Fighting Stealth Malware – Towards Verifiable OSes

Joanna Rutkowska

Advanced Malware Labs

COSEINC

23rd **C**haos **C**ommunication **C**ongress

Berlin, Germany, December 28th, 2006

Stealth Malware

- rootkits, backdoors, keyloggers, etc…
- ***stealth*** is a key feature!
  - *stealth* – means that legal processes can't see it (A/V)
  - *stealth* – means that administrator can't see it (admin tools)
- ***stealth* – means that we should never know whether we're infected or not!**

Paradox…

- If a stealth malware does its job well…
- …then we can not detect it…
- …so how can we know that we are infected?

How we know that we were infected?

- We count on the bug in the malware! We hope that the author forgot about something!
- We use *hacks* to detect some known stealth malware (e.g. hidden processes).

- We need to change this!
- We need a systematic way to check for a system integrity!
- We need a solution which would allow us to detect malware which is not buggy!

The 4 Myths About Stealth Malware Fighting!

Myth #1

- **Remove the disk and scan for anomalies on a trusted machine!**
  - Alternative version: boot system from the clean CD

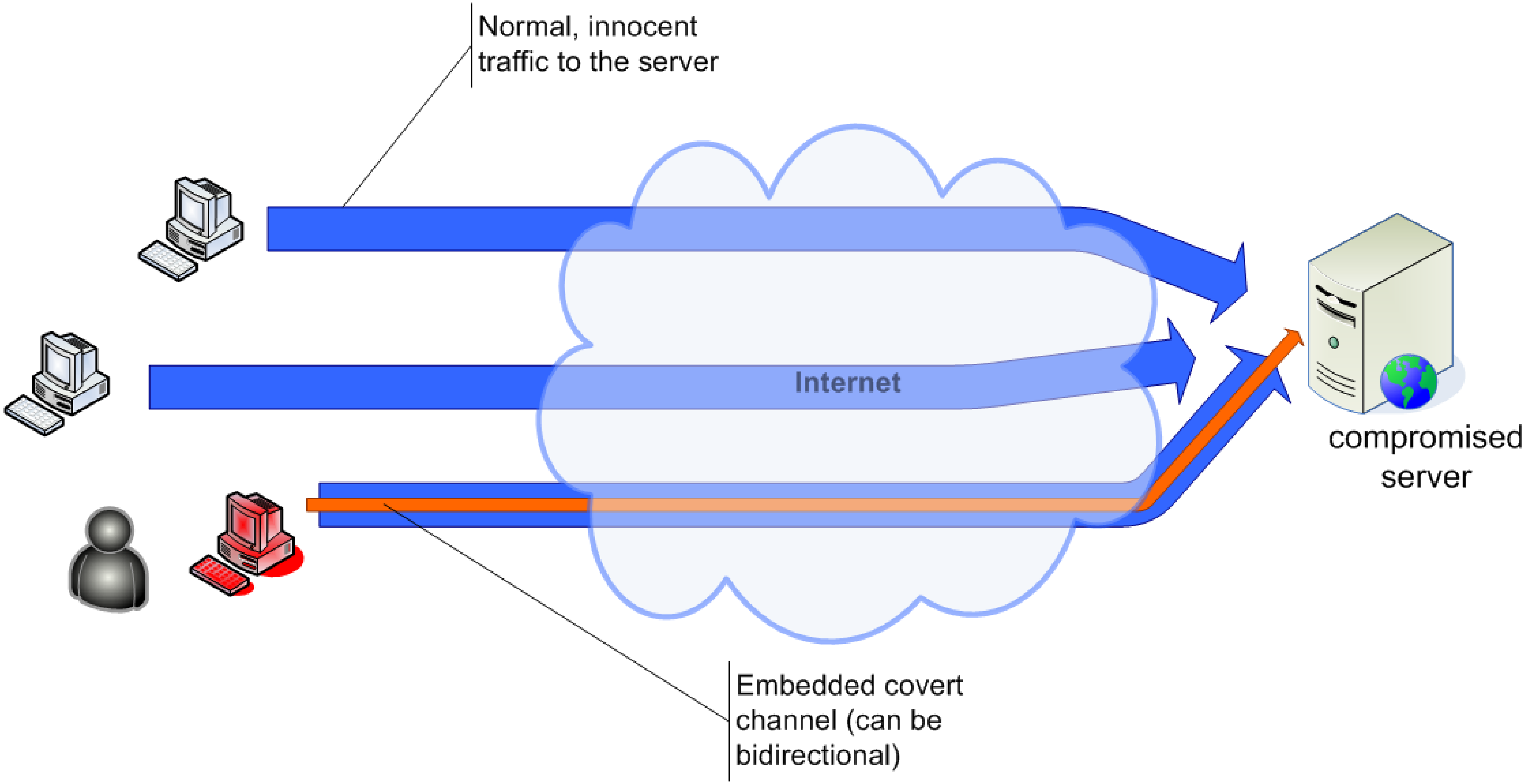
- This will not work against non-persistent rootkits!
- This will also not work against persistent rootkits, which do not rely on disk to survive reboot
  - BIOS-resident rootkits
  - PCI-ROM resident rootkits
  - Ask John Heasman for details ;)

Myth #2

- **Detect malware by analyzing network activity!**
  - After all, every piece of malware needs to “call home” or allow for some other form of communication with the attacker… We will catch it then, right?

- Malware may use advanced forms of covert channels, making its network-based detection virtually impossible in practice…

Covert channels



Simple data hiding in HTTP

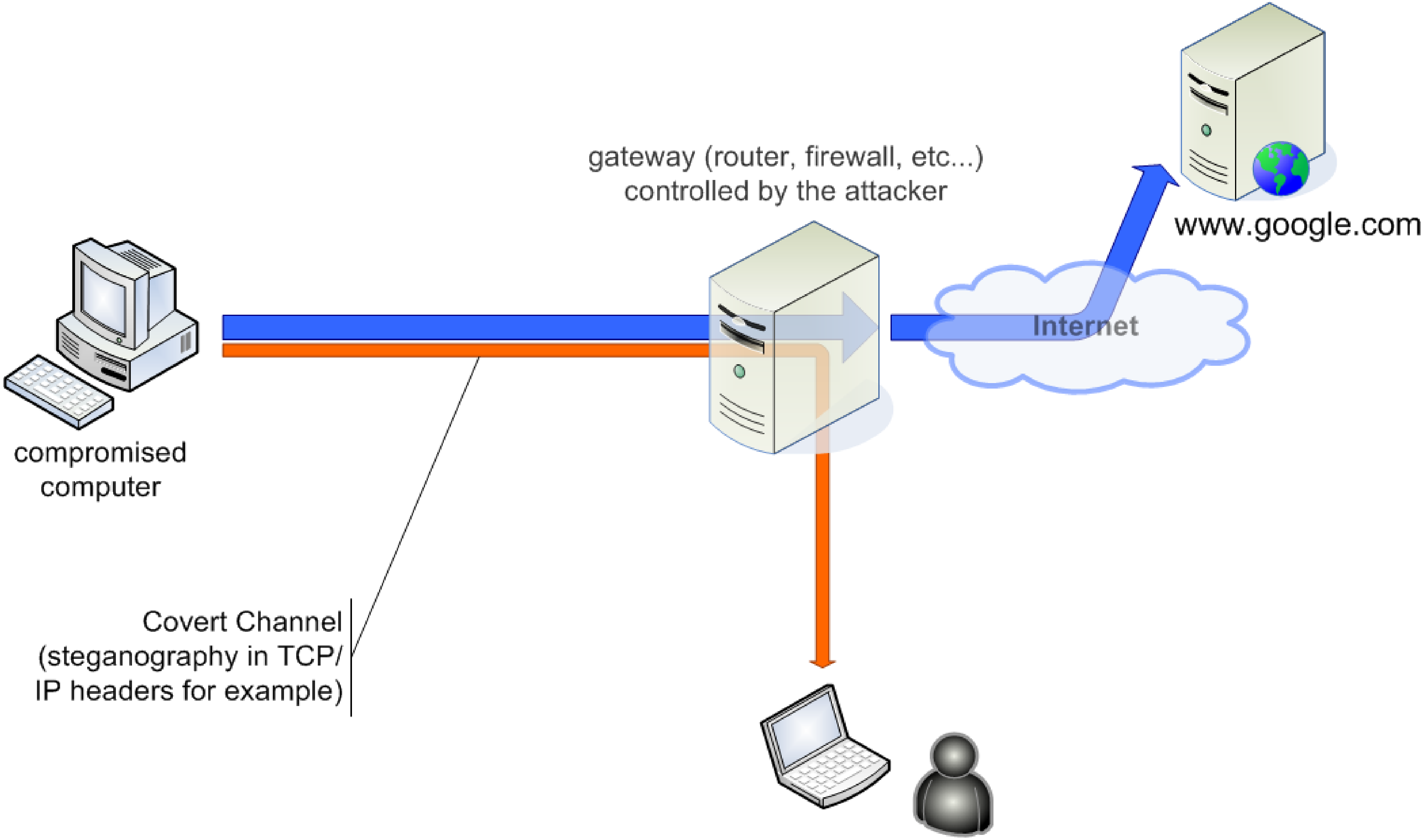
```
GET http://www.somehost.com/cgi-bin/board.cgi?view=12121212
/ HTTP/1.0
Host: www.somehost.com
User-Agent: Mozilla/5.0 (12121212)
Accept: text/html
Accept-Language: en,fr,en,fr,en,en,en,en
Accept-Encoding: gzip,deflate,compress
Accept-Charset: ISO-8859-1,utf-8,ISO-1212-1
CONNECTION: close
Proxy-Connection: close
X-Microsoft-Plugin: unexpected error #12121212
```

source: `http://gray-world.net`

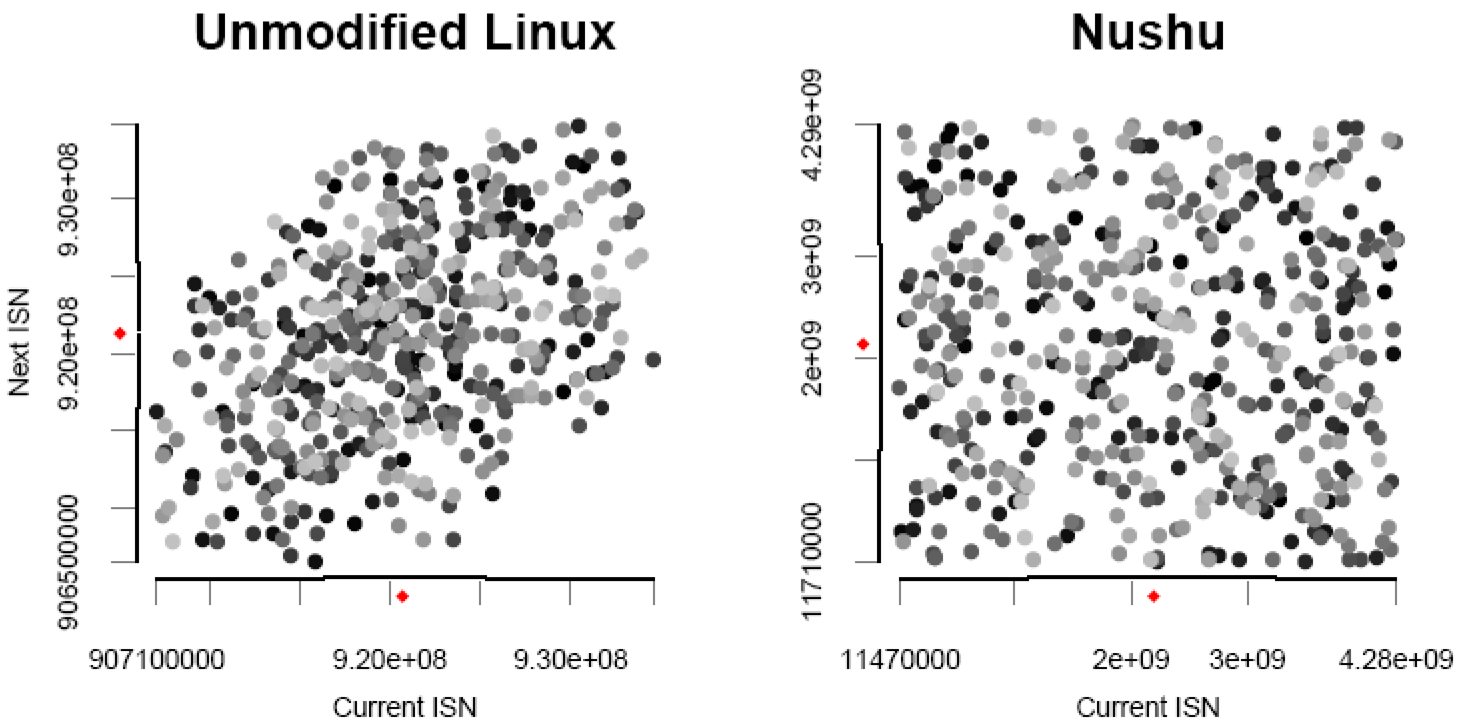
NUSHU proof of concept

- Presented at 21st CCC in 2004 by yours truly
- Passive – do not generate any packets, just changes some fields (TCP ISN) in the packets generated by a user
- Uses strong encryption to make the new ISN’s look random
- Implements error recovery protocol
- Exemplary implementation for Linux 2.4 kernel

Passive Covert Channels



Statistical detection



Source: Steven J. Murdoch and Stephen Lewis, Computer Laboratory University of Cambridge, 22nd CCC, 2005.

Improving NUSHU

- Not enough randomness is bad, but too much of randomness is not good either!
- We need to mimic the original OS ISN generator as closely as possible
- Murdoch and Lewis analyzed the Linux and BSD kernel generators in detail and proposed how to improve NUSHU so that it won’t be detectable by such easy statistical analysis
- Does that mean the improved scheme is totally undetectable?
  - In theory: probably not…
  - In practice: for sure!

Myth #3

- **Find malware by looking for hidden objects!**
  - **Hidden processes, threads, files, reg keys, etc.**

- Malware may be “Stealth by Design” and does not create any objects
  - See e.g. my deepdoor proof-of-concept presented at BH Federal 2006.

Myth #4

- **Deploy efficient protection technology and don't worry about rootkits anymore!**

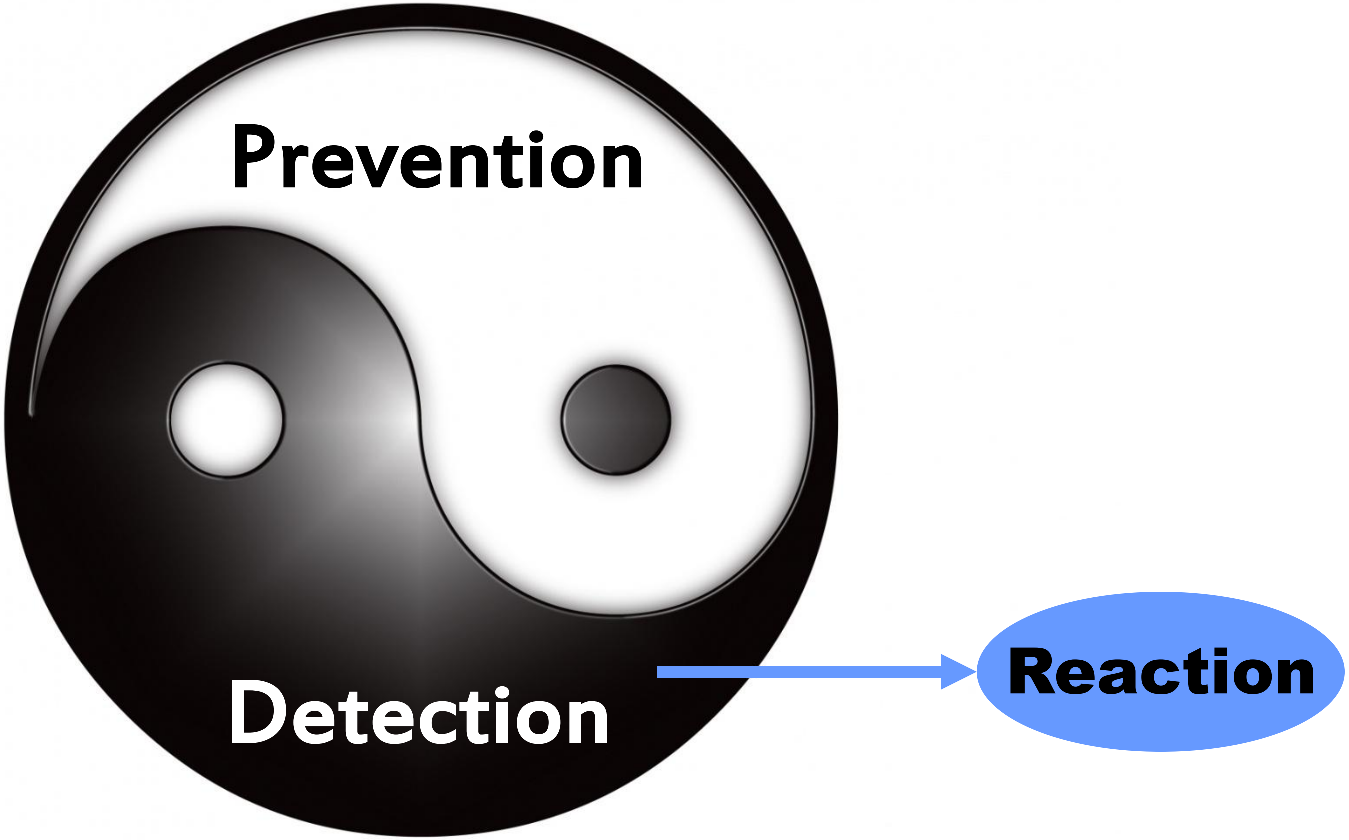
- Do you know a prevention technology which has never been bypassed (having a clear vulnerability record)? Anyone?
- Also – an attacker can always exploit a bug (e.g. in kernel graphics card driver)

Kernel Protection bypasses

- Linux + grsecurity: /dev/kmem protection bypass, Guillaume Pelat, 2002.
- SELinux local priv esacaltion, Rafal Wojtczuk, 2003.
- OpenBSD securelevel bypass, Loïc Duflot at el., 2006.
- Vista kernel drivers signature check bypass, Joanna Rutkowska, 2006.
- [+ all overflows in kernel drivers]

- Still believe that we can come up with a 100% kernel protection? ;)

Yin & Yang of Security



Detection

- In my work I focus on *detection*
- And I believe that the proper approach to detection is a *Systematic Verification* of running OS & Applications
- In contrast to “chaotic detection” as we see today, which is based on implementing various “hacks” against known rootkiting techniques…
  - Yes, I also created several “chaotic detectors” in the past ;)

Detection vs. Prevention

- Prevention:
  - do everything so that attacker doesn’t get into your system
  - firewalls, OS hardening (least priv, NX, ASLR, etc…)
- Detection:
  - Am I compromised or not?

- We do need detection!
- We do need a systematic approach to detection! Not just hacks!

Detection vs. Prevention

- If our prevention is not perfect (and it never is!) then:
  - We can **not** be sure that somebody already not exploited the bug in it and “owned our system”
- Updating our prevention system **today** will help to mitigate only **future** incidents. But how do we know that it already hasn’t been exploited? Yes, we need detection!
- Detection, on the other hand, can always be improved and it gives immediate answer whether somebody compromised the system in the past.
- **The point: detection doesn’t need to be 100% to be useful, prevention (if used alone) does!**

How malware gets installed?

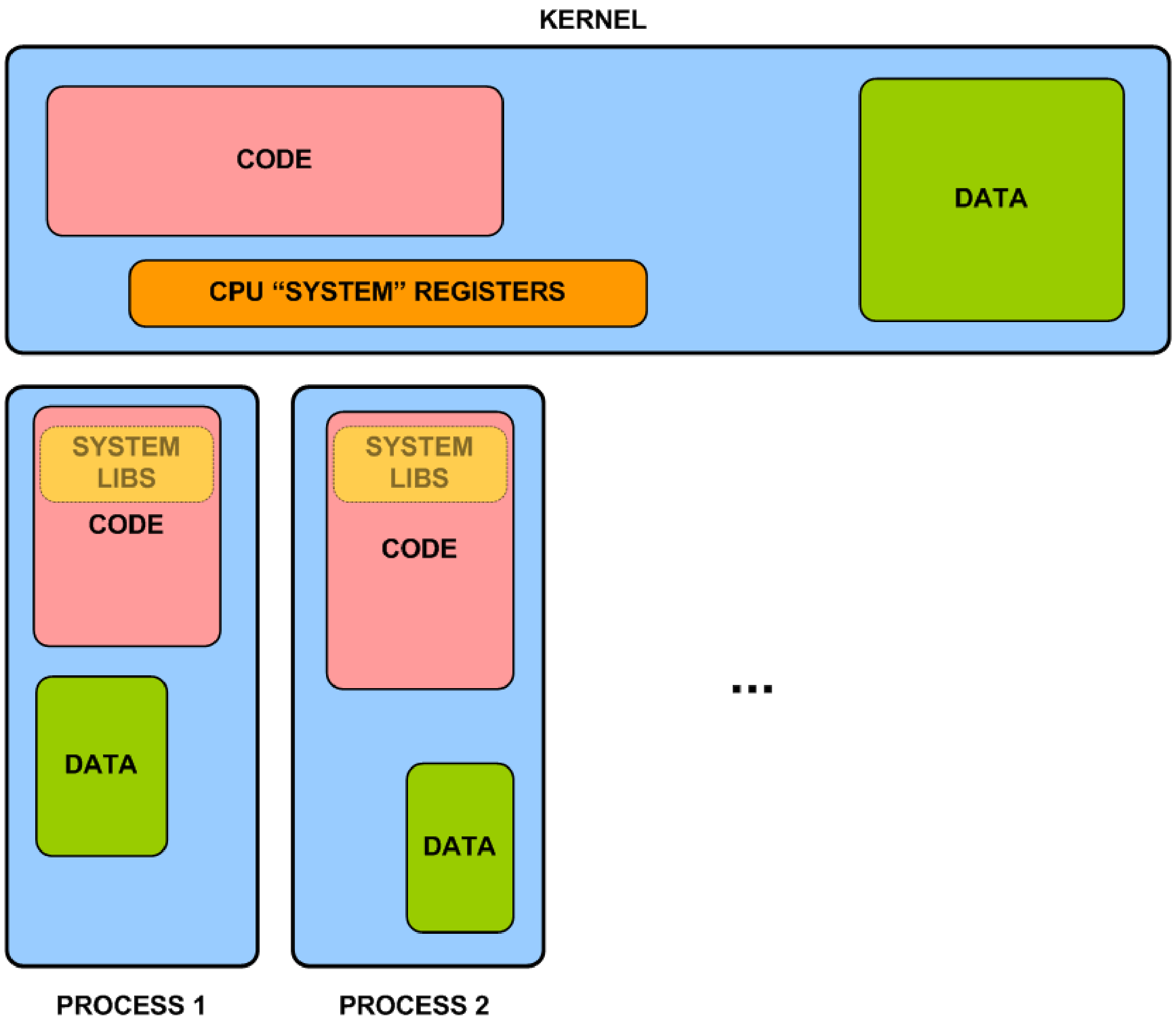
- Is it:
  - Exploit?
  - Unauthorized use of a (stolen) password?
  - Malicious employee?
  - User’s stupidity (click on an attachment)?
- Important for prevention
- Irrelevant for detection!
- We don’t care about the history of infection – we just want to evaluate the state of the system at the present time – here and now!

What is System Compromise?

- Action which *subverts* at least one of the:
  - operating system
  - (critical) applications running in the system
- Does this without user consent
- There is no documented way to find out that the subversion took place

- This is a different definition then all A/V programs use!
- This is a narrowed problem.

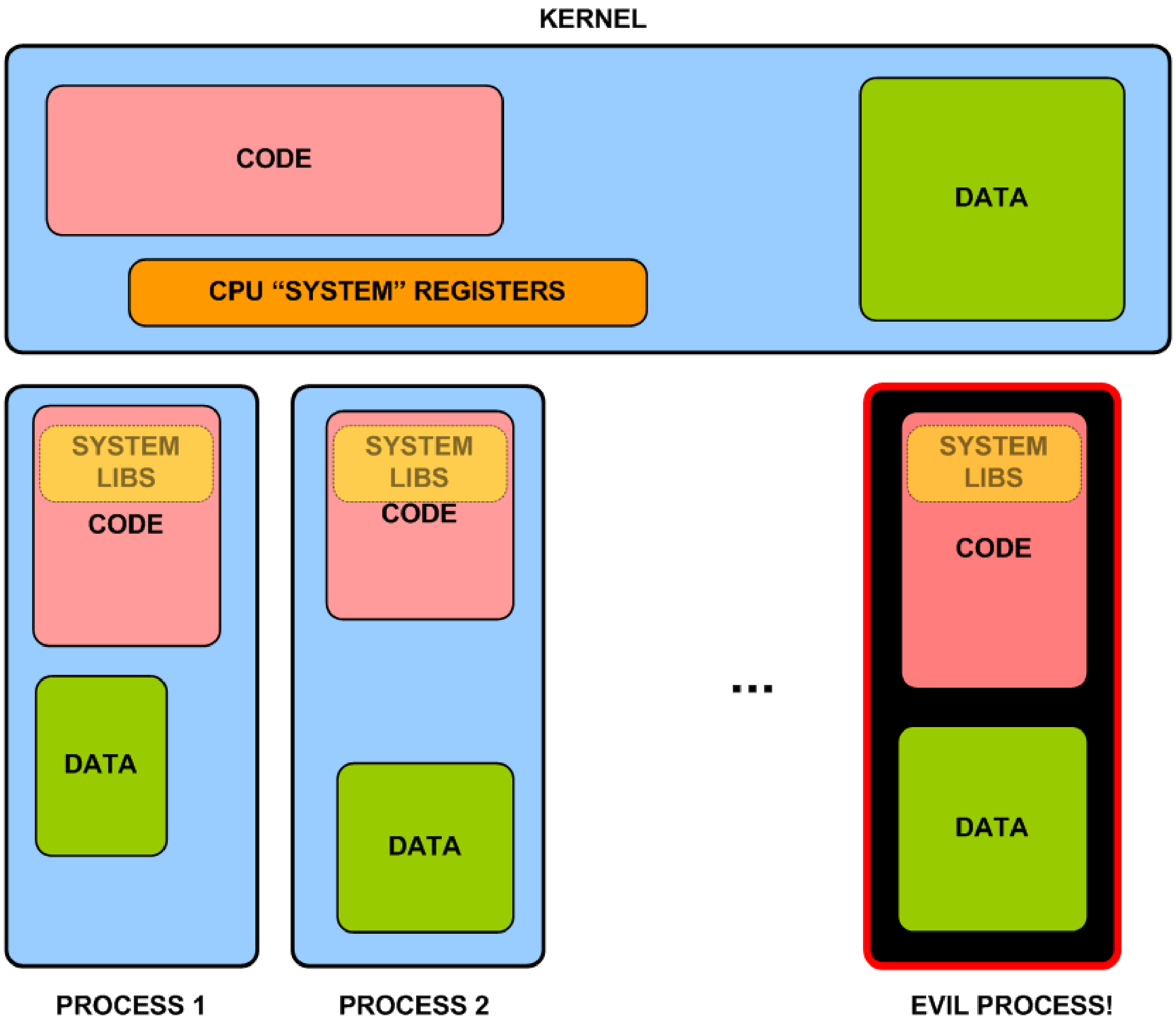
OS View


KERNEL
CODE
DATA
CPU "SYSTEM" REGISTERS
SYSTEM LIBS
CODE
DATA
SYSTEM LIBS
CODE
DATA
...
PROCESS 1
PROCESS 2

Malware classification proposal

- Type 0: Malware which doesn’t modify OS in any undocumented way nor any other process (non-intrusive),
- Type I: Malware which modifies things which should never be modified (e.g. Kernel code, BIOS which has it’s HASH stored in TPM, MSR registers, etc…),
- Type II: Malware which modifies things <u>which are designed to be modified</u> (e.g. DATA sections).
- Type III: Malware which doesn’t modify OS nor Apps at all – but still intercepts and controls the system!

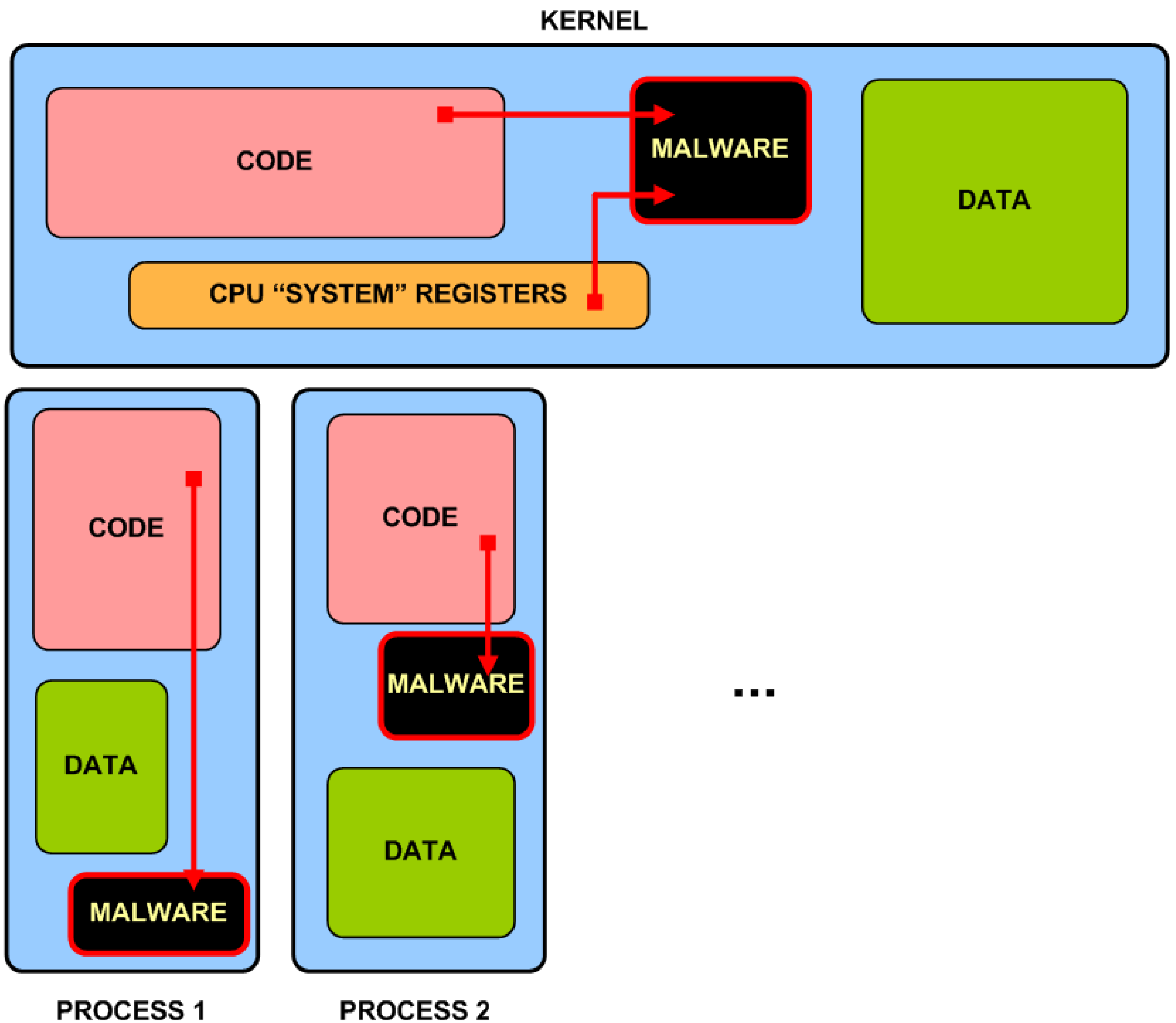
Type 0 Malware



Type 0 Malware

- Doesn't fall under *my* the definition of system compromise!
- They do not compromise other Applications or kernel.
- They might be "bad", but they not make other's applications "bad".
- Examples include all simple botnets, trojans, etc…, which are implemented in the form of a standalone application (i.e. do not interact with other processors nor kernel)
- Favorite are of research of all A/V vendors ;)
  - Whether evil.exe is "bad" or "legitimate"?

Type I Malware



Type I Malware examples

- Hacker Defender (and all commercial variations)
- Sony Rootkit
- Apropos
- Adore (although syscall tables is not part of kernel code section, it’s still a thing which should not be modified!)
- Suckit
- Shadow Walker – Sherri Sparks and Jamie Butler
  - Although IDT is not a code section (actually it’s inside an `INIT` section of `ntoskrnl`), it’s still something which is not designed to be modified!

Type I Malware detection

- We need to verify that all those “constant things”, like e.g. code sections, has not been touched…
- We need a *baseline* to compare with
  - Hash (requires “learning phase”)
  - Digital signature (requires some sort of PKI)

Digital Signatures: Prevention vs. Detection

- Executables signing may play two different roles:
  - To prevent execution of untrusted code (prevention)
  - To allow for detection of code modifications (detection)
- The preventive role of code signing usually doesn't work too well – there are always many ways to bypass it (e.g. exploit + arbitrary shellcode)…
- But that is not important for us as we focus on **detection!**

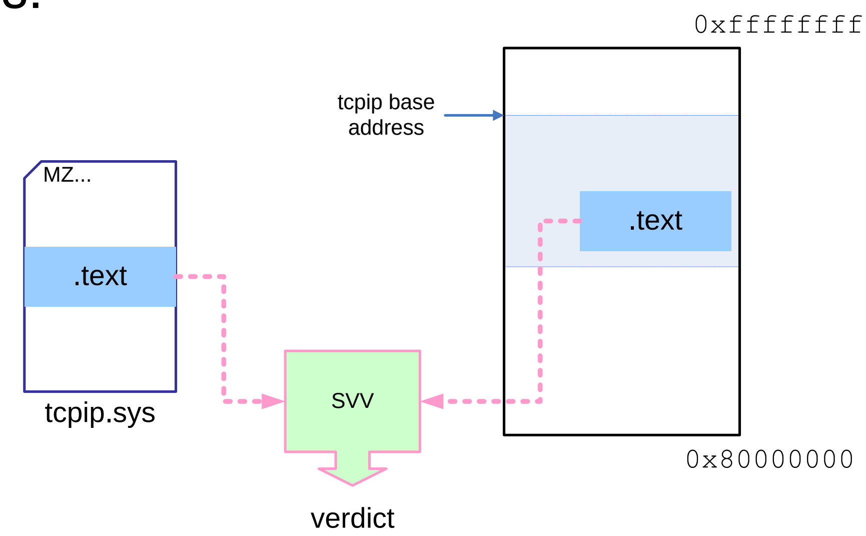
Fighting Type I malware on Windows

- VICE
- SVV
- Patch Guard by MS on 64 bit Windows

- Today's challenge: false positives
- Lots of nasty apps which use tricks which they shouldn't use (mostly AV products)
- Tomorrow: Patch Guard should solve all those problems with false positives for Type I Malware detection…
- … making Type I Malware detection a piece of cake!

System Virginity 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…



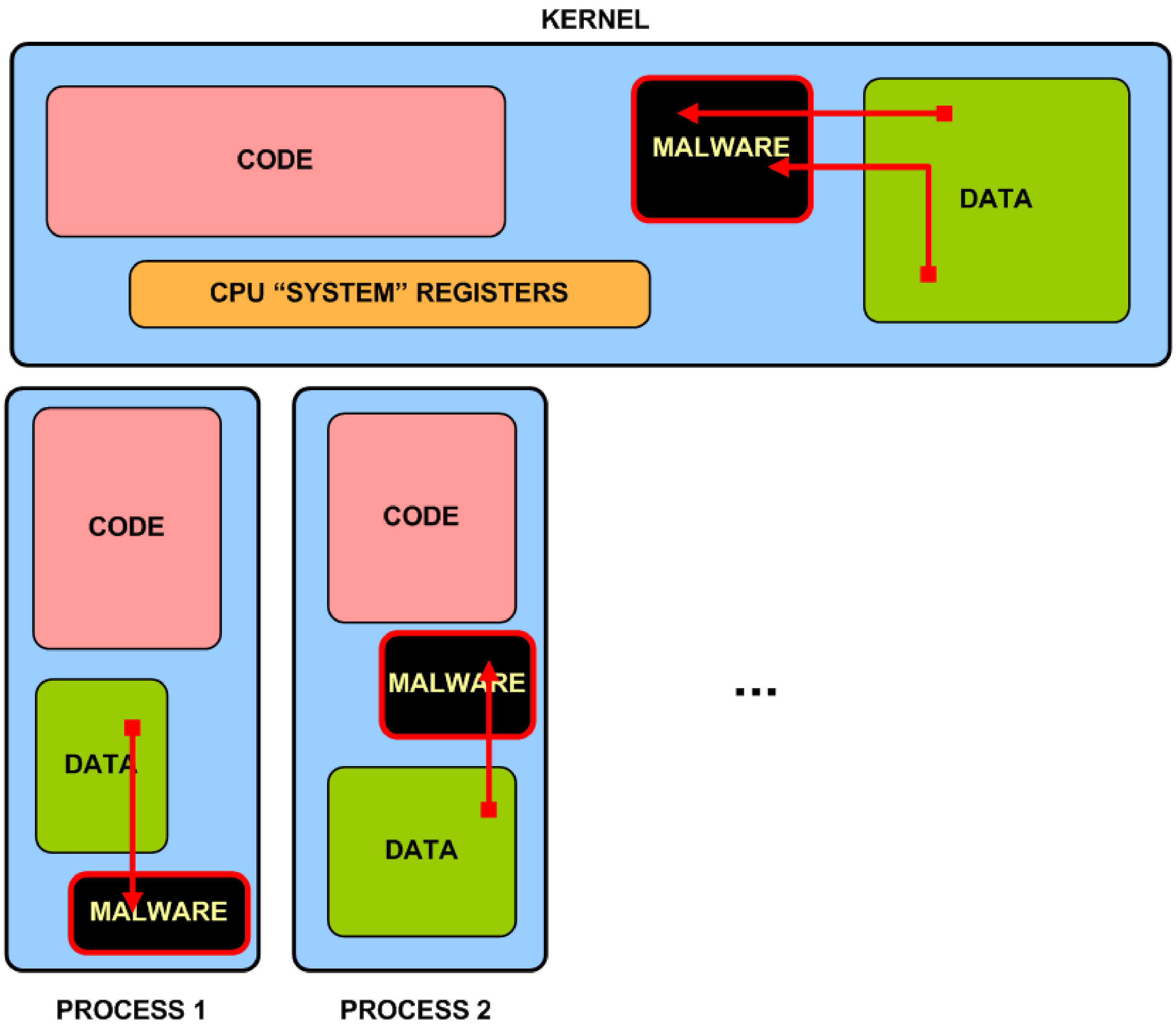
Patch Guard

- By Microsoft, to be (is) included in all x64 Windows
  http://www.microsoft.com/whdc/driver/kernel/64bitPatching.mspx
- Actions forbidden:
  - Modifying system service tables
  - Modifying the IDT
  - Modifying the GDT
  - Using kernel stacks that are not allocated by the kernel
  - Patching any part of the kernel (detected on AMD64-based systems only) [*I assume they mean code sections here*]
- Can PG be subverted? Ask Metasploit ;)
  - Also PG doesn't prevent Type II and Type III malware
- But this is not important!

Patch Guard

- Important thing is: PG should force all the *legal* (non malicious) apps to not use all those rootkit-like tricks (which dozens of commercial products use today)
- PG should clear the playground, making it much easier to create tools like SVV in the future
- It won't be necessary to implement smart heuristics to distinguish between Personal Firewall-like hooking and rootkit-like hooking
- It's unlikely that PG bypassing techniques could be used by serious software companies, because it will give MS the right to treat their products as malware!

Type II Malware



Type II Malware examples

- He4Hook (only some versions) – Raw IRP hooking on fs driver
- prrf by palmers (Phrack 58!) – Linux procfs smart data manipulation to hide processes (possibility to extend to arbitrary files hiding by hooking VFS data structures)
- FU by Jamie Butler, FUto by Jamie and Peter Silberman
- PHIDE2 by 90210 – very sophisticated process hider, still however easily detectable with X-VIEW...
- Deepdoor by yours truly

Fighting Type II Malware

- There are three issues here:
  - To know where to look
  - To understand what we read
  - To be able to read memory
- But… we all know how to read memory, don’t we?

Type II Malware Detection cont.

- “To know where to look” issue
  - there is lots of data inside the OS…
  - how to make sure that we check all the potential places?
- Consider network backdoor implementation on Windows:
  - …
  - TDI hooking
  - NDIS additional protocol registered
  - NDIS_OPEN_BLOCK hooking (deepdoor)
  - X_BINDING_INFO hooking (Alex Tereshkin, BH Vegas 2006)
  - …

Type II Malware Detection cont.

- To understand what we read
  - What does it mean that e.g.
  - `openBlk->ReceivePacketHandle == 0xffab0042`
- We need to know how to interpret those values – so, we need to understand the semantics behind the data structures which we need to verify…
- The simple solution in case of *function pointers* is:
  - Check whether it points into a *valid* code section of a *trusted* module
  - We need to first determine whether the section is *valid* and whether the modules is *trusted (*solve Type I detection*)*!
  - BTW, this scheme can still be cheated ;)

Type II Malware Detection cont.

- To be able to read memory
  - Hey, that shouldn't be a problem
  - All computers are just Turning machines, right?
  - Yes, but complicated ones…

Memory Reading: software based

- Usually we use a kernel driver or LKM to access all memory (including kernel)
- This might be implemented as a hypervisor on processors which support hardware virtualization to resist ISA attacks
- However:
  - We need to properly synchronize with memory manager
  - We can read only virtual memory – see Shadow Walker of how easy it is to cheat about virtual memory
  - We need a way to access page directory reliably (and securely) – but how we convert CR3 physical address to virtual address? No, we can't rely on OS "default" mapping here!

Memory Reading: DMA (hardware based)

- Super Reliable!
- Does not require additional software on a target computer – non-invasive
- Hardware is cheap (FireWire cable will do the job)
- But:
  - Can not read paged-out memory :(
  - Also – are you sure that it's actually *that* reliable? ;)

Type II Malware – bottom line

- Today we can not design a systematic detection method against type II malware for most (all?) of the OSes
- Changes (little) in the design of the OS are needed to make type II malware detection feasible – see later.

Towards Systematic Verification of the OS & Applications…

Ultimate Goal

- We want a *recipe* to create a detector
  - Software based
  - Hardware based (DMA)
  - Hypervisor based
- Detector, once run against a given system, would return the *verdict:*
  - 0 – means system is *clean*
  - 1 – means system is *compromised* (i.e. infected with Type I, II or III malware)

Problem

- We don’t know how to solve the problem of Type II malware…
- Because the operating systems are too complex:
  - We don’t know what to check
  - We can’t safely and reliably read memory

Changing the rules a bit...

- But maybe we could change the rules of the game a little bit?
- How about we introduced the following requirement :

  The only executable pages in the system (kernel) are those which contain trusted code. All others pages should be marked as non-executable.

- For now, let's focus on kernel only, so we avoid the problem of some usermode applications, which would like to violate this requirement: e.g. JIT compilers.

Verifiable OS: requirements

1. Underlying processor must support non-executable attribute on a per-page level
2. OS must maintain strong data/code separation on a per-page level
3. There must exist a way to verify code on a per-page level (e.g. digital signatures implemented)
4. OS must allow to safely read raw memory by a 3rd party program (kernel driver). Alternatively, paging must be disabled, if hardware based access is to be used.

Verification of the OS

For each memory page in the system:

- If the page is marked as executable then check if the code contained within the page is trusted (e.g. verify the fingerprint).

Code/Data separation in various OSes

- Code/Data separation becomes more and more popular recently as a way to mitigate exploitation attempts (AKA non-exec)
- Windows x64 (including Vista):
  - Kernel stack, Paged Pool, Session Pool – NX bit set
  - All other pages are executable, including those from Non-paged pool
- NetBSD – implements code/data separation as a result of W^X policy (probably also Open- and FreeBSD)
- Linux – PaX enforces code/data separation (not sure if it works in kernel already, on only in usermode)

Code signing in various OSes

- Windows:
  - all system executables are digitally signed
  - Vista x64: all 3rd party kernel drivers are signed
- NetBSD: veriexec allows for associating fingerprints with files and then for per-page verification (see later)
  - Doesn’t work with digital signatures, yet
- Linux: as a 3rd party extensions only (?)
- Solaris: Implements signed binaries (ELFs), starting from version 10

NetBSD – the first verifiable OS?

- Veriexec – could be used to implement verification of code on a page-level granularity
- Work done by Brett Lymn
- Implements code/data separation on page-level granularity (as a result of W^X)
- Memory reading problem is still not solved
  - but is being looked into by Elad Efrat :)
- We should get some running POC within c months!
- Ask Elad Efrat for more details!

Implementation Specific Attacks (ISA)

- Any given program (executable) can be cheated using an implementation specific attack! (if it runs with the same privileges as malware does)
- E.g. we can patch the 'IF' statement in the program:

```
If (kernelInfected)
    printf ("Dear user, you're in troubles!\n");
```

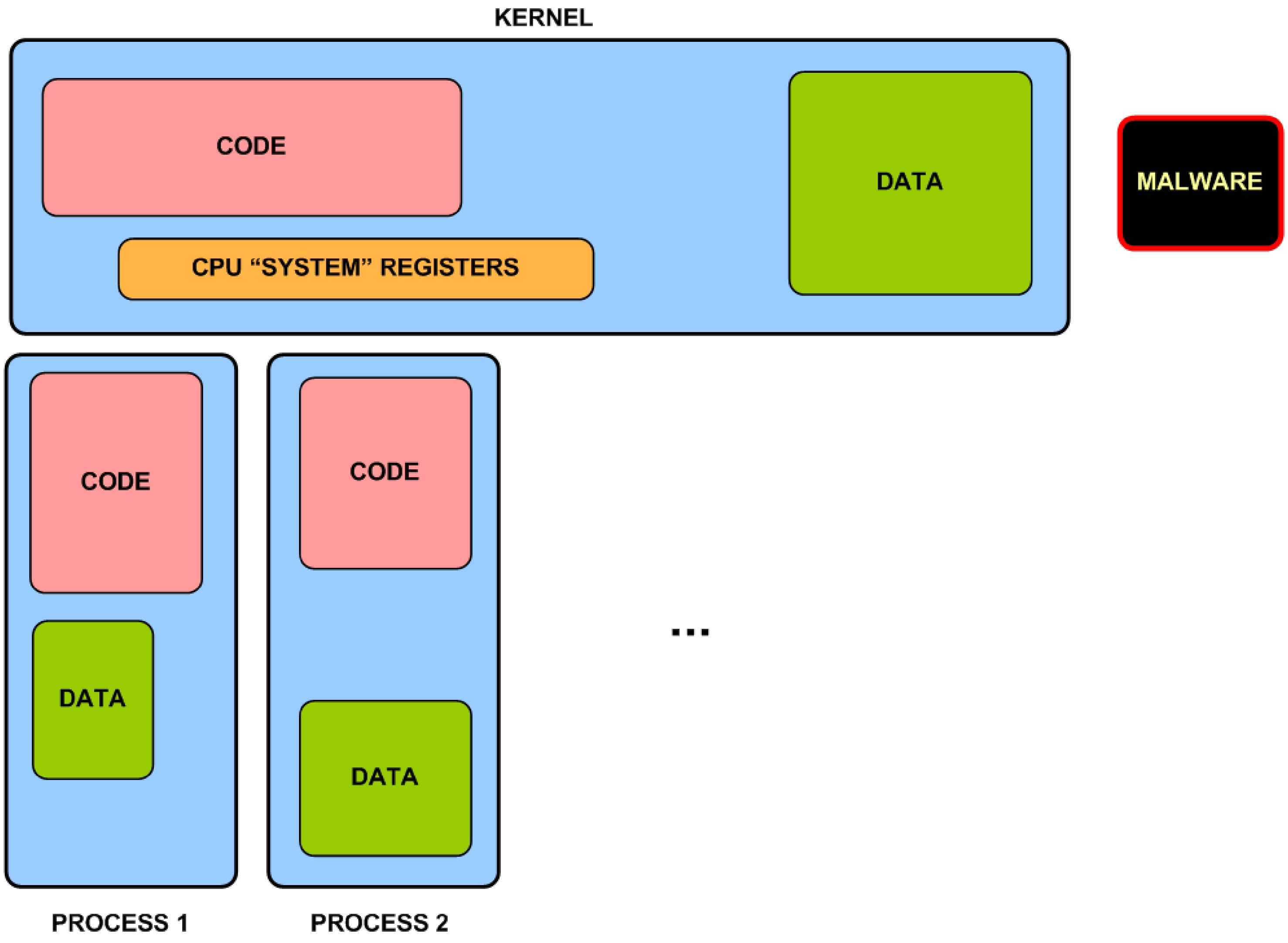
- It's trivial for a rootkit to implement ISA against well known detectors – see e.g. commercial Hacker Defender rootkit (now defunct)

Mitigating ISA

- Detector/Verifier located in hypervisor
- Using a hardware based memory access
  - PCI card
  - FireWire
  - Problem with paged-out memory
- Using a non-public detector
  - acceptable in e.g. corporate environment

Now imagine we have solved all those problems…

Type III Malware



Type III malware examples

- Vitriol (Dino Dai Zovi, Black Hat Vegas 2006) – abuses Intel VT-x virtualization technology, works on MacOS
- Blue Pill (J. Rutkowska, SyScan/Black Hat Vegas 2006) – abuses AMD SVM virtualization technology, works on Vista x64.

Fighting Type III malware

- Timing analysis in case of virtualization based malware
  - Practical problems (trusted time source?)
  - Next generation VM based software might be immune to timing analysis – ongoing research – stay tuned ;)
- Detecting side effects
  - **Network communication**
  - Disk usage (in case of persistent malware – not necessarily hidden files, but e.g. infected files)

Type III malware detection in practice

- Today it seems that it’s not possible to detect (verify OS against) type III malware in any systematic way :(
- We may imagine exploiting a bug in hardware virtualization implementation which would reveal the presence of a hypervisor (something ala redpill)…
- … but that would be just that – a (temporarily) hack
- And we want a systematic way (documented, legal, reliable)!

Hardware Red Pill?

- How about creating a new instruction – **SVMCHECK:**

```
mov rax, <password>
svmcheck
cmp rax, 0
jnz inside_vm
```

- Password should be different for every processor
- Password is necessary so that it would be impossible to write a *generic* program which would behave differently inside VM and on a native machine.
- Users would get the passwords on certificates when they buy a new processor or computer
- Password would have to be entered to the AV program during its installation.

kernel protection vs. hypervisor protection

- On an average general purpose OS there is *a lot of* kernel code:
  - core kernel
  - all kernel drivers
- Making sure there is no bug in all kernel mode software is impossible – kernel prevention will never be satisfactory
- Hypervisor can be very thin – easily auditable and most importantly there is no need for 3rd party code in hypervisor (ala kernel drivers).
- Thus: effective hypervisor protection seems feasible, (unlike kernel protection).

Bottom Line

- Type 0 malware is beyond the scope of my research
- Type I malware is relatively easy to fight
- We can’t fight type II malware effectively without changing the OS design
- ISA are always possible, but we can mitigate those attacks in practice
- Systematic verification against type III malware seems to be unfeasible without the help from CPU-vendors (hardware redpill)
- Prevention against type III malware seems feasible

Happy New Year?

- Gartner: *10 Key Predictions for 2007*:

#5: By the end of 2007, 75 percent of enterprises will be infected with undetected, financially motivated, targeted malware that evaded their traditional perimeter and host defenses. (source: eWeek)

Stealth Malware – The Battle

- So, can the good guys win?
  - No, unless OS vendors will join the game!
- **Who can we trust**, then?
  - For sure not our operating systems :(

Credits

- Elad Efrat of NetBSD
- All the people behind cool prevention projects :)
  - PaX, grsecurity, veriexec/Stephanie

Thank you!

joanna@research.coseinc.com