the wild)



Seems ok



Innocent hooking detected



“Debugger” like hooking detected



Very suspected but may be a false positive



Compromised!

SVV 1.x Release

- ⊕ SVV 1.x checks only `.text` sections
- ⊕ Should catch all code-alternating rootkits
- ⊕ Some backdoors (like EEYE BootRoot) can modify other code sections as well (like `PAGE*` inside `NDIS.SYS`)
- ✗ Checking of other sections will be added soon
- ✗ Big Implementation Problem: how to reliably scan kernel memory (not causing BSOD)? `MmProbeAndLockPages()` seems to be not enough! SVV 1.3 uses heuristics before calling `MmProbe...()`.

SVV future

- ⊕ SVV 1.x takes care only about CODE VIRGINITY
- ⊕ It is only the very first step in building integrity-based rootkit detector
- ⊕ Next versions should include:
 - ⊕ EAT/IAT, SSDT, IDT and IRP tables verification
 - ⊕ NDIS important data pointers verification
 - ⊕ More ...
- ⊕ So, we see the need to **define all the vital OS components** which should be verified...
- ⊕ ... this is what OMCD project (see later) is about.

Related work (integrity checkers)

- ⊕ VICE by Jamie Butler
 - ⊕ Tries to detect hook, not code modifications
 - ⊕ Relatively easy to bypass when rootkit uses non standard (polymorphic) JMP-ing code
 - ⊕ Checks also EAT, SDT and IRP (good!)
 - ⊕ Uses API to read other processes memory (bad!)

Related work (integrity checkers)

- ⊕ SDT restore by SK Chong
 - ⊕ SDT integrity checking
 - ⊕ Relocated SDT detection (good!)
 - ⊕ BTW, would be nice if it also checked IDT table

Related work (integrity checkers)

- ⊕ IceSword by pjf USTC
 - ⊕ SSDT
 - ⊕ Win32 Message hooks
 - ⊕ ASEPS (Some auto-starting programs, BHO, SPI)
 - ⊕ Hidden Processes and Services enumeration (hidden not marked – not like in X-DIFF)
 - ⊕ File system and registry explorer (including *hidden objects*)
- ⊕ It is only enumeration tool – does not give any verdict if the system is compromised or not.

System Integrity Verifiers

	SVV 1.0	SDT Restore 0.2	VICE 2.0	IceSword 1.12	Kernel Debugger (+ scripts)
Code verification	YES		Only “JMP” detection		YES
SDT		YES	YES	YES	YES
IDT				?	YES
IAT/EAT			YES (lots of false positives)		YES
IRP dispatch Tables			YES		YES

OMCD

- ⊕ New project just started by the author with the help of the community (hopefully)
- ⊕ Currently focuses only on Windows systems
- ⊕ Hosted at ISECOM:
`http://www.isecom.org/omcd/`

OMCD goals

- ⊕ Create list of all important OS elements which should be verified to find system compromise
- ⊕ Not only a list but also a road map (methodology) of how those parts should be verified
- ⊕ Intended mainly for developers of compromise detection tools
- ⊕ Could be useful as a reference for comparing different tools on the market (“*our tool implements OMCD sections A, B, C & D!*”)
- ⊕ Author *believes* that there are only a **finite number of ways that the rootkits and network backdoors can be implemented** in the OS (see the slide about custom network backdoors and implementation specific attacks later)...

OMCD outline

- A. Preliminaries
- B. File system verification
- C. Registry Verification
- D. Application Integrity Verification
- E. Usermode-level memory verification**
- F. Kernel-mode level memory verification**
- G. Network activity analysis

Section E (Usermode-level memory) overview

- ⊕ Discovering suspected (not hidden) processes and threads
- ⊕ Code sections verification
- ⊕ IAT/EAT tables verification
- ⊕ Considerations for memory access techniques
 - ⊕ Using API for accessing other processes memory
 - ⊕ Direct virtual memory access (DVMA)
- ⊕ ??? [your suggestions here]

Section F (kernel memory) overview

- ⊕ Considerations for memory access technique
- ⊕ Code sections verification
- ⊕ Dispatch tables and function pointers verification
 - ⊕ Service Dispatch Table (SDT) verification
 - ⊕ Interrupt Dispatch Table (IDT) verification
 - ⊕ IRP dispatch tables hook discovery
 - ⊕ NDIS function pointers hook discovery (Network DKOM rootkits)
 - ⊕ Configuration Manager function pointers hook discovery (Registry DKOM rootkits)
- ⊕ System Registers verification
 - ⊕ IA-32 processors (SYSENTER, Debug registers)
 - ⊕ Other processors?
- ⊕ Discovering suspected attached devices (filer based rootkits)
- ⊕ Discovering suspected kernel modules
- ⊕ Discovering hidden threads and processes

OMCD example: Shadow Walker detection

- ⊕ SW modifies IDT (INT 0xE hooking) → OMCD.F (IDT Verification)
- ⊕ SW can modify Page Fault Handler code instead (Inline code hook) → OMCD.F (Code verification)
- ⊕ SW can use Debug registers to hide IDT or Page fault handler modifications → OMCD.F (CPU Registers Verification)

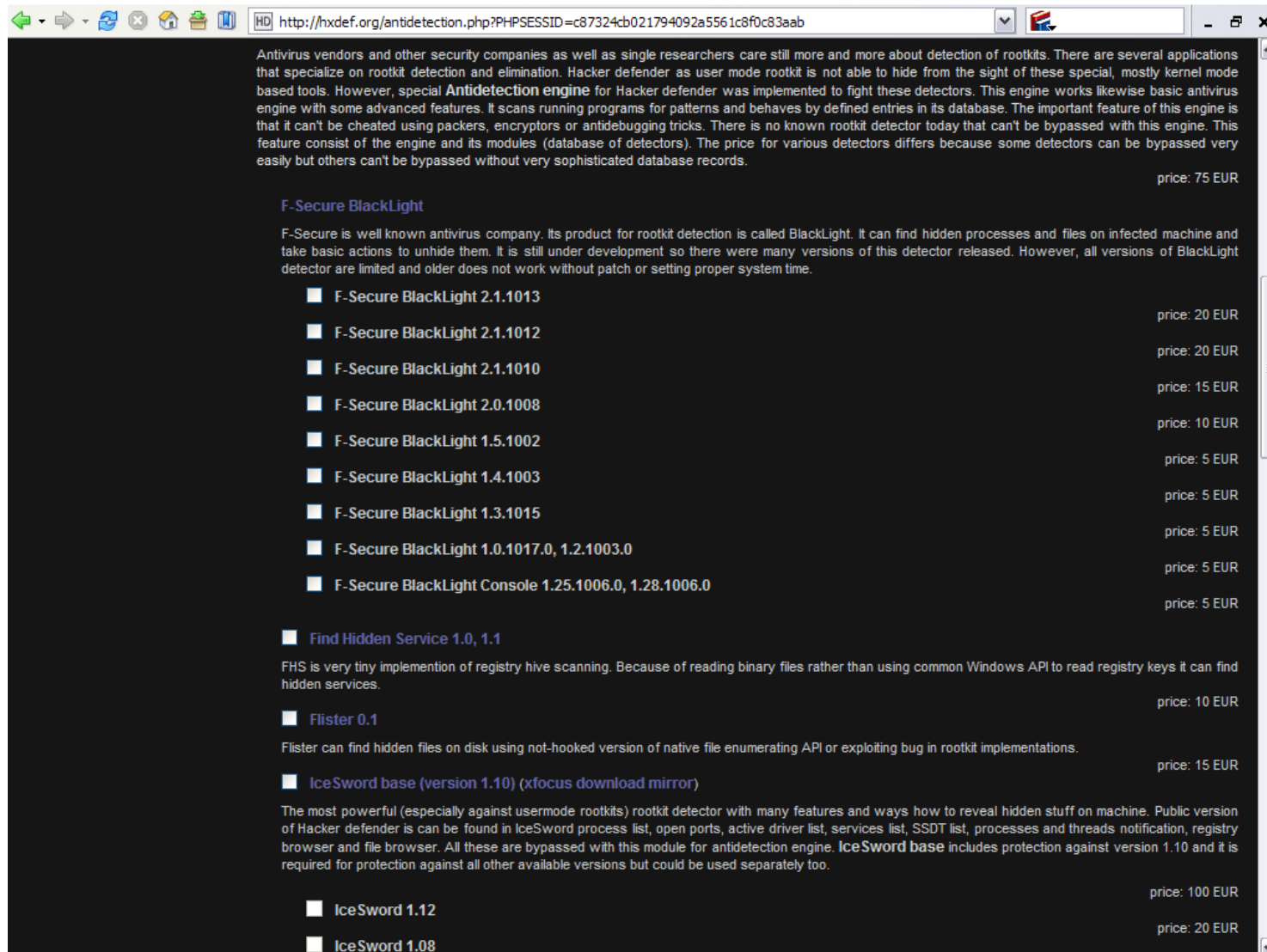
OMCD development

1. Release some early OMCD version (lots of gaps)
 2. Feedback from the community how rookits/backdoors can be implemented so that they are not detected by the latest OMCD version
 3. Update OMCD and goto #2
- ⊕ Thesis: The above algorithm is not infinite :)
 - ⊕ We are leaving less and less space for the malware to survive...
 - ⊕ If OMCD worked then in the future we would be observing only *implementation specific attacks*.

Implementation specific attacks

- ⊕ Malware author decides to cheat *particular* detector
- ⊕ It can for example:
 - ⊕ Hook IRP communication between detector and its kernel agent.
 - ⊕ “Exploit” design bug in the detector
 - ⊕ Cheat by hooking UI functions! (Win32 message hooking)
 - ⊕ Detect particular process (by signature scanning) and replace it with own version – which looks the same but reports clear system
 - ⊕ etc...

Implementation specific attacks



Antivirus vendors and other security companies as well as single researchers care still more and more about detection of rootkits. There are several applications that specialize on rootkit detection and elimination. Hacker defender as user mode rootkit is not able to hide from the sight of these special, mostly kernel mode based tools. However, special **Antidetection engine** for Hacker defender was implemented to fight these detectors. This engine works likewise basic antivirus engine with some advanced features. It scans running programs for patterns and behaves by defined entries in its database. The important feature of this engine is that it can't be cheated using packers, encryptors or antidebugging tricks. There is no known rootkit detector today that can't be bypassed with this engine. This feature consist of the engine and its modules (database of detectors). The price for various detectors differs because some detectors can be bypassed very easily but others can't be bypassed without very sophisticated database records.

price: 75 EUR

F-Secure BlackLight

F-Secure is well known antivirus company. Its product for rootkit detection is called BlackLight. It can find hidden processes and files on infected machine and take basic actions to unhide them. It is still under development so there were many versions of this detector released. However, all versions of BlackLight detector are limited and older does not work without patch or setting proper system time.

- F-Secure BlackLight 2.1.1013 price: 20 EUR
- F-Secure BlackLight 2.1.1012 price: 20 EUR
- F-Secure BlackLight 2.1.1010 price: 15 EUR
- F-Secure BlackLight 2.0.1008 price: 10 EUR
- F-Secure BlackLight 1.5.1002 price: 5 EUR
- F-Secure BlackLight 1.4.1003 price: 5 EUR
- F-Secure BlackLight 1.3.1015 price: 5 EUR
- F-Secure BlackLight 1.0.1017.0, 1.2.1003.0 price: 5 EUR
- F-Secure BlackLight Console 1.25.1006.0, 1.28.1006.0 price: 5 EUR

Find Hidden Service 1.0, 1.1

FHS is very tiny implementation of registry hive scanning. Because of reading binary files rather than using common Windows API to read registry keys it can find hidden services.

price: 10 EUR

Flister 0.1

Flister can find hidden files on disk using not-hooked version of native file enumerating API or exploiting bug in rootkit implementations.

price: 15 EUR

IceSword base (version 1.10) (xfocus download mirror)

The most powerful (especially against usermode rootkits) rootkit detector with many features and ways how to reveal hidden stuff on machine. Public version of Hacker defender is can be found in IceSword process list, open ports, active driver list, services list, SSDT list, processes and threads notification, registry browser and file browser. All these are bypassed with this module for antidetection engine. **IceSword base** includes protection against version 1.10 and it is required for protection against all other available versions but could be used separately too.

price: 100 EUR

- IceSword 1.12 price: 20 EUR
- IceSword 1.08

Implementation specific attacks

- ⊕ Let's say it aloud: they are ugly and stupid!
- ⊕ They're not advancing the state of the art in rootkit/malware research – they're extremely boring.
- ⊕ Such attacks are *always* possible against a *particular* program (even if the detector has no bugs) – so don't confuse with bugs exploitation (Buffer Overflows and so)

Implementation Specific attacks: possible solutions

- ⊕ The more (various) detectors exist on the market the less profitable such attacks are for the malware authors
 - ⊕ Now the attacker starts acting just like his fellows from AV companies ;)
 - ⊕ OMCD aims to stimulate more various tools to be created
- ⊕ For commercial tools: make use of the update feature to constantly introduce small changes into detector (communication interface, UI, exec signatures, etc...). This could be automated, but the program for doing this should be kept private by the company.
- ⊕ Offer paid, custom, rootkit detection service for companies.

Custom Network Backdoors

- ⊕ Another form of implementation specific attack, but not against detector, but rather against application which is exploited.
- ⊕ Attacker needs to find function pointers which are used for calling send/recv packet handlers in the target application (e.g. a web server).
- ⊕ This is definitely not a generic way for creating a backdoor...
- ⊕ ...but programs written in C++ should be easy targets for such attacks (VPTRs). Should we add VPTRs verification to OMCD?

Detection vs. Prevention

- ⊕ Real world examples show that it was possible to break most of the “unbreakable” prevention systems.
- ⊕ Claiming the we don't have bugs in kernel is also kind of “prevention system” approach.
- ⊕ If we rely only on prevention, then we cannot tolerate even a single bug. We're blind, we trust that our prevention system was always accurate.
- ⊕ Prevention system cannot be upgraded – if we realize that it needs to be fixed, then we cannot be sure anymore that it hadn't been compromised before...

Acknowledgements

- ⊕ Dave Aitel (for discussing custom network backdoors idea)
- ⊕ Greg Wroblewski / Microsoft (for interesting discussions about kernel self modifications)
- ⊕ Sherri Sparks & Jamie Butler (for exploring new ideas for rootkit creation)