Detect Kernel-Mode Rootkits via Real Time Logging & Controlling Memory Access

Igor Korkin, PhD
Independent Researcher
Moscow, Russia

Satoshi Tanda
CrowdStrike, Inc
Vancouver, Canada

2017 CDFSL
The slides are here – www.bit.ly/MemoryMonRWX
We Protect the Computer Memory

Igor Korkin, PhD
Independent Researcher
Moscow, Russia

Satoshi Tanda
CrowdStrike, Inc
Vancouver, Canada

2017 CDFSL
The slides are here – www.bit.ly/MemoryMonRWX
• Igor Korkin, Ph.D.
  • His 5 recent papers are double blind peer reviewed
  • He has spoken at the ADFSL conferences since 2014

• Satoshi Tanda
  • He has 7 years of experience in reverse engineering & Windows internals
  • He spoke at the BlueHat v16, REcon 2011 and 2016
Memory accesses look like driving without rules

It is needed to control the memory accesses
Agenda

• Malware avoids detection: trends & experts’ views

• Intercepting memory access attempts: methods & projects

• The new memory interceptor MemoryMonRWX: idea & prototype

• Demos

• Future plans with IoT & Digital Security
“... malware, or more specifically, a kernel rootkit, can often tamper with kernel memory data, putting the trustworthiness of memory analysis under question”

## What do we have now?

<table>
<thead>
<tr>
<th>Windows security features</th>
<th>What do we have now?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver Signature Enforcement</td>
<td></td>
</tr>
<tr>
<td>PatchGuard (Kernel Patch Protection)</td>
<td></td>
</tr>
</tbody>
</table>

What do we have now?

<table>
<thead>
<tr>
<th>Windows security features</th>
<th>What do we have now?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver Signature Enforcement</td>
<td>3 million of signed malicious binaries(^1)</td>
</tr>
</tbody>
</table>
| PatchGuard (Kernel Patch Protection) | New malware is able to bypass PatchGuard:  
  • ‘Turla’ rootkit\(^2\)  
  • TDL4/TDSS |

---

## Defeat and Protect PatchGuard

<table>
<thead>
<tr>
<th>№</th>
<th>Pre-emptive Actions</th>
<th>Malware actions</th>
<th>Results &amp; Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Rootkit is hiding the process</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Exploit is disabling PatchGuard</td>
<td>Rootkit is hiding the process</td>
</tr>
<tr>
<td>3</td>
<td>Memory protector limits memory access</td>
<td>Exploit is disabling PatchGuard</td>
<td>Rootkit is hiding the process</td>
</tr>
</tbody>
</table>
# Defeat and Protect PatchGuard

<table>
<thead>
<tr>
<th>№</th>
<th>Pre-emptive Actions</th>
<th>Malware actions</th>
<th>Results &amp; Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Rootkit is hiding the process</td>
<td><strong>OS has crashed</strong>&lt;br&gt;(PatchGuard has generated 0x109 BSOD)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exploit is disabling PatchGuard</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rootkit is hiding the process</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Exploit is disabling PatchGuard</td>
<td><strong>OS has been infected</strong>&lt;br&gt;(PatchGuard has been disabled, no BSOD)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rootkit is hiding the process</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Memory protector limits memory access</td>
<td>Exploit is disabling PatchGuard</td>
<td><strong>OS has been protected</strong>&lt;br&gt;(Exploit has failed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rootkit is hiding the process</td>
<td></td>
</tr>
</tbody>
</table>
What malware attacks do we want to monitor & prevent?

Control memory accesses:
- Reading
- Writing
- Executing
Memory Interceptor Requirements

1) All types of memory accesses: read, write, execute

2) Triples for each access:

Source Address → Type of access → Destination Address
Memory Interceptor Requirements

1) All types of memory accesses: read, write, execute

2) Triples for each access:

3) Access only from Source range \(\rightarrow\) Destination range:

4) A kernel-mode driver, which supports Windows 10 x64 and multi-core CPUs
What can we use as a basis for the memory interceptor?

Windows security model protects only the user-mode memory

Memory monitoring methods based on OS & hypervisor facilities

⇒ There is no build-in tools for controlling kernel mode memory
Intercepting memory access: methods & projects

Methods for monitoring access to memory

OS-based
- Hooking Memory Management routines
- Handling Page-Fault Exceptions by IDT

Hypervisor-based
- Handling #PF Exceptions by Hypervisor
- Leveraging Intel EPT technology
### Intercepting memory access: methods & projects

#### Methods for monitoring access to memory

<table>
<thead>
<tr>
<th>Method</th>
<th>Read</th>
<th>Write</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS-based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hooking Memory Management routines</td>
<td>+/+/−</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handling Page-Fault Exceptions by IDT</td>
<td>−/+/+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hypervisor-based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handling #PF Exceptions by Hypervisor</td>
<td>+/−/−</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leveraging Intel EPT technology</td>
<td>−/−/+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Projects

<table>
<thead>
<tr>
<th>Project title, year</th>
<th>Read</th>
<th>Write</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIDER, 2013</td>
<td>+/+/−</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SecVisor, 2007</td>
<td>−/+/+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HyperSleuth, 2010</td>
<td>+/−/−</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CXPInspector, 2013</td>
<td>−/−/+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HyperTap, 2014</td>
<td>−/+/+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAKVUF, 2014</td>
<td>−/−/+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemoryMonRWX, 2017</td>
<td>+/+/+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(The proposed system)
New Advanced Technology: Intel VT-x with Extended Page Tables (EPT)

- EPT Overview
- EPT paging structures
- Applying EPT to monitor & limit memory access

EPT plays the role of traffic lights for memory accesses
Processing memory access: VT-x vs. VT-x with EPT

VT-x without EPT

Access to virtual memory

Paging structures

Guest OS

Guest Physical address:

Hypervisor

Host Physical address:

Hardware

Host Memory
Processing memory access: VT-x vs. VT-x with EPT

VT-x without EPT
- Access to virtual memory
  - Paging structures
- Guest Physical address:
  - Guest OS
  - Hypervisor
  - Hardware
- Host Physical address:
  - Host Memory
  - Host Physical address:

VT-x with EPT
- Access to virtual memory
  - Paging structures
- Guest physical address:
  - Guest OS
  - Hypervisor
  - Hardware
- EPT Paging structures
  - Host Physical address:
  - Host Memory
  - Host Physical address:
Applying EPT features to trap and skip memory access

EPT Paging structures

- EPT Page Directory
  - Entry
  - Entry
- EPT Page Table
  - Access Bits | PFN
  - Access Bits | PFN
  - Access Bits | PFN

Intercept access:
- Read
- Write
- Execute

EPT violation

Hypervisor skips these accesses

Change mapping:
- Substitute page
- Change nothing
Intercept access:
- Read
- Write
- Execute

Change mapping:
- Substitute page
- Change nothing

EPT Paging structures
- EPT Page Directory
- EPT Page Table
- Access Bits | PFN

Applying EPT features to trap and skip memory access

EPT violation
Hypervisor skips these accesses

Entry
Entry

Memory Content

Page walk via EPT pages

1
4

1

2
Hypervisor

EPT violation

3

2

Memory Content

Page walk via EPT pages

1
3
How to apply EPT to monitor access only
- from Source range to Destination range
- and skip all the rest?

We propose the following 5 steps

<table>
<thead>
<tr>
<th>Source Address</th>
<th>Access Type</th>
<th>Destination Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceAddr1</td>
<td>read</td>
<td>DestinationAddr1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Logs this access

Skips all other accesses

We propose the following 5 steps
Step 1. Trapping execution on Source range

Guest OS memory

Source range

Destination range

Others

An access from the source range → destination range

An access from the destination range → others
Step 1. Trapping execution on Source range

These are memory accesses from the source range

Guest OS memory

Source range

Destination range

Others
Step 1. Trapping execution on Source range

These are memory accesses from the source range:

- **Guest OS memory**
  - **Source range**
  - **Destination range**
  - **Others**

**Hypervisor**

**EPT Normal View Structure**

- **SRC.execute = false**
- **SRC.read/write = true**
- **DST.execute = true**
- **DST.read/write = true**
- **OTH.execute = true**
- **OTH.read/write = true**

If ** SRC.execute = false** and ** DST.execute = true**, **Trap & go to Step 2**. If **OTH.execute = true**, **Skip**.
Step 2. Process VM-Exit to separate access to the Destination range

This is the memory access to the Destination range

Guest OS memory

Source range

Destination range

Others
Step 2. Process VM-Exit to separate access to the Destination range

This is the memory access to the Destination range

**Guest OS memory**

- **Source range**
- **Destination range**
- **Others**

**Hypervisor**

**EPT Monitor View Structure**

- **SRC.execute = true** → **Skip**
- **SRC.read/write = true**
- **DST.read/write = false** → **Trap, log & go to Step 3**
- **DST.execute = false** → **Trap & go to Step 5**
- **OTH.execute = false**
- **OTH.read/write = true** → **Skip**

**Memory Accesses**
Step 3. Process VM-Exit, because of access on Destination range

<table>
<thead>
<tr>
<th>Source Address</th>
<th>Access Type</th>
<th>Destination Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceAddr1</td>
<td>read</td>
<td>DestinationAddr1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Step 3. Process VM-Exit, because of access on Destination range

### EPT Monitor View Structure (Modified)

<table>
<thead>
<tr>
<th>Memory Accesses</th>
<th>Source range</th>
<th>Destination range</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC.execute = true</td>
<td>PhysAddrA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRC.read/write = true</td>
<td>PhysAddrB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DST.read/write = true</td>
<td>PhysAddrNew</td>
<td>PhysAddrOriginal</td>
<td></td>
</tr>
<tr>
<td>DST.execute = true</td>
<td>PhysAddrC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OTH.execute = false</td>
<td>PhysAddrD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OTH.read/write = true</td>
<td>PhysAddrE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Substitute Page
- Original Page
- MTF is set
- Trap & go to Step 4
Step 4 (Restore setting). Process VM-Exit, because of MTF

Source range | Destination range | Others
--- | --- | ---

EPT Monitor View Structure (Restored)

<table>
<thead>
<tr>
<th>Memory Accesses</th>
<th>Value</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC.execute</td>
<td>true</td>
<td>PhysAddrA</td>
</tr>
<tr>
<td>SRC.read/write</td>
<td>true</td>
<td>PhysAddrB</td>
</tr>
<tr>
<td>DST.read/write</td>
<td>false</td>
<td>PhysAddrOriginal, PhysAddrNew</td>
</tr>
<tr>
<td>DST.execute</td>
<td>false</td>
<td>PhysAddrC</td>
</tr>
<tr>
<td>OTH.execute</td>
<td>false</td>
<td>PhysAddrD</td>
</tr>
<tr>
<td>OTH.read/write</td>
<td>true</td>
<td>PhysAddrE</td>
</tr>
</tbody>
</table>

MTF is clear

Original Page

Substitute Page

Trap & go to Step 3

Trap & go to Step 5
Step 5. Process VM-Exit, because of execution on Destination range

Hypervisor traps these code executions, but we don’t need to control them.

**EPT Normal View Structure**
- SRC.execute = false
- SRC.read/write = true
- DST.execute = true
- DST.read/write = true
- OTH.execute = true
- OTH.read/write = true

**EPT Monitor View Structure**
- SRC.execute = false
- SRC.read/write = true
- DST.execute = true
- DST.read/write = true
- OTH.execute = true
- OTH.read/write = true

So, hypervisor changes EPT to normal view for trapping an execution on SRC range.

Trap & go to Step 2

Skip
Five steps together

1. INIT
2. EPT Normal View Structure
   - We are waiting for an execution on SRC range
3. EPT Monitor View Structure
   - Any access to destination range will be trapped and logged.
4. Step 3
   - EPT Monitor View with
     - Replaced EPT.DST.PFN
     - EPT.DST.read=true, EPT.DST.write=true
     - MTF is set
5. Step 2
   - Step 5
   - Step 4
   - We restore setting to the original page
   - We protect the destination page via redirecting access to substitute page

We protect the destination page via redirecting access to substitute page.
MemoryMonRWX architecture

Guest Physical Address

MemoryMonRWX

HyperPlatform

Image Load Detector
V2P Map Manager
EPT controller
SRC/DST Range Manager

EPT Paging Data Structures

EPT Normal View Structure
EPT Monitor View Structure

Host Physical Address

The source code is here - http://bit.ly/MemoryMonRWX
MemoryMonRWX architecture

The source code is here - http://bit.ly/MemoryMonRWX
MemoryMonRWX architecture

The source code is here - http://bit.ly/MemoryMonRWX
MemoryMonRWX architecture

The source code is here - http://bit.ly/MemoryMonRWX
MemoryMonRWX architecture

The source code is here - http://bit.ly/MemoryMonRWX
MemoryMonRWX architecture

The source code is here - http://bit.ly/MemoryMonRWX
MemoryMonRWX architecture

MemoryMonRWX

Guest Physical Address

HyperPlatform

Image Load Detector  V2P Map Manager  EPT controller  SRC/DST Range Manager

EPT Paging Data Structures

EPT Normal View Structure  EPT Monitor View Structure

Host Physical Address

The source code is here - http://bit.ly/MemoryMonRWX
MemoryMonRWX is small and fast

Comparison of hypervisors (lines of code)

MemoryMonRWX is made up of less than 12,000 lines of code, which is less than 3% of Xen

0% - the system without hypervisor,
100% – the full system overload
Conclusions

• MemoryMonRWX logs & controls all memory accesses in a real time

• It is a hypervisor, which supports newest Windows 10 x64

• MemoryMonRWX can be used in various tasks:
  • Trace malware activity
  • Protect memory of 3rd party drivers
Acquire Physical Memory & Detect Hidden Software by Raspberry Pi

CaptureGUARD Physical Memory Acquisition Hardware

$7,799.00

“This is an ExpressCard device capable of imaging the physical memory of the computer it's connected to. Creates dump files in the standard WinDD format.”

Acquire Physical Memory & Detect Hidden Software by Raspberry Pi

CaptureGUARD Physical Memory Acquisition Hardware
$7,799.00

“This is an ExpressCard device capable of imaging the physical memory of the computer it's connected to. Creates dump files in the standard WinDD format.”¹

CardBus FPGA dev platform
Xilinx Spartan-3 400²
$295.00

Raspberry Pi 3 Model B
$35.00

$330.00
Lower price with more features

In the USA in upwards of 2.5 million people depend on wireless implantable medical devices, which all can be hijacked remotely.\(^1\)

### Consequences of attacks on implants:

1. Pacemakers by St. Jude Medical Inc. (2016)
   - manipulation of beat rates
   - battery drain

2. OneTouch Ping Insulin Pump by J&J (2016)
   - unauthorized insulin injections

---

Protection of Wireless Implantable Medical Devices

Our Team:
- Veronika Domova
- Software developer, Sweden
- IoT and Industrial Cyber Security
- Igor Korkin, Ph.D.

Our Idea:
1. Input implant’s technical specifications
2. Choose the lightweight crypto cipher
3. Verify the firmware

Vulnerable implant → Protected implant
Thank you!

Igor Korkin  
igor.korkin@gmail.com

Satoshi Tanda  
tanda.sat@gmail.com

The slides, source code and all details are here –  
www.bit.ly/MemoryMonRWX