

# Malware-Analyse und Reverse Engineering

12: Rootkits

22.6.2017

Prof. Dr. Michael Engel

# Überblick

## Themen:

- Rootkits
  - User Mode
  - Checksumming von Binaries
  - Kernel Mode
  - Speicher-Rootkits
  - VM-Rootkits

# Rootkits

## **Zweck: Permanenter Zugriff auf ein kompromittiertes System**

- Kompromittierung des Systems durch eine Sicherheitslücke
  - Buffer overflow etc. => root-Shell
- Installation zusätzlicher Softwarekomponenten („Backdoor“)
  - Erlauben dem Angreifer zukünftigen Zugriff auf das System
  - Benötigen dafür nicht mehr das Vorhandensein der ursprünglichen Sicherheitslücke
- Verbergen der Aktivität der zusätzlichen Komponenten vor Benutzern und Administratoren des kompromittierten Systems
  - Ziel: so lange wie möglich unentdeckt bleiben
- Verwendung z.B. für Botnetze

# Rootkits (2)

## Herkunft des Namens

- von „root“ für den üblichen Unix-Administratoraccount
- und „kit“ für Bausatz

## Historie und Hintergrund

- Erste Entwicklungen in den 90er-Jahren für Unix-Systeme
  - Ersetzung einiger Systemprogramme auf SunOS [1]
  - Später andere Unix-Systeme wie Ultrix und HP-UX
- Problem von Unix: „allmächtiger“ root-Account
  - Erlangt ein Angreifer root-Rechte, kann er das gesamte System manipulieren

# Funktion von Rootkits

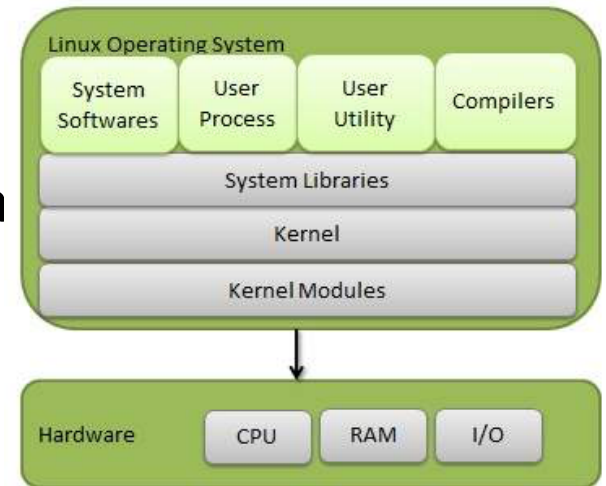
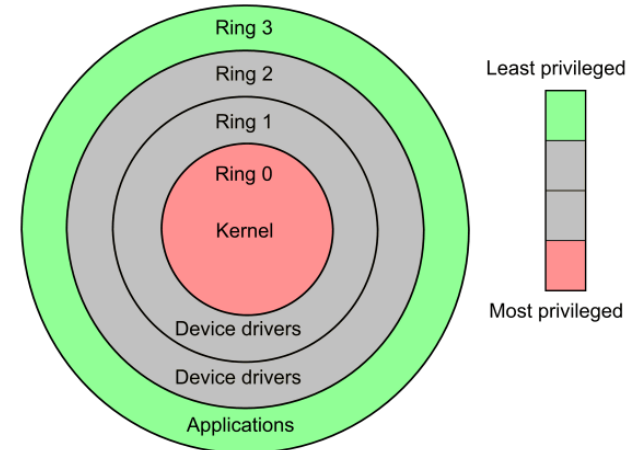
## Zwei Hauptaufgaben von rootkits

- Bereitstellung eines permanenten Zugangs für Angreifer
  - Üblicherweise über Netzwerk (TCP/IP)
  - z.B. Anlegen eines weiteren Users mit root-Rechten (uid 0)
  - Setzen eines Passworts dafür durch Angreifer
  - Öffnen eines Ports und Starten eines Netzwerkdienst-Prozesses mit login-Funktionalität (sshd, telnetd)
- Verbergen der Aktivität des rootkits
  - Verbergen von zum rootkit gehörenden Prozessen
  - Verbergen von zum rootkit gehörenden Daten
  - Verbergen von zum rootkit gehörenden Netzwerkverbindungen

# Kernel vs. User Mode Rootkits (1)

## Ansätze zum Verbergen des rootkits:

- Manipulation von Systemprogrammen, die im User Mode ablaufen
  - z.B. Überschreiben des „ps“-Programms durch Version, die rootkit-Prozesse ausblendet
- Manipulation von shared libraries
  - Mit LD\_PRELOAD wird das Verhalten von einzelnen Systemaufrufen verändert, z.B. in libc
  - z.B. filtert getdirentries(2) Dateien des rootkits aus



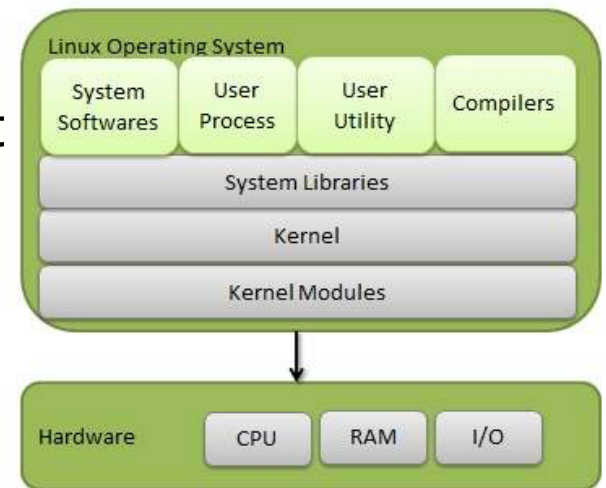
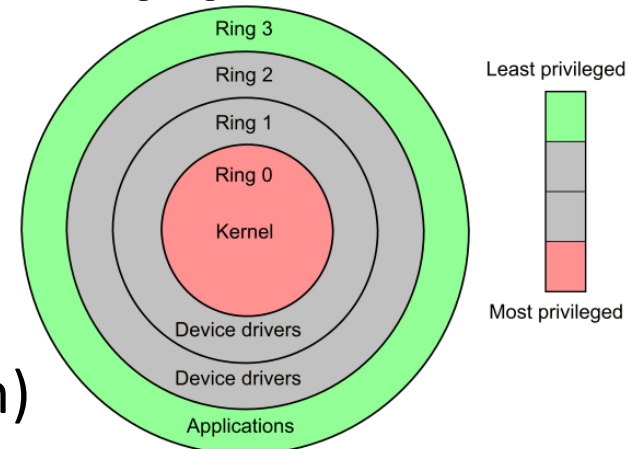
# Kernel vs. User Mode Rootkits (2)

## Problem der Ansätze im user mode:

- Evtl. leicht zu erkennen (Prüfen von Checksummen, z.B. md5, sha1, sha2, sha3) => muss auch manipuliert werden)

## Anderer Ansatz: rootkit im kernel mode

- Moderne Betriebssysteme stellen Schnittstellen für ladbare Module bereit
  - z.B. Gerätetreiber, Dateisysteme
- Module werden ähnlich wie shared libraries *zur Laufzeit* geladen
  - Code der Module läuft im Kernelmodus => hat alle Zugriffsrechte!



# Checksumming von Programmen (1)

## Idee: Erzeugen eines Hashwertes über die Bytes des ELF-Binaries

- Hilft nur gegen Manipulation von Binaries im user mode
- Oft genutzte Verfahren: md5 oder sha1
  - Kryptographische Hashverfahren, die (relativ) kurzen eindeutigen Wert aus dem Inhalt der Datei generieren
  - MD5 verwendet 128 Bit lange Hashwerte
  - sha1 verwendet 160 Bit
- Beispiel:
  - `$ md5 /bin/ls`  
`MD5 (/bin/ls) = ec6643db985ec684c66e8ae4a9996b87`
- Problem: *Kollisionen* der Hashwerte
  - Zwei unterschiedliche Dateien erzeugen selben Hashwert



# Checksumming von Programmen (2)

## Zwei unterschiedliche Dateien erzeugen selben Hashwert

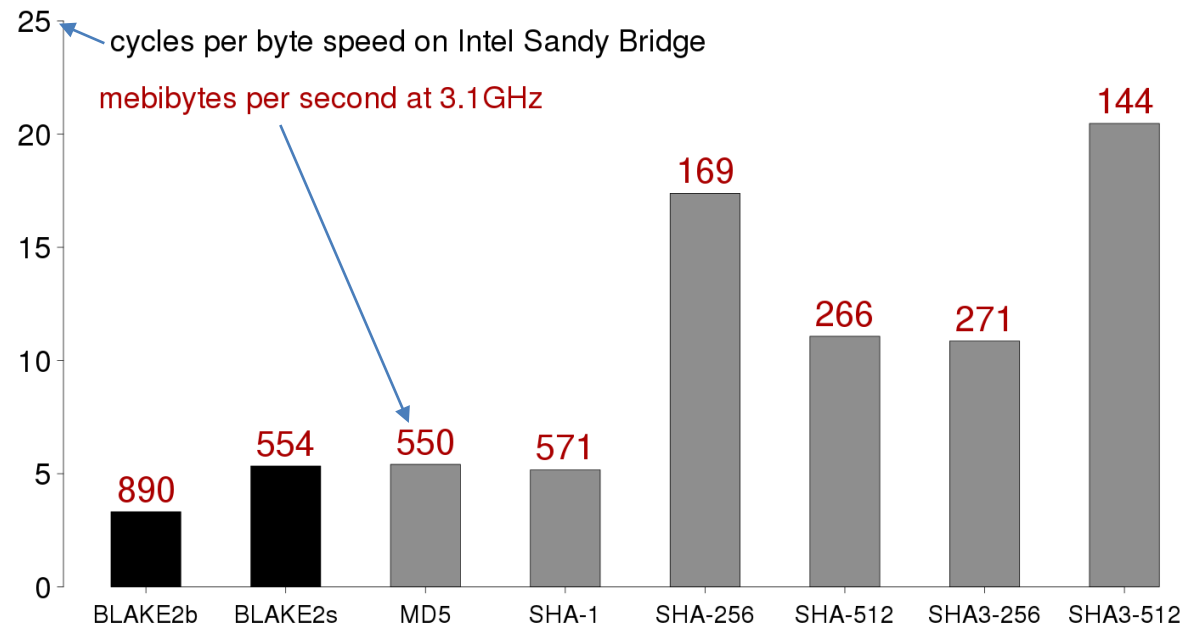
- **Klar: ist nicht vermeidbar**
  - MD5 verwendet 128 Bit lange Hashwerte, sha1 160 Bit
    - 128 Bit =>  $3 \cdot 10^{39}$  verschiedene Werte
  - Aber: Schon bei Datei mit 1 kB Länge:  $2^{1024}$  Byte =  $2^{8192}$  Bit  
 $\sim 1 \cdot 10^{2467}$  (!) verschiedene Dateien!
- Annahme: *Wahrscheinlichkeit* der Kollision nahe 0
  - Aber: Erfolgreiche Angriffe gegen md5 und sha1
  - Datei mit anderem Inhalt, aber gleichem Hashwert erzeugen
  - Benötigt viel Rechenleistung, mittlerweile aber machbar
    - Bereits 2009 war md5-Kollision mit zwei Nvidia GeForce 9800 GPUs in ca. 35 Minuten berechenbar [2]

# Checksumming von Programmen (3)

## Aktuelle Verfahren: sha2 (2001) und sha3 (2015)

- Verwenden aufwendigere Berechnungen
- Größerer interner Zustand der Algorithmen
- **Aber**

- Benötigen mehr Rechenzeit zum Erzeugen des Hash-Wertes



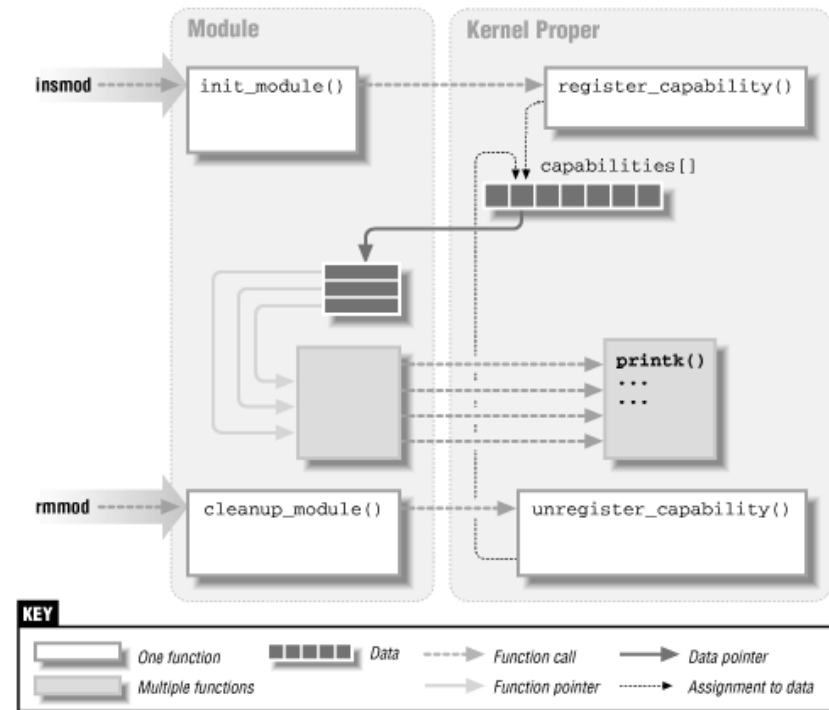
# Kernel Mode Rootkits

## Vorteil von kernel mode rootkits:

- Nicht mehr von Unix-Rechten abhängig => aller Kernel-Code hat vollen Zugriff auf Hardware
- Schwieriger zu erkennen als user mode rootkits

## Weiter reichende Möglichkeiten zur Manipulation des Systems

- Veränderung des Systemaufruf-Verhaltens (ähnlich zu libc)
- Bereitstellung von Diensten im Kernel (z.B. Netzwerkdienste)
- Virens Scanner haben auch mit Verhaltensanalyse keine Chance, das rootkit zu entdecken

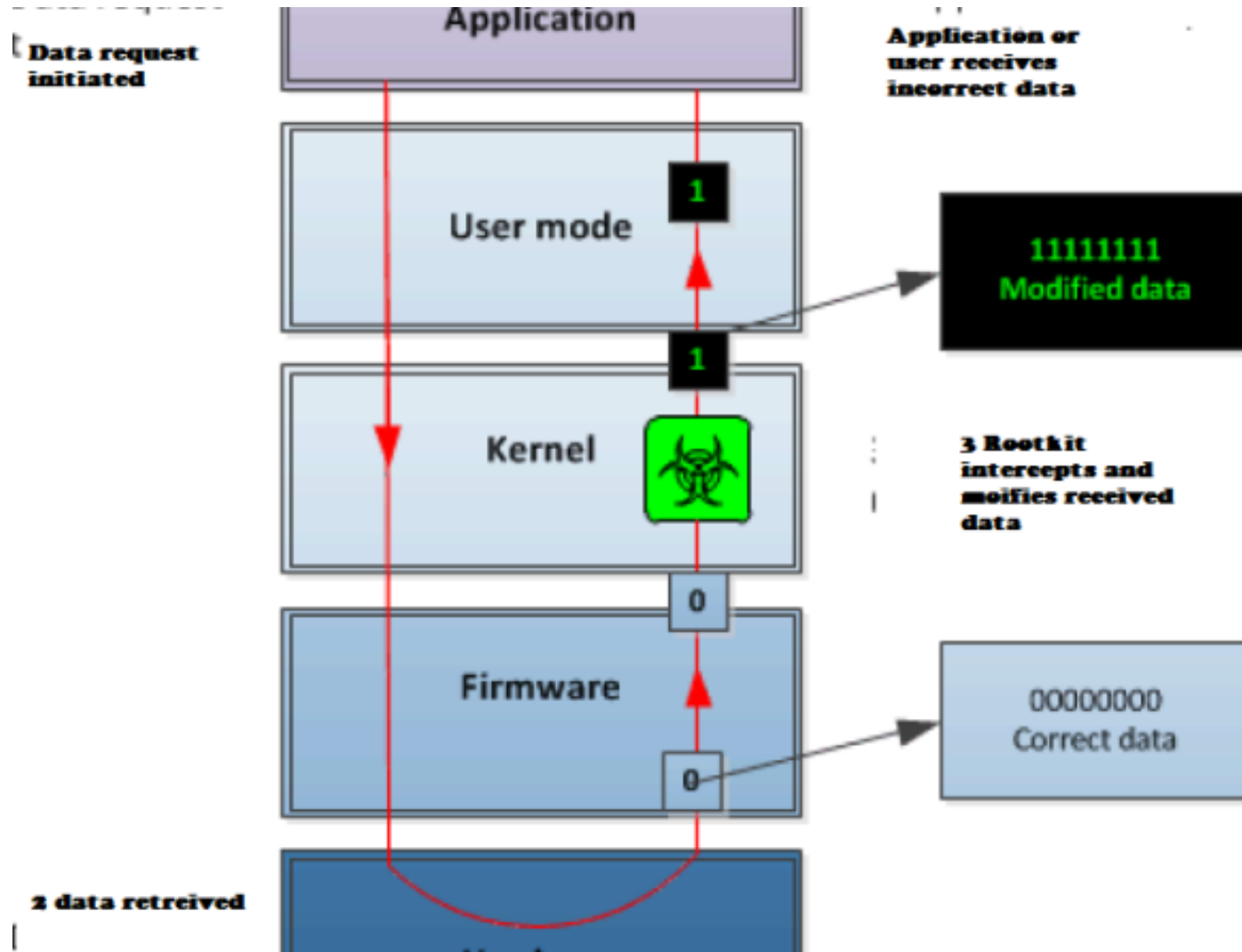


# Kernel Mode Rootkits

## „Vorteil“ von kernel mode rootkits:

- Von user mode aus schwer zu erkennen [3]
- Direkter oder indirekter Zugriff auf Kernel-interne Datenstrukturen und Funktionen
  - Direkt: Symbol (von Struktur, Array, Funktion, etc.) ist vom Kernel an Modul exportiert
  - Indirekt: Mehr Aufwand, programmatisches Nachverfolgen von Adressen von einer Basisadresse aus
- Direkte Manipulation von Prozesstabelle, Puffern, Dateideskriptoren, Dateisystemen usw...

# Kernel Mode Rootkits



# Speicher-Rootkits

## Nur zur Laufzeit im Speicher

- Spurlos entfernbar: nach Reboot nicht nachweisbar
- Manipulation des Speichers
  - Unter Linux ist (fast) alles ein Gerät, auch der Speicher...
  - `/dev/mem`: Zugriff auf den physikalischen Hauptspeicher
  - `/dev/kmem`: Zugriff auf den virtuellen Speicher des Kernels (inkl. ausgelagerter Seiten im Swap)
  - `/dev/port`: Zugriff auf x86 I/O-Ports
  - Normalerweise nur für root möglich (geregelt durch Zugriffsrechte auf die Devicedateien in `/dev`)
- Erste Ansätze 1998 in [4] veröffentlicht

# Speicher-Rootkits: Beispielcode (1)

## Schreiben in physikalischen Speicher

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <phys_addr> <offset>\n", argv[0]);
        return 0;
    }

    off_t offset = strtoul(argv[1], NULL, 0);
    size_t len = strtoul(argv[2], NULL, 0);
    ...
}
```

# Speicher-Rootkits:

## Beispielcode (2)

```
// Truncate offset to a multiple of the page size, or mmap will fail.
size_t pagesize = sysconf(_SC_PAGE_SIZE);
off_t page_base = (offset / pagesize) * pagesize;
off_t page_offset = offset - page_base;

int fd = open("/dev/mem", O_SYNC);
unsigned char *mem = mmap(NULL, page_offset + len,
                          PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, page_base);
if (mem == MAP_FAILED) {
    perror("Can't map memory");
    return -1;
}

size_t i;
for (i = 0; i < len; ++i)
    printf("%02x ", (int)mem[page_offset + i]);

return 0;
```

```
}
```



# VM rootkits

## Auch Kernel-basierte rootkits sind erkennbar...

- Im Zweifel mit Hilfe eines weiteren Kernelmoduls!

## Kann man eine Erkennung im Kernelmodus vermeiden?

- Klassische Lösung aus der Informatik:  
Eine weitere Abstraktionsebene  
einführen! 😊
  - Hier: Virtualisierung des Systems:  
schafft Ring -1
- Erste Ansätze: SubVirt [5], blue pill [6]

# Software- vs. Hardware-basierte Virtualisierung

## S/W based (x86)

- Requires ‘emulation’ of guest’s privileged code
  - can be implemented very efficiently: Binary Translation (BT)
- Does not allow full virtualization
  - sensitive unprivileged instructions (SxDt)
- Widely used today
  - VMWare, VirtualPC

## H/W virtualization

- VT-x (Intel IA32)
- SVM/Pacifica (AMD64)
- Does not require guest’s priv code emulation
- Should allow for full virtualization of x86/x64 guests
- Still not popular in commercial VMMs

aus “Virtualization Based Malware” von Johanna Rutkowska

# Vollständige Virtual Machine Monitors vs. „schlanke“ Hypervisors

## Full VMMs

- Create full system abstraction and isolation for guest,
- Emulation of I/O devices
  - Disks, network cards, graphics cards, BIOS...
- Trivial to detect,
- Usage:
  - server virtualization,
  - malware analysis,
  - Development systems

## “Thin hypervisors”

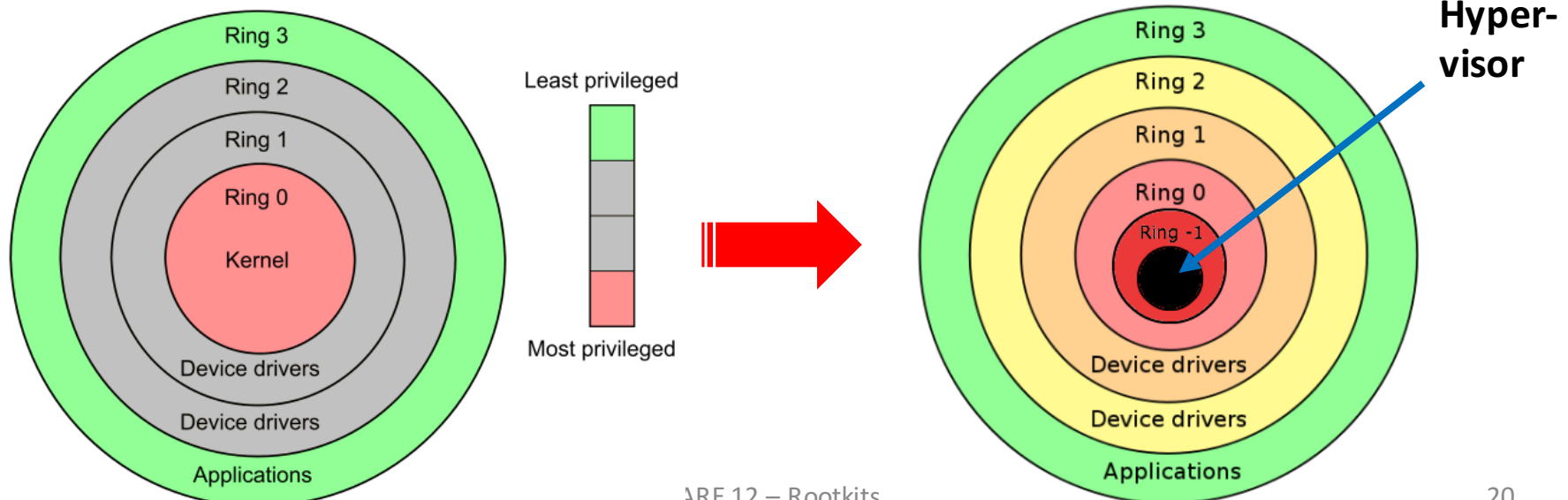
- Transparently control the target machine
- Based on hardware virtualization (SVM, VT-x)
- Isolation not a goal!
  - native I/O access
  - Shared address space with guest (sometimes)
- Very hard to detect
- Usage:
  - stealth malware,
  - Anti-DRM

aus “Virtualization Based Malware” von Johanna Rutkowska

# Hardware-basierte Virtualisierung

## Einführung eines Privilegienrings „unterhalb“ von Ring 0

- Ring -1 – ist für den Hypervisor reserviert
- Kernel läuft wie gehabt in Ring 0, Anwendungen in Ring 3
- Kernelcode kann **unverändert** laufen
  - VM schafft Illusion des Zugriffs auf gesamten Rechner!



# Idee des „Blue Pill“-Rootkits

**Blue Pill „schiebt“ dem Kernel zur Laufzeit einen schlanken Hypervisor unter**

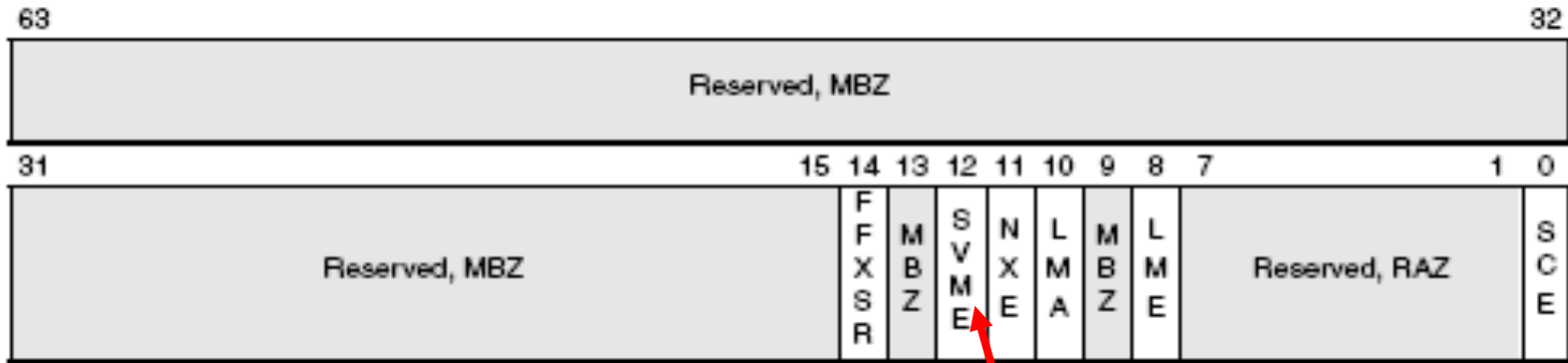
- Verschafft sich dazu zunächst root-Rechte
- Ausnutzung von speziellen SVM-Instruktionen von AMD64 (später auch auf intel)
- Der Hypervisor fängt nur die „interessanten“ Ereignisse innerhalb des (jetzt) Gast-Betriebssystems ab

# Steuerung des SVM-Modus

## Relativ einfache Kontrolle über Konfigurationsbits

- SVM ist die AMD64-Befehlssatzerweiterung zur Implementierung sicherer virtueller Maschinen
- Aktivieren/deaktivieren über MSR (Machine Status Register)  
EFER register: bit 12 (SVME)
- EFER.SVME muss vor der Ausführung einer SVM-Instruktion auf „1“ gesetzt werden (sonst Exception)
- Referenz:  
AMD64 Architecture Programmer's Manual Vol. 2: System Programming Rev 3.11

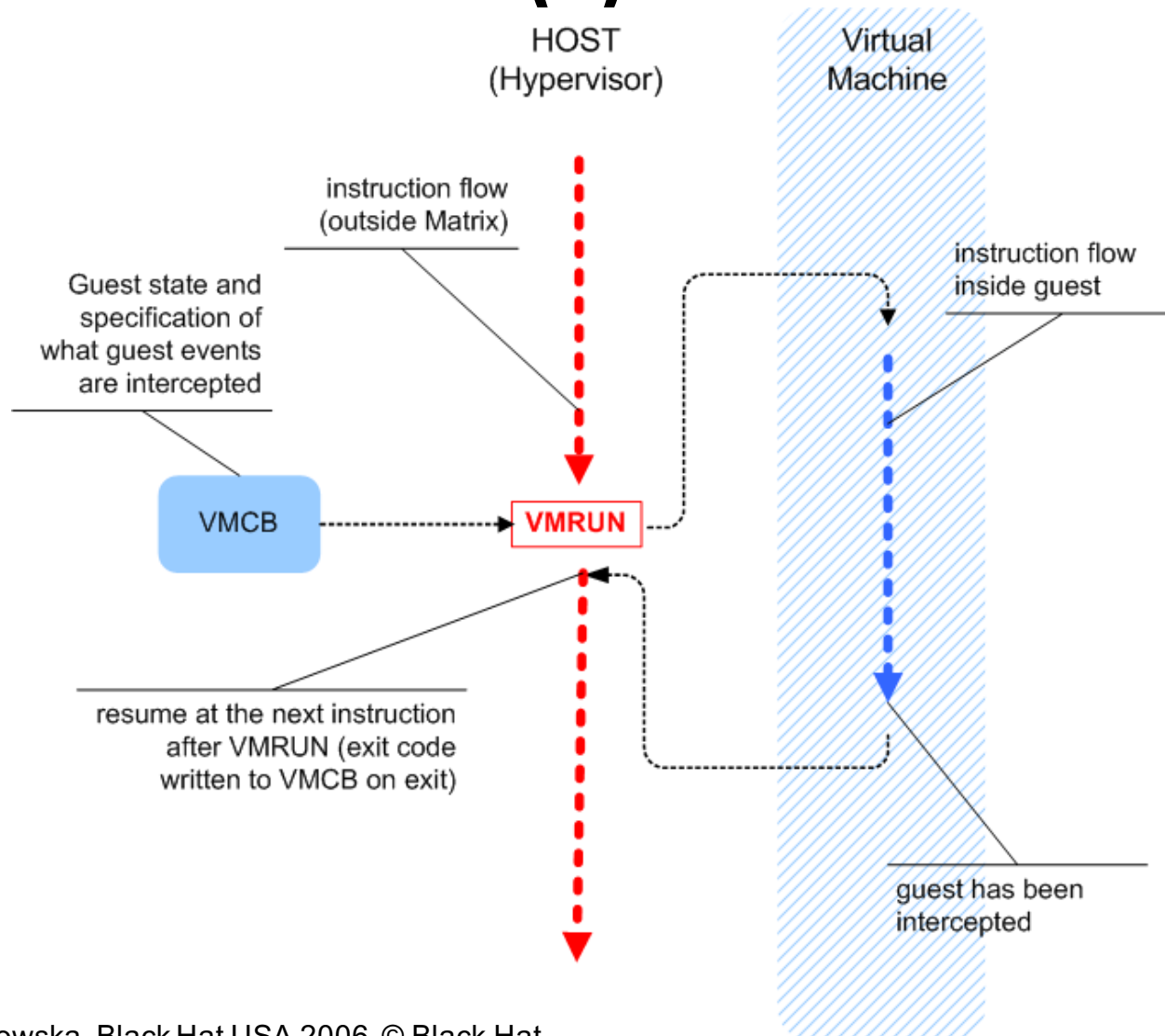
# Aktivieren von SVM: EFER-Register



Bits	Mnemonic	Description	R/W
63–15	Reserved, MBZ	Reserved, Must be Zero	
14	FFXSR	Fast FXSAVE/FXRSTOR	R/W
13	Reserved, MBZ	Reserved, Must be Zero	
12	SVME	Secure Virtual Machine Enable	R/W
11	NXE	No-Execute Enable	R/W
10	LMA	Long Mode Active	R
9	Reserved, MBZ	Reserved, Must be Zero	
8	LME	Long Mode Enable	R/W
7-1	Reserved, RAZ	Reserved, Read as Zero	
0	SCE	System Call Extensions	R/W

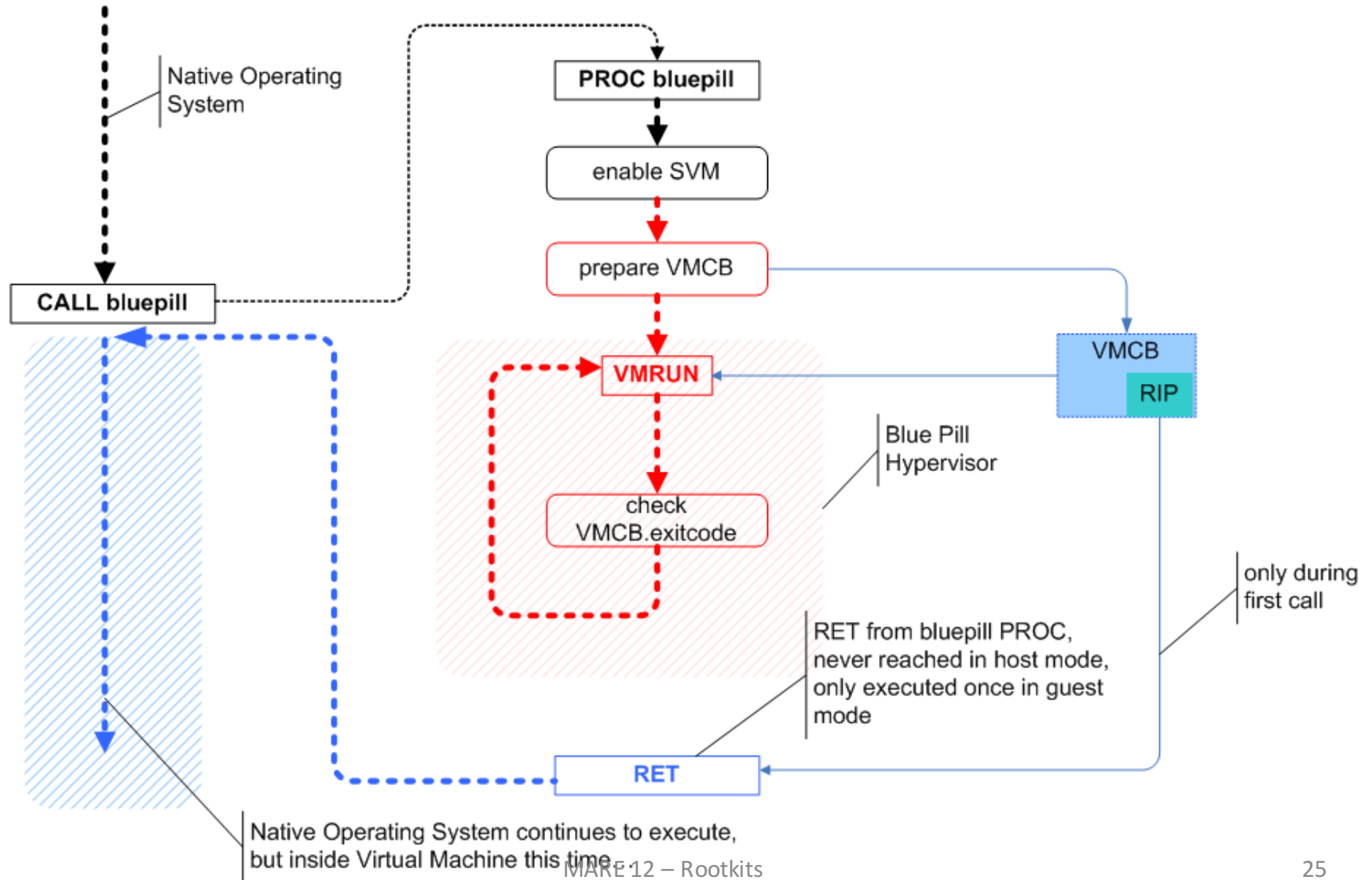
Aktiviert SVM

# Prinzip von Blue Pill (1)





# Prinzip von Blue Pill (2)



# Erkennen einer virtuellen Maschine

## Sind Blue Pill und verwandte rootkits damit unerkennbar?

- Seiteneffekte (z.B. Timing) können zur Erkennung genutzt werden!

## Unterscheidung:

- Erkennung, dass der Kernel innerhalb einer VM läuft (kann natürlich auch legitime Gründe haben)
- Erkennung, dass eine VM-basierte Malware läuft

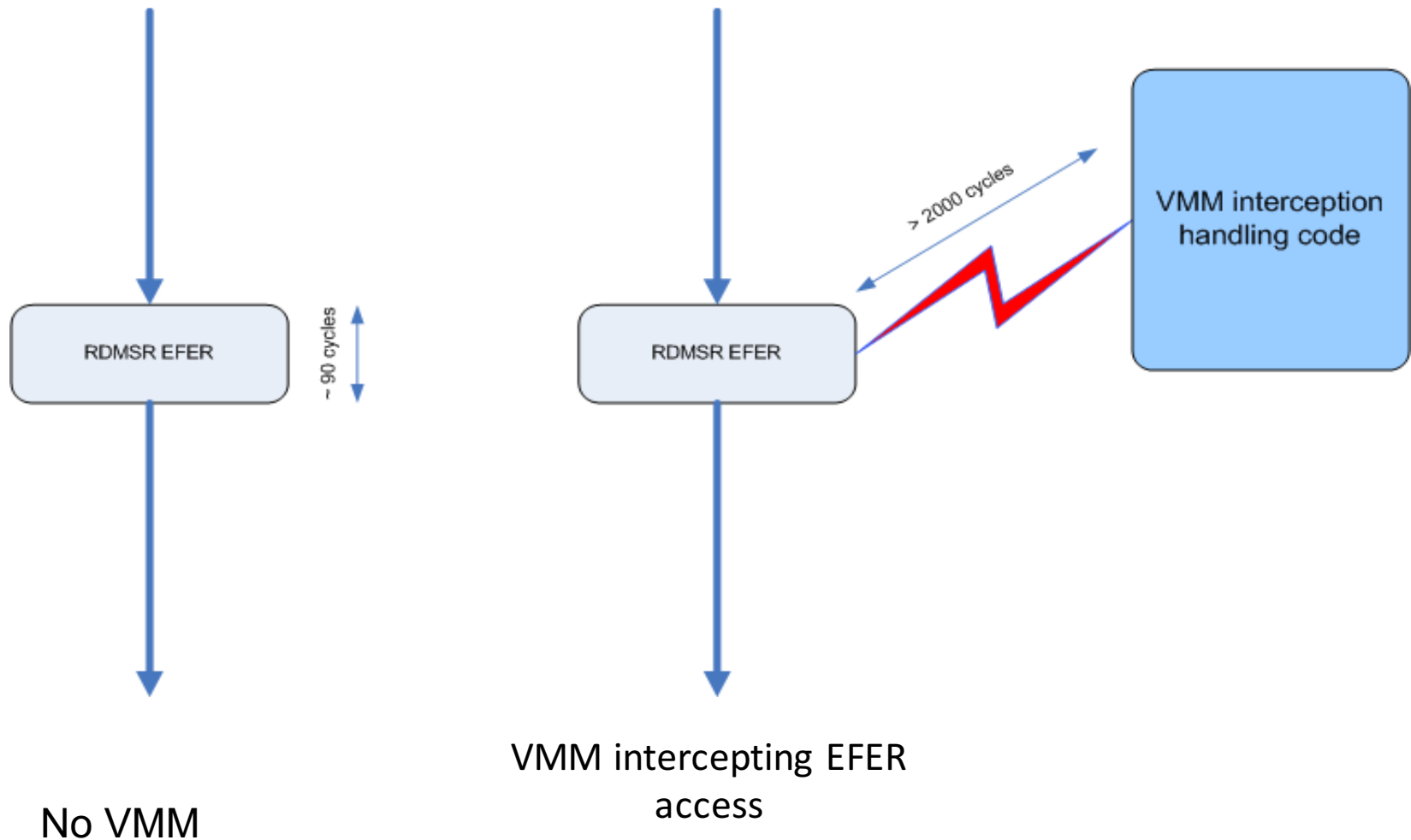
# Virtualisierung erkennen (1)

## Schreiben von Kernelcode...

- der versucht, auf das EFER-Register zuzugreifen
- Direkter Zugriff wird durch Hypervisor verweigert
  - Hypervisor muss dafür sorgen, dass der Kernel das „SVME“-Bit als „0“ (SVM deaktiviert) liest – es ist aber ja „1“
- Bei Kernelzugriff auf EFER-Register: Exception
  - Verursacht Sprung in den Hypervisor
  - Hypervisor simuliert die Instruktion, die aus EFER liest **und löscht dabei das SVME-Bit** im zurückgegebenen Wert/Register

68																32																																							
Reserved, MBZ																																																							
31																15	14	13	12	11	10	9	8	7	1																0														
Reserved, MBZ																F	M	X	S	R	S	V	M	E	N	X	E	L	M	A	M	B	Z	L	M	E	Reserved, RAZ																S	C	E
Bits	Mnemonic	Description	R/W																																																				
63-15	Reserved, MBZ	Reserved, Must be Zero																																																					
14	FXSR	Fast FXSAVE/FXRSTOR	R/W																																																				
13	Reserved, MBZ	Reserved, Must be Zero																																																					
12	SVME	Secure Virtual Machine Enable	R/W																																																				
11	NXE	No-Execute Enable	R/W																																																				
10	LMA	Long Mode Active	R																																																				
9	Reserved, MBZ	Reserved, Must be Zero																																																					
8	LME	Long Mode Enable	R/W																																																				
7-1	Reserved, RAZ	Reserved, Read as Zero																																																					
0	SCE	System Call Extensions	R/W																																																				

# Virtualisierung erkennen (2)



# Virtualisierung erkennen (3)

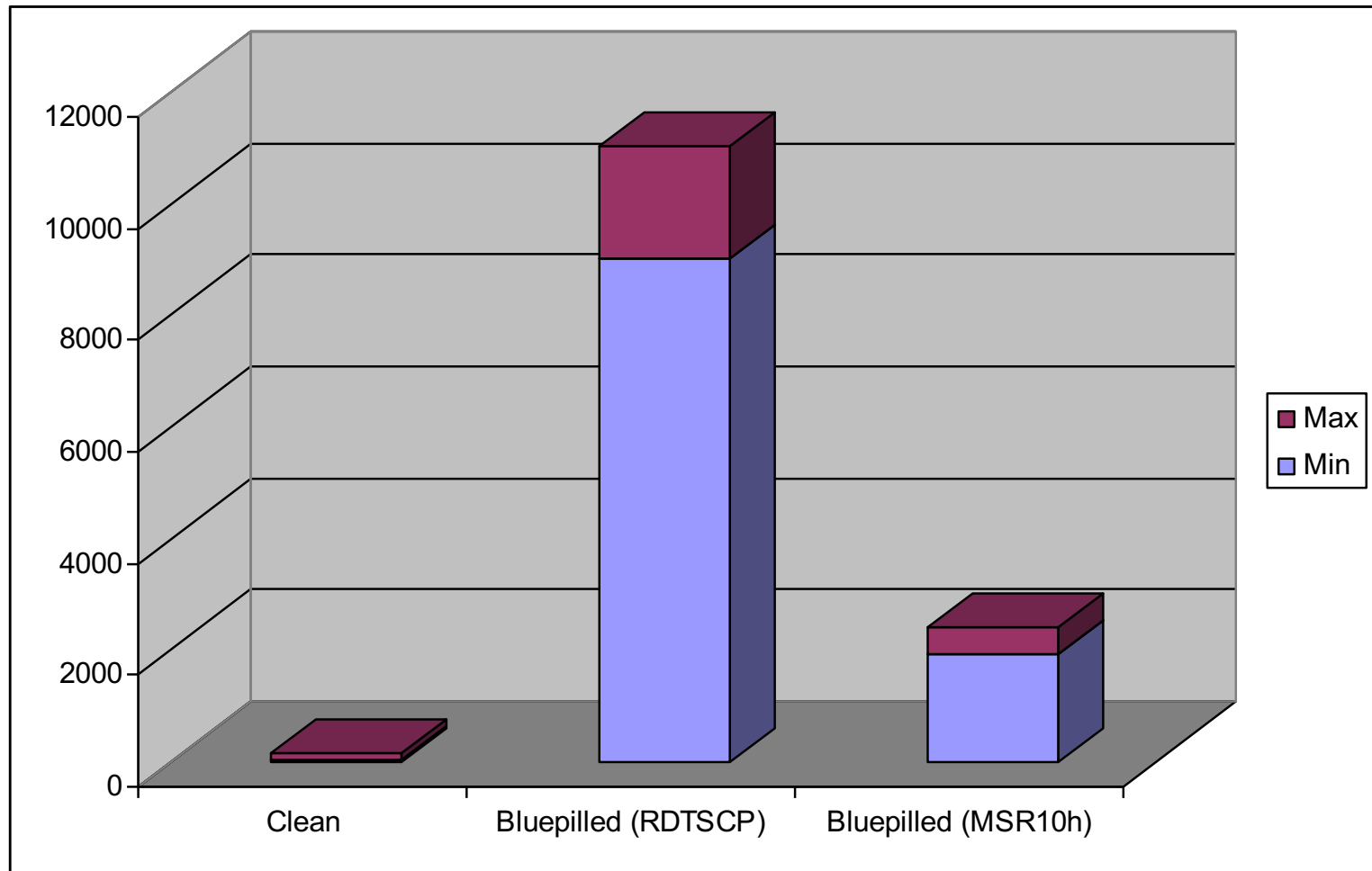
- Hypervisor simuliert die Instruktion, die aus EFER liest **und löscht dabei das SVM-Bit** im zurückgegebenen Wert/Register
  - Folge: Ausführung dauert **deutlich länger** (ca. 100x) als eine direkte Ausführung des Lesens aus EFER ohne Hypervisor!
- Zeitunterschied kann von Kernelcode ermittelt werden
  - RDTSC-Instruktion liest internen „Tick“-Zähler

## RDTSC

### Read Time-Stamp Counter

Opcode	Mnemonic	Description
0F 31	RDTSC	Read time-stamp counter into EDX:EAX.

# Virtualisierung erkennen (4)



# Virtualisierung erkennen (5)

## Zeitunterschied kann von Kernelcode ermittelt werden

- RDTSC-Instruktion liest internen „Tick“-Zähler
- Berechnen der Differenz der Zählerstände vor und nach Ausführung des Lesens des EFER-Registers
  - Wenn  $>$  Threshold  $\Rightarrow$  Folgerung: Hypervisor aktiv!

## Und wieder ein Katz- und Maus-Spiel...

- Der Hypervisor kann auch den Zähler manipulieren!
  - Also die Uhr zurückdrehen
- Schafft für den Kernel die Illusion, dass weniger Zeit vergangen ist...



# Fazit

## Rootkits...

- In vielen Farben und Formen
- Versuche des Verbergens immer näher an der Hardware
  - User mode -> kernel mode -> Hypervisor
- Aktuell auch:
  - Einnisten von backdoors in Hardwarekomponenten
    - BIOS/UEFI-rootkits
    - Firmware von embedded Mikrocontrollern
      - Keylogger in Tastaturcontrollern
      - Logs von Schreibvorgängen auf Festplatten
    - Die NSA hat mehr davon ☹...





**“This is your last chance ... After this, there is no turning back.  
You take the blue pill - the story ends, you wake up in your bed,  
and believe whatever you want to believe.**



**You take the red pill, ...  
you stay in Wonderland,  
and I show you,**



**how deep the rabbit-hole goes.”**

*~ Morpheus' Warning To Neo (From The Film; "The Matrix") ~*

# Referenzen

1. SunOS rootkit: CERT Advisory CA-1994-01  
<https://www.cert.org/historical/advisories/CA-1994-01.cfm>
2. Stefan Arbeiter, Matthias Deeg: *Bunte Rechenknechte – Grafikkarten beschleunigen Passwort-Cracker*. In: c't, Ausgabe 06/2009, S. 204
3. [halflife@infonexus.com](mailto:halflife@infonexus.com), "Abuse of the Linux Kernel for Fun and Profit", Phrack 50, <http://phrack.org/issues/50/5.html>
4. Silvio Cesare, "Runtime Kernel kmem Patching"  
<http://www.ouah.org/runtime-kernel-kmem-patching.txt>
5. King, S. T.; Chen, P. M. (2006). "SubVirt: implementing malware with virtual machines", IEEE Symposium on Security and Privacy (S&P'06) 2006
6. BluePill: <http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf>
7. Joseph Kong, "Designing BSD Rootkits. An Introduction to Kernel Hacking," No Starch Press 2007, ISBN-13: 978-1-59327-142-8