Detecting and Surviving Intrusions

Exploring New Host-Based Intrusion Detection, Recovery, and Response Approaches

Ronny Chevalier¹,²
Ph.D. Thesis Defense
December 17th, 2019

¹ HP Labs (ronny.chevalier@hp.com)

² CIDRE Team, CentraleSupélec/Inria/CNRS/IRISA (ronny.chevalier@centralesupelec.fr)
Information Security: Overview and Concepts

Information security aims at protecting information assets and mitigating risks.
Information security **aims** at protecting information assets and mitigating risks

Confidentiality
Information Security: Overview and Concepts

Information security aims at protecting information assets and mitigating risks.

- Confidentiality
- Integrity
Information security aims at protecting information assets and mitigating risks.

Confidentiality

Integrity

Availability
Computing Platforms Rely on Preventive Security Mechanisms

Preventive security mechanisms aim at enforcing a security policy on our devices.
Preventive Security is not Sufficient

Examples of preventive security mechanisms

- Access control
- Cryptography
- Firewalls
Preventive Security is not Sufficient

Examples of preventive security mechanisms

• Access control
• Cryptography
• Firewalls

Attackers will eventually bypass our security policy

• (Unknown) vulnerability
• System not updated
• Misconfiguration
Preventive Security is not Sufficient

Examples of preventive security mechanisms

- Access control
- Cryptography
- Firewalls

Computing platforms should not only prevent but **detect** and **survive** intrusions

- System not updated
- Misconfiguration
Focus of This Work: Detecting and Surviving

Preventing Intrusions
Detecting Intrusions
Surviving Intrusions
Focus of This Work: Detecting and Surviving

How computing platforms **detect** and **survive** intrusions?

Preventing Intrusions  Detecting Intrusions  Surviving Intrusions
Computing Platforms Are Made of Multiple Layers

Computing platforms

- Hardware
- BIOS
- Operating System
- Applications

Abstraction layers

- Privileges
  - BIOS
  - Operating System
  - Applications

- More
- Less
Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level

Detecting Intrusions at the Firmware Level

Conclusion and Perspectives
Commodity Operating Systems Can Detect Intrusions

Intrusion Detection Systems (IDSs)\textsuperscript{1}
Knowledge-based vs anomaly-based

IDSs exist in commodity OSs
e.g., Antivirus software share many aspects of host-based IDSs\textsuperscript{2}

---

\textsuperscript{1} Anderson, *Computer Security Threat Monitoring and Surveillance*; Denning, “An Intrusion-Detection Model”.
\textsuperscript{2} Morin and Mé, “Intrusion detection and virology: an analysis of differences, similarities and complementariness”.
Commodity Operating Systems Can Detect Intrusions

**Intrusion Detection Systems (IDSs)**\(^1\)
Knowledge-based vs anomaly-based

**IDSs exist in commodity OSs**
e.g., Antivirus software share many aspects of host-based IDSs\(^2\)

**What can we do after a system has been compromised?**
Eventually we want to patch the system

---


\(^2\) Morin and Mé, “Intrusion detection and virology: an analysis of differences, similarities and complementariness”.
Commodity Operating Systems Can Detect Intrusions

Intrusion Detection Systems (IDSs)\(^1\)
Knowledge-based vs anomaly-based

IDSs exist in commodity OSs
e.g., Antivirus software share many aspects of host-based IDSs\(^2\)

What can we do after a system has been compromised?
Eventually we want to patch the system

What can we do while waiting for the patches?
• Stop the system? \(\rightarrow\) system unavailable for a long time
• Restore to a previous state? \(\rightarrow\) system still vulnerable

---

\(^1\) Anderson, *Computer Security Threat Monitoring and Surveillance*; Denning, "An Intrusion-Detection Model".

\(^2\) Morin and Mé, "Intrusion detection and virology: an analysis of differences, similarities and complementariness".
Commodity Operating Systems Can Detect Intrusions

Intrusion Detection Systems (IDSs)\(^1\)
Knowledge-based vs anomaly-based

IDSs exist in commodity OSs
e.g., Antivirus software share many aspects of host-based IDSs\(^2\).

\section*{Commodity OSs can detect but cannot survive intrusions}

What can we do while waiting for the patches?

- Stop the system? → system unavailable for a long time
- Restore to a previous state? → system still vulnerable

\section*{References}
\(^1\) Anderson, Computer Security Threat Monitoring and Surveillance, Denning, "An Intrusion-Detection Model".
\(^2\) Morin and Mé, "Intrusion detection and virology: an analysis of differences, similarities and complementariness".
Computing Platforms Are Made of Multiple Layers

Computing platforms:
- Hardware
- BIOS
- Operating System
- Applications

Abstraction layers:
- Privileges
- BIOS
- Operating System
- Applications

More

Less
Low-Level Components Are Increasingly Targeted

OS and Application Security Improved Nonetheless
It is more difficult to compromise systems stealthily

Attackers start to focus on lower abstraction layers
Stealthiness and persistence at the BIOS level₃

---

₃ Researchers, LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group.

₄ Regenscheid, Platform Firmware Resiliency Guidelines; Trusted Computing Group, TPM Main, Part 1 Design Principles; Cooper et al., BIOS protection guidelines; UEFI Forum, Unified Extensible Firmware Interface Specification.

₅ HP Inc., HP Sure Start: Automatic Firmware Intrusion Detection and Repair.
Low-Level Components Are Increasingly Targeted

OS and Application Security Improved Nonetheless
It is more difficult to compromise systems stealthily.

Attackers start to focus on lower abstraction layers
Stealthiness and persistence at the BIOS level

Existing solutions
Many at boot time, few at runtime

Hardware
BIOS
Operating System
Applications

Talks and papers about BIOS and firmware attacks

3 Researchers, LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group.
4 Regenscheid, Platform Firmware Resiliency Guidelines; Trusted Computing Group, TPM Main, Part 1 Design Principles; Cooper et al., BIOS protection guidelines; UEFI Forum, Unified Extensible Firmware Interface Specification.
5 HP Inc., HP Sure Start: Automatic Firmware Intrusion Detection and Repair.
Low-Level Components Are Increasingly Targeted

OS and Application Security Improved Nonetheless
It is more difficult to compromise systems stealthily

Attackers start to focus on lower abstraction layers
Stealthiness and persistence at the BIOS level\(^3\)

Existing solutions
Many at boot time\(^4\), few at runtime\(^5\)

---

\(^3\) Researchers, LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group.

\(^4\) Regenscheid, Platform Firmware Resiliency Guidelines; Trusted Computing Group, TPM Main, Part 1 Design Principles; Cooper et al., BIOS protection guidelines; UEFI Forum, Unified Extensible Firmware Interface Specification.

\(^5\) HP Inc., HP Sure Start: Automatic Firmware Intrusion Detection and Repair.
Low-Level Components Are Increasingly Targeted

OS and Application Security Improved Nonetheless
It is more difficult to compromise systems stealthily

Attackers start to focus on lower abstraction layers
Stealthiness and persistence at the BIOS level\(^3\)

Computing platforms **are lacking** generic IDS monitoring the **runtime behavior** of the **BIOS**.

---

\(^{3}\) Researchers, LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group.


\(^{5}\) HP Inc., *HP Sure Start: Automatic Firmware Intrusion Detection and Repair*. 
Thesis and Problems Addressed

Surviving Intrusions at the Operating System Level
How to design an OS so that its services can survive ongoing intrusions while maintaining availability?

Contribution published at RESSI’18\textsuperscript{6} and ACSAC’19\textsuperscript{7}

Detecting Intrusions at the Firmware Level
How to detect intrusions at the firmware level without impacting the quality of service to the rest of the platform?

Contribution published at ACSAC’17\textsuperscript{8}

\textsuperscript{6}Chevalier, Plaquin, and Hiet, “Intrusion Survivability for Commodity Operating Systems and Services: A Work in Progress”.


\textsuperscript{8}Chevalier, Villatel, et al., “Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode”.

Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level
  State of the Art
  Approach and Prototype
  Evaluation
  Conclusion

Detecting Intrusions at the Firmware Level

Conclusion and Perspectives
Running Example

Service: Gitea, a Git Self-Hosting Server
Open source clone of Github (git repositories, bug tracking,...)

Intrusion: Ransomware
It compromises data availability
State of the Art: Intrusion Survivability, Recovery, and Response

**Intrusion Survivability**
Trade-off between the availability and the security risk

---

State of the Art: Intrusion Survivability, Recovery, and Response

**Intrusion Survivability**⁹
Trade-off between the availability and the security risk

**Intrusion Recovery**¹⁰
Restore the system in a safe state when an intrusion is detected

---


13
Intrusion Survivability
Trade-off between the availability and the security risk

Intrusion Recovery
Restore the system in a safe state when an intrusion is detected

Intrusion Response
Limit the impact of an intrusion on the system

---


State of the Art: Limitations we are addressing

Intrusion Survivability
Lack of focus on commodity OSs
State of the Art: Limitations we are addressing

Intrusion Survivability
Lack of focus on commodity OSs

Intrusion Recovery
• The system is still vulnerable and can be reinfected
• Lack of integration between intrusion recovery and response
State of the Art: Limitations we are addressing

Intrusion Survivability
Lack of focus on commodity OSs

Intrusion Recovery
- The system is still vulnerable and can be reinfected
- Lack of integration between intrusion recovery and response

Intrusion Response
Coarse-grained responses and few host-based solutions
State of the Art: Limitations we are addressing

Intrusion Survivability
Lack of focus on *commodity OSs*

Intrusion Recovery
- The system is *still vulnerable* and can be *reinfected*

Commodity OSs are lacking solutions to make them *survive* while *waiting* for the patches to be available

Coarse-grained responses and few host-based solutions
Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level
  State of the Art
  Approach and Prototype
  Evaluation
  Conclusion

Detecting Intrusions at the Firmware Level

Conclusion and Perspectives
Approach Overview

Illustrative Example

Running Example
Gitea infected by some ransomware

When Detected

- Recovery: We restore the service and the encrypted files to a previous state
- Apply restrictions: We remove the ability to write on the file system

Positive Impact
If the ransomware reinfects the service → cannot compromise the files

Degraded Mode
Users can no longer push to repositories → trade-off between availability and security risk
Approach Overview

During the normal operation of the system
Approach Overview

During the normal operation of the system

- Intrusion Detection
- Monitor
- Operating System
  - Service
  - Apache
  - Gitea
- Devices
- Network
- Filesystem

Checkpoint & Log
Approach Overview

During the normal operation of the system

1. Periodic checkpointing
Approach Overview

During the normal operation of the system

Intrusion Detection → Monitor

Operating System
- Service
- Apache
- Gitea

Devices
- Network
- Filesystem

States → Store

Checkpoint & Log
1. Periodic checkpointing
2. Log file write accesses

Checkpoint → Log → Checkpoint

Store → Logs
Approach Overview

How our approach allows the system to survive intrusions after their detection?
Approach Overview

How our approach allows the system to survive intrusions after their detection?
Approach Overview

How our approach allows the system to survive intrusions after their detection?

1. Restore infected objects
2. Restore service
3. Restore files

Intrusion Detection → Monitor → Operating System

Alert

Recovery & Response

Policies
Logs/States

Use

Service n
Apache
Gitea

Devices
Network
Filesystem
Approach Overview

How our approach allows the system to survive intrusions after their detection?

Intrusion Detection → Monitor → Operating System

Alert

Recovery & Response
1. Restore infected objects
2. Withstand reinfection

Restore service
Apply restrictions
Restore files

Logs / States

Remove privileges and decrease resource quotas

Per-service responses to prevent attackers to achieve their goals
Approach Overview

How our approach allows the system to survive intrusions after their detection?

1. Restore infected objects
2. Withstand reinfection
3. Maintain core functions

Potential Degraded Mode
The degraded mode maintains core functions while waiting for patches
Approach Overview

How our approach allows the system to survive intrusions after their detection?

1. Restore infected objects
2. Withstand reinfection
3. Maintain core functions
We select responses that **minimize** the availability impact on the service while **maximizing** the security.

1. Restore infected objects
2. Withstand reinfection
3. Maintain core functions
Cost-Sensitive Response Selection

understand the intrusion → find possible responses → assign costs → select a response
Cost-Sensitive Response Selection

1. understand the intrusion
2. find possible responses
3. assign costs
4. select a response

Costs

very low, low, moderate, high, very high, critical

Malicious behaviors

- Availability violation
- Consume system resources
- Crack passwords
- Mine for cryptocurrency
- Compromise data availability
- Compromise access to information assets
- Command and Control
- Determine C2 server
- Generate C2 domain name(s)
- Receive data from C2 server
- Control malware via remote command
- Update configuration
- ...

Example of malicious behaviors
Cost-Sensitive Response Selection

1. understand the intrusion → find possible responses → assign costs → select a response

Costs
very low, low, moderate, high, very high, critical

Example of a non-exhaustive malicious behavior hierarchy (Source: MAEC of the STIX project)
Cost-Sensitive Response Selection

understand the intrusion → find possible responses → assign costs → select a response

Costs

very low, low, moderate, high, very high, critical

Example of a non-exhaustive malicious behavior hierarchy (Source: MAEC of the STIX project)
Cost-Sensitive Response Selection

1. Understand the intrusion → 2. Find possible responses → 3. Assign costs → 4. Select a response

Malicious behaviors:
- Availability violation: moderate
- Consume system resources: moderate
- Crack passwords: moderate
- Mine for cryptocurrency: moderate
- Compromise data availability: moderate
- Compromise access to information assets: moderate

Command and Control:
- Determine C2 server
- Generate C2 domain name(s)
- Receive data from C2 server
- Control malware via remote command
- Update configuration

Example of a non-exhaustive malicious behavior hierarchy (Source: MAEC of the STIX project)

Costs
- very low, low, moderate, high, very high, critical
Cost-Sensitive Response Selection

1. Understand the intrusion
2. Find possible responses
3. Assign costs
4. Select a response
Cost-Sensitive Response Selection

understand the intrusion → find possible responses → assign costs → select a response

Example of a non-exhaustive per-service response hierarchy

Responses may be provided via the exchange format STIX (e.g., the course of action field)
Cost-Sensitive Response Selection

understand the intrusion → find possible responses → assign costs → select a response
Cost-Sensitive Response Selection

1. Understand the intrusion
2. Find possible responses
3. Assign costs
4. Select a response

Risk Matrix

<table>
<thead>
<tr>
<th>Confidence (Likelihood)</th>
<th>Very low (0–0.2)</th>
<th>Low (0.2–0.4)</th>
<th>Moderate (0.4–0.6)</th>
<th>High (0.6–0.8)</th>
<th>Very high (0.8–1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very likely</td>
<td>L</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Likely</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Probable</td>
<td>L</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>H</td>
</tr>
<tr>
<td>Unlikely</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Very unlikely</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

Example

- **Risk Matrix**

Defined by the administrator/developer
- Defined by threat intelligence
- Defined by the organization

- **Text**

Example

- **Initial Alert**

- **Additional Information**

- **Threat Intelligence**

- **Intrusion Detection**

- **Responses**

- **Malicious Behaviors Costs**

- **Response Costs**

- **Response Efficiency**

- **Malicious Behavior Cost**

- **Cost**

- **Very likely**

- **Likely**

- **Probable**

- **Unlikely**

- **Very unlikely**

- **read-only FS, disable syscall,...**
Cost-Sensitive Response Selection

understand the intrusion → find possible responses → assign costs → select a response

Cost vs Efficiency
It prioritizes efficiency if the risk is high, and cost if the risk is low

\[
\text{max}\ (Risk \times Efficiency + (1 - Risk) \times (1 - Cost))
\]
Cost-Sensitive Response Selection

1. Understand the intrusion → Find possible responses → Assign costs → Select a response

Cost vs Efficiency

\[ \max (Risk \times Efficiency + (1 - Risk) \times (1 - Cost)) \]

We rely on:
- Possible responses
- Malicious behaviors
- Likelihood

We assign:
- Response costs
- Malicious behavior costs
- Risk matrix

We select responses based on:
- Response cost
- Risk
- Response efficiency
## Prototype Implementation for Linux-Based Systems

### Projects Used or Modified

<table>
<thead>
<tr>
<th>Project</th>
<th>What does it do? What is it?</th>
<th>Why do we use/modify it?</th>
<th>Lines of C code added</th>
</tr>
</thead>
<tbody>
<tr>
<td>systemd</td>
<td>system and service manager</td>
<td>Orchestration</td>
<td>2639</td>
</tr>
<tr>
<td>CRIU</td>
<td>checkpoint &amp; restore processes</td>
<td>Restoration</td>
<td>383</td>
</tr>
<tr>
<td>snapper</td>
<td>manage snapshots of file systems</td>
<td>Restoration</td>
<td>0</td>
</tr>
<tr>
<td>Linux kernel</td>
<td></td>
<td>Logging &amp; Responses</td>
<td>460</td>
</tr>
<tr>
<td>cgroups</td>
<td>set of processes bound to a set of limits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seccomp</td>
<td>filter system calls</td>
<td></td>
<td></td>
</tr>
<tr>
<td>namespaces</td>
<td>partition kernel resources</td>
<td></td>
<td></td>
</tr>
<tr>
<td>audit</td>
<td>record security relevant events</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[...]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level
  State of the Art
  Approach and Prototype
  Evaluation
  Conclusion

Detecting Intrusions at the Firmware Level

Conclusion and Perspectives
Evaluation Setup

What Do We Evaluate?

- Responses effectiveness
- Cost-sensitive response selection
- Availability cost and performance impact
- Stability of degraded services
Evaluation Setup

What Do We Evaluate?

- Responses effectiveness
- Cost-sensitive response selection
- Availability cost and performance impact
- Stability of degraded services

Malware and Attacks

- Different types of malicious behaviors (botnet, ransomware, cryptominer,...)

Performance Evaluation Setup

- Various types of services (Apache, nginx, mariadb, beanstalkd, mosquitto, gitea)
- Both synthetic and real-world benchmarks using Phoronix test suite
## Security Evaluation

### Restoration and Responses Effectiveness

<table>
<thead>
<tr>
<th>Attack Scenario</th>
<th>Malicious Behavior</th>
<th>Per-service Response Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux.BitCoinMiner</td>
<td>Mine for cryptocurrency</td>
<td>Ban mining pool IPs</td>
</tr>
<tr>
<td>Linux.BitCoinMiner</td>
<td>Mine for cryptocurrency</td>
<td>Reduce CPU quota</td>
</tr>
<tr>
<td>Linux.Rex.1</td>
<td>Join P2P botnet</td>
<td>Ban bootstrapping IPs</td>
</tr>
<tr>
<td>Hakai</td>
<td>Communicate with C&amp;C</td>
<td>Ban C&amp;C servers’ IPs</td>
</tr>
<tr>
<td>Linux.Encoder.1</td>
<td>Encrypt data</td>
<td>Read-only filesystem</td>
</tr>
<tr>
<td>GoAhead exploit</td>
<td>Open reverse shell</td>
<td>Forbid connect syscall</td>
</tr>
<tr>
<td>GoAhead exploit</td>
<td>Data theft</td>
<td>Render paths inaccessible</td>
</tr>
</tbody>
</table>

### Results

- The service is restored
- The service can withstand the reinfection
Security Evaluation

Cost-Sensitive Response Selection

Goal
Evaluate the impact of the IDS accuracy when selecting responses
→ accurate likelihood (1), inaccurate likelihood (2), false positive (3)

Scenario
Survive ransomware that compromised Gitea

Results

• High risk: read-only filesystem (1, 3)
  • Ransomware failed to reinfect
  • Gitea still usable (can access all repositories, clone them, log in)
• Low risk: read-only paths of important git repositories (2)
  • Ransomware could not encrypt important repositories
  • Gitea still usable (can access important repositories, clone them)
Performance Evaluation

Availability Cost

- less than 300 ms to checkpoint
- less than 325 ms to restore
Performance Evaluation

Availability Cost
- less than 300 ms to checkpoint
- less than 325 ms to restore

Monitoring Cost
- Overhead present only on applications that write to the file system

(a) MB/s score with the Compilebench benchmark (more is better)
Performance Evaluation

**Availability Cost**
- less than 300 ms to checkpoint
- less than 325 ms to restore

**Monitoring Cost**
- Overhead present only on applications that write to the file system
- Small overhead in general (0.6% - 4.5%)

![Diagram showing time (in seconds) to build the Linux kernel (less is better)](image)

(b) Time (in seconds) to build the Linux kernel (less is better)
Performance Evaluation

Availability Cost
- less than 300 ms to checkpoint
- less than 325 ms to restore

Monitoring Cost
- Overhead present only on applications that write to the file system
- Small overhead in general (0.6% - 4.5%)
- Worst case (28.7% overhead): writing small files asynchronously in burst

(c) Time (in seconds) to extract the archive (.tar.gz) of the Linux kernel source code (less is better)
Performance Evaluation

Availability Cost

- less than 300 ms to checkpoint
- less than 325 ms to restore

Monitoring Cost

- Overhead present only on applications that write to the file system
- Small overhead in general (0.6 % - 4.5 %)
- Worst case (28.7 % overhead): writing small files asynchronously in burst
- e.g., SHELF\textsuperscript{12} has 8 % and 67 % overhead

\textsuperscript{12}Xiong, Jia, and P. Liu, “SHELF: Preserving Business Continuity and Availability in an Intrusion Recovery System”.
Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level
  State of the Art
  Approach and Prototype
  Evaluation
  Conclusion

Detecting Intrusions at the Firmware Level

Conclusion and Perspectives
Scientific Contributions and Future Work

What were the challenges?

- The system survives while waiting for the patches
- Realistic use cases
- Maintain availability while maximizing security

Future work

- Checkpointing limitations (e.g., with CRIU)
- Models input

RESSI’18
Ronny Chevalier, David Plaquin, and Guillaume Hiet. “Intrusion Survivability for Commodity Operating Systems and Services: A Work in Progress”. May 2018

ACSAC’19
Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level

Detecting Intrusions at the Firmware Level
  Background, Use Case, and State of the Art
  Approach and Prototype
  Evaluation
  Conclusion

Conclusion and Perspectives
Computers rely on firmware

Where can we find firmware?
Mother boards (e.g., BIOS), hard disks, network cards,…

Here, we focus on BIOS/UEFI-compliant firmware

What is it?
• Stored in a flash
• Low-level software
• Tightly linked to hardware

Boot time vs Runtime
• Early execution and configuration
• Highly privileged runtime software
What is the problem?

BIOSs are often written in unsafe languages (i.e., C & assembly)
Memory safety errors (e.g., use after free or buffer overflow)

BIOSs are not exempt from vulnerabilities\(^\text{13}\)

Why compromise a BIOS?
- Malware can be hard to detect (stealth)
- Malware can be persistent (survives even if the HDD/SSD is changed) and costly to remove

What do we want?
- Boot time integrity
- Runtime integrity → some platforms are rarely rebooted

\(^{13}\) Kallenberg et al., “Defeating Signed BIOS Enforcement”; Bazhaniuk et al., “A new class of vulnerabilities in SMI handlers”; Researchers, LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group.
What are the currently used solutions?

Boot time

- Signed updates
- Signature verification before executing
- Measurements and reporting to a TPM chip
- Immutable hardware root of trust
What are the currently used solutions?

**Boot time**
- Signed updates
- Signature verification before executing
- Measurements and reporting to a TPM chip
- Immutable hardware root of trust

**Runtime**
Isolation of critical services available while the OS is running
→ our focus is with the System Management Mode (SMM)
Introducing the System Management Mode (SMM)

Highly privileged execution mode for x86 processors

Runtime services
BIOS update, power management, UEFI variables handling, etc.

How to enter the SMM?

• Trigger a System Management Interrupt (SMI) → needs kernel privileges
• SMI's code & data are stored in a protected memory region: System Management RAM (SMRAM)

BIOS code is not exempt from vulnerabilities affecting SMM

Why is it interesting for an attacker?

• Only mode that can write to the flash containing the BIOS
• Arbitrary code execution in SMM gives full control of the platform

---

14 Bazhaniuk et al., “A new class of vulnerabilities in SMI handlers”; Bulygin, Bazhaniuk, et al., “BARing the System: New vulnerabilities in Coreboot & UEFI based systems”; Pujos, SMM unchecked pointer vulnerability; Researchers, LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group.
State of the Art: Runtime Intrusion Detection for Low-Level Components

Few solutions were designed to monitor the SMM at runtime

Snapshot-Based Approaches\textsuperscript{15}

- Periodic snapshot of the target’s state
- Limitations: transient attacks

Event-Based Approaches\textsuperscript{16}

- Observe events generated by the target
- Limitations: performance issues, lack of flexibility, or semantic gap

\textsuperscript{15}Petroni et al., “Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor”; Bulygin and Samyde, “Chipset based approach to detect virtualization malware”.

State of the Art: Runtime Intrusion Detection for Low-Level Components

Few solutions were designed to monitor the SMM at runtime.

Snapshot-Based Approaches\textsuperscript{15}

- Periodic snapshot of the target’s state

• Observe events generated by the target

• Limitations: performance issues, lack of flexibility, or semantic gap

How computing platforms can be designed to detect intrusions modifying the runtime behavior of the SMM?

\textsuperscript{15} Petroni et al., “Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor”; Bulygin and Samyde, “Chipset based approach to detect virtualization malware”.


33
Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level

Detecting Intrusions at the Firmware Level
  Background, Use Case, and State of the Art
  Approach and Prototype
  Evaluation
  Conclusion

Conclusion and Perspectives
Our objective

Our goal is to detect attacks that modify the expected behavior of the SMM by monitoring its behavior at runtime.

Such a goal raises the following questions:

- How to ensure the integrity of the monitor?
- How to define a correct behavior?
- How to monitor?
Approach overview

- Co-processor
- RAM
- Processor
- RAM
- Target
- Monitor
- Unidirectional FIFO
- Co-processor
- Processor

Expected target behavior

- SMM code

How to ensure the integrity of the monitor?

- Semantic gap?
- How to monitor?
- Bridging the semantic gap

- LLVM-based Compiler
- SMM source code
- BIOS source code

How to define a correct behavior?
Approach overview

How to ensure the integrity of the monitor?

How to monitor?

Semantic gap?
Approach overview

How to ensure the integrity of the monitor?

Semantic gap?

How to define a correct behavior?
Approach overview

How to ensure the integrity of the monitor?

How to monitor?
bridging the semantic gap
Approach overview

How to ensure the integrity of the monitor?

How to monitor?
bridging the semantic gap

How to define a correct behavior?
How to define a correct behavior?

Our use case: SMM code

- Written in unsafe languages (i.e., C & assembly)
  → Such languages are often targeted by attacks hijacking the control flow
- Tightly coupled to hardware
  → Its behavior rely on hardware configuration registers

Control Flow Graph (CFG)
Define the control flow that the software is expected to follow
→ Control Flow Integrity (CFI)

Invariants on CPU registers
Define rules that registers are expected to satisfy
→ CPU registers integrity
How to define a correct behavior?

Control Flow Integrity (CFI): principle

Example

```c
void auth(int a, int b) {
  char buffer[512];

  [...vuln...]
  verification(buffer);
}
void verification(char *input) {
  if (strcmp(input, "secret") == 0)
    authenticated();
  else
    non_authenticated();
}
```

Simplified graph
How to define a correct behavior?

Control Flow Integrity (CFI): principle

Example

```c
void auth(int a, int b) {
    char buffer[512];

    [...vuln...]
    verification(buffer);
}

void verification(char *input) {
    if (strcmp(input, "secret") == 0)
        authenticated();
    else
        non_authenticated();
}
```

Simplified graph

```
Authenticated

Non authenticated

verIFICATION

auth
```

38
How to define a correct behavior?

Control Flow Integrity (CFI): principle

Example

```c
void auth(int a, int b) {
    char buffer[512];

    // [...vuln...]
    verification(buffer);
}

void verification(char *input) {
    if (strcmp(input, "secret") == 0)
        authenticated();
    else
        nonAuthenticated();
}
```

Goal: constrain the execution path to follow a control-flow graph (CFG)
How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

We focus on indirect branches integrity

Type-based verification
Ensures the integrity of indirect calls

typedef struct SomeStruct {
    [...]  
    char (*foo)(int);
} SomeStruct;

int bar(SomeStruct *s) {
    char c;
    [...]  

c = s->foo(31);
    [...]  
}
How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

We focus on indirect branches integrity

Type-based verification
Ensures the integrity of indirect calls

typedef struct SomeStruct {
    [...] 
    char (*foo)(int);
} SomeStruct;

int bar(SomeStruct *s) {
    char c;
    [...] 

    c = s->foo(31);
    [...] 
}
How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification
We focus on indirect branches integrity

Type-based verification
Ensures the integrity of indirect calls

typedef struct SomeStruct {
    [...] char (*foo)(int);
} SomeStruct;

int bar(SomeStruct *s) {
    char c;
    [...] c = s->foo(31);
    [...] }

How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

We focus on indirect branches integrity

Type-based verification
Ensures the integrity of indirect calls

typedef struct SomeStruct {
    [...]  
    char (*foo)(int);
} SomeStruct;

int bar(SomeStruct *s) {
    char c;
    [...]  

c = s->foo(31);
    [...]  
}
How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

We focus on indirect branches integrity

Type-based verification

Ensures the integrity of indirect calls

typedef struct SomeStruct {
    [...]  
    char (*foo)(int);
} SomeStruct;

int bar(SomeStruct *s) {
    char c;
    [...]  
    c = s->foo(31); /* Call Site ID = 1561 */
    [...]  
}
How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

We focus on indirect branches integrity

Type-based verification
Ensures the integrity of indirect calls

typedef struct SomeStruct {
    [...] char (*foo)(int);
} SomeStruct;
int bar(SomeStruct *s) {
    char c;
    [...] [SendMessage(1561, s->foo)]
    c = s->foo(31); /* Call Site ID = 1561 */
    [...]}
How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

We focus on indirect branches integrity

Type-based verification
Ensures the integrity of indirect calls

typedef struct SomeStruct {
    [...] char (*foo)(int);
} SomeStruct;

int bar(SomeStruct *s) {
    char c;
    [...]}

[SendMessage(1561, s->foo)]
c = s->foo(31); /* Call Site ID = 1561 */ [...]
}
How to define a correct behavior?

Control Flow Integrity (CFI): shadow call stack

Shadow call stack

Ensures integrity of the return address on the stack
How to define a correct behavior?

**CPU registers integrity**

SMM code is tightly coupled to hardware

- Generic detection methods (e.g., CFI) are not aware of hardware specificities
- Adhoc detection methods are needed

**Some interesting registers for an attacker**

- **SMBASE**: Defines the SMM entry point
- **CR3**: Physical address of the page directory

→ Their value is stored in memory and is not supposed to change at runtime

**How to protect such registers?**

- Send the expected values at boot time
- Send messages at runtime containing these values to detect any discrepancy
How to monitor?

Communication channel constraints

Security constraints

• Message integrity
• Chronological order
• Exclusive access

Performance constraints

• Acceptable latency of an SMI as defined by Intel BIOS Test Suite: 150 µs
• More than 150 µs per SMI handler leads to degradation of performance or user experience
How to monitor?

Communication channel design

Additional hardware component

• Chronological order
  → FIFO (queue)

• Message integrity
  → Restricted FIFO

• Exclusive access
  → Check if CPU is in SMM (SMIACK# signal)

• Performance
  → Use a low latency interconnect
Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level

Detecting Intrusions at the Firmware Level
  Background, Use Case, and State of the Art
  Approach and Prototype
  Evaluation
  Conclusion

Conclusion and Perspectives
Our experimental setup

Our prototype is implemented in a simulated and emulated environment

**SMM code implementations used**

- EDK2: foundation of many BIOSes (Apple, HP, Intel,...)  
  → UEFI Variables SMI handlers
- coreboot: perform hardware initialization (used on some Chromebooks)  
  → Hardware-specific SMI handlers

**We want to emulate SMM environment and features**

QEMU emulator for security evaluation

**We want to simulate accurately the performance impact**

gem5 simulator for performance evaluation
We simulated attacks that exploited vulnerabilities similar to those found in real-world BIOSes.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Attack Target</th>
<th>Security Advisories</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer overflow</td>
<td>Return address</td>
<td>CVE-2013-3582</td>
<td>Yes</td>
</tr>
<tr>
<td>Arbitrary write</td>
<td>Function pointer</td>
<td>CVE-2016-8103</td>
<td>Yes</td>
</tr>
<tr>
<td>Arbitrary write</td>
<td>SMBASE</td>
<td>LEN-4710</td>
<td>Yes</td>
</tr>
<tr>
<td>Insecure call</td>
<td>Function pointer</td>
<td>LEN-8324</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Performance evaluation

Running time overhead for SMI handlers

- Under the 150 microseconds limit defined by Intel
- Most of the communication overhead is due to the shadow call stack

**Graphs**

**EDK2**

- SetVariable
- GetVariable
- QueryVariableInfo
- GetNextVariableName

**coreboot**

- i82801gx
- i82801gx TCO
- i82801gx PM1
- AMD Agesa
- AMD Agesa GPE
Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level

Detecting Intrusions at the Firmware Level
  
  Background, Use Case, and State of the Art

  Approach and Prototype

  Evaluation

  Conclusion

Conclusion and Perspectives
Scientific Contributions and Future Work

What were the challenges?

- Detect privileged attacks against runtime firmware
- Do not impact quality of service (< 150 µs Intel threshold)
- Simulation-based prototype implementation

Future work

- Hardware-based prototype
- Intel CET

ACSAC’17

Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level

Detecting Intrusions at the Firmware Level

Conclusion and Perspectives
Conclusion

Computing platforms should not only prevent but detect and survive intrusions

Surviving Intrusions at the Operating System Level
- The system survives while waiting for the patches
- Maintains availability while maximizing security
- Linux-based prototype implementation

Detecting Intrusions at the Firmware Level
- The platform detects attacks targeting runtime firmware
- Maintains quality of service while detecting privileged attacks
- Simulation-based prototype with the SMM as a use case
Perspectives

How to adapt the system so that we can deactivate our responses?
- Can we automatically find the vulnerabilities exploited by the attackers?
- How can we automatically patch them?

How to survive intrusions at the firmware level?
- How to recover the SMRAM and the SMI handlers’ state?
- How to apply restrictions per-SMI handler?
Thanks for your attention!
Questions?

Computing platforms should not only prevent but detect and survive intrusions

Surviving Intrusions at the Operating System Level
- The system survives while waiting for the patches
- Maintains availability while maximizing security
- Linux-based prototype implementation

Detecting Intrusions at the Firmware Level
- The platform detects attacks targeting runtime firmware
- Maintains quality of service while detecting privileged attacks
- Simulation-based prototype with the SMM as a use case

ACSAC’19, RESSI’18

ACSAC’17
Backup: Surviving
Prototype Implementation for Linux-Based Systems

Architecture Overview

User land

Isolated Components

Monitored Services

Linux Kernel

Protocols Use

Responses Selection

Logs

States

systemd

CRIU

Policies

Logs

States

snapper

Systemd

Checkpoints

Log

Per-service Privileges & Quotas

MAC

Resources, Files, Devices, Network...

Isolate

Log

Checkpoints

Configure

Send Alert

Monitor

Store

Use

Response

Store & Fetch

Trigger checkpoint

Manage

Use

Checkpoints

Configure

1 2 n

Resources, Files, Devices, Network,...

dynamic policy

static policy

54
Models That Depends on Vulnerabilities

Various approaches rely on knowledge about vulnerabilities

Issues

- It requires to continuously check for the presence of vulnerabilities
- There are unknown vulnerabilities that can be exploited

“Exploits and their underlying vulnerabilities have a rather long average life expectancy (6.9 years)”

“For a given stockpile of zero-day vulnerabilities, after a year, approximately 5.7 percent have been discovered by an outside entity”.

---


Stability of the Degraded Services

Core Functions
Our policies help to define the privileges that should never be removed

None of The Services We Tested Crashed
Apache, nginx, mariadb, beanstalkd, mosquitto, gitea

• They performed error checking
• They logged errors but did not crash

Generalization

• Such a degradation should work with other services that perform error checking
• Static analysis tools highlight missing error checks\textsuperscript{19}

\textsuperscript{19}CERT C Coding Standard, ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy; CERT C Coding Standard, EXP12-C. Do not ignore values returned by functions.
## Storage Cost Overhead

### Checkpointing Services Requires Storage Space

<table>
<thead>
<tr>
<th>Service</th>
<th>Checkpoint Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>26.2 MiB</td>
</tr>
<tr>
<td>nginx</td>
<td>7.5 MiB</td>
</tr>
<tr>
<td>mariadb</td>
<td>136.0 MiB</td>
</tr>
<tr>
<td>beanstalkd</td>
<td>130.1 KiB</td>
</tr>
</tbody>
</table>

Memory pages took at least 95.3% of the size of their checkpoint.
## Availability Cost Details

### Checkpoint

<table>
<thead>
<tr>
<th>Checkpoint Operation</th>
<th>Mean</th>
<th>Standard Deviation</th>
<th>Standard Error of the Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service-independent operations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialize (µs)</td>
<td>643.20</td>
<td>90.75</td>
<td>14.35</td>
</tr>
<tr>
<td>Checkpoint service metadata (µs)</td>
<td>51.47</td>
<td>8.45</td>
<td>1.33</td>
</tr>
<tr>
<td>Snapshot file system (ms)</td>
<td>98.95</td>
<td>1.38</td>
<td>2.19</td>
</tr>
<tr>
<td><strong>Checkpoint processes (CRIU)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>httpd (ms)</td>
<td>199.24</td>
<td>11.05</td>
<td>3.49</td>
</tr>
<tr>
<td>nginx (ms)</td>
<td>51.59</td>
<td>3.99</td>
<td>1.26</td>
</tr>
<tr>
<td>mariadb (ms)</td>
<td>171.77</td>
<td>8.52</td>
<td>2.69</td>
</tr>
<tr>
<td>beanstalkd (ms)</td>
<td>16.25</td>
<td>1.37</td>
<td>0.43</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>httpd (ms)</td>
<td>298.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nginx (ms)</td>
<td>151.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mariadb (ms)</td>
<td>271.41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beanstalkd (ms)</td>
<td>115.89</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time to perform the checkpoint operations of a service
## Availability Cost Details

### Restore

<table>
<thead>
<tr>
<th>Restore Operation</th>
<th>Mean</th>
<th>Standard deviation</th>
<th>Standard error of the mean</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kill processes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>httpd</td>
<td>16.39</td>
<td>2.52</td>
<td>1.13</td>
</tr>
<tr>
<td>nginx</td>
<td>19.24</td>
<td>3.69</td>
<td>1.65</td>
</tr>
<tr>
<td>mariadb</td>
<td>28.48</td>
<td>2.16</td>
<td>0.97</td>
</tr>
<tr>
<td>beanstalkd</td>
<td>10.85</td>
<td>1.19</td>
<td>0.53</td>
</tr>
<tr>
<td>nginx</td>
<td>59.88</td>
<td>4.88</td>
<td>2.18</td>
</tr>
<tr>
<td>mariadb</td>
<td>147.07</td>
<td>2.59</td>
<td>1.16</td>
</tr>
<tr>
<td>beanstalkd</td>
<td>36.63</td>
<td>2.87</td>
<td>1.28</td>
</tr>
<tr>
<td><strong>Service-independent operations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialize</td>
<td>209.40</td>
<td>32.07</td>
<td>7.17</td>
</tr>
<tr>
<td>Compare Snapshots</td>
<td>148.23</td>
<td>32.01</td>
<td>7.16</td>
</tr>
<tr>
<td>Restore service metadata</td>
<td>212.75</td>
<td>36.23</td>
<td>8.10</td>
</tr>
<tr>
<td><strong>Restore processes (CRIU)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>httpd</td>
<td>132.42</td>
<td>6.09</td>
<td>2.72</td>
</tr>
<tr>
<td>nginx</td>
<td>59.88</td>
<td>4.88</td>
<td>2.18</td>
</tr>
<tr>
<td>mariadb</td>
<td>147.07</td>
<td>2.59</td>
<td>1.16</td>
</tr>
<tr>
<td>beanstalkd</td>
<td>36.63</td>
<td>2.87</td>
<td>1.28</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>httpd</td>
<td>299.29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nginx</td>
<td>227.79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mariadb</td>
<td>324.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beanstalkd</td>
<td>196.16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Time to perform the restore operations of a service**
Backup: Detecting
Security evaluation

Number and size of equivalence classes for the type-based verification

Our analysis with EDK II gave:

- 158 equivalence classes of size 1,
- 24 of size 2,
- 42 of size 3,
- 2 of size 5,
- 1 of size 9,
- and 1 of size 13.
Performance evaluation

Co-processor time to process messages

<table>
<thead>
<tr>
<th>Function</th>
<th>Intel i82801gx</th>
<th>AMD Agesa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Variable</td>
<td>230</td>
<td>18</td>
</tr>
<tr>
<td>Get Variable</td>
<td>152</td>
<td>7</td>
</tr>
<tr>
<td>Query Variable</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>Query VariableInfo</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>GVN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Performance evaluation

#### Number of packets sent due to the instrumentation

<table>
<thead>
<tr>
<th>SMI Handler</th>
<th>Shadow stack (SS)</th>
<th>Indirect call (IC)</th>
<th>SMBASE &amp; CR3 (SC)</th>
<th>Total number of packets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EDK II</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VariableSmm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetVariable</td>
<td>384</td>
<td>4</td>
<td>4</td>
<td>392</td>
</tr>
<tr>
<td>GetVariable</td>
<td>240</td>
<td>4</td>
<td>4</td>
<td>248</td>
</tr>
<tr>
<td>QueryVariableInfo</td>
<td>299</td>
<td>4</td>
<td>4</td>
<td>208</td>
</tr>
<tr>
<td>GetNextVariableName</td>
<td>212</td>
<td>4</td>
<td>4</td>
<td>220</td>
</tr>
<tr>
<td><strong>coreboot</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel i82801gx APMC/TCO/PM1</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>AMD Agesa Hudson APMC/GPE</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

**Figure 1:** Number of packets sent during one SMI handler (Number of packets per message type: SS=2, IC=2, SC=4)
The target sends messages to describe its own behavior

**Key point**
The attacker must alter the control flow (i.e., behavior) in order to forge messages

→ The attacker cannot send messages in lieu of the target **without first being detected**

**What are the attacker’s capabilities before the attack?**
Complete control over the OS (e.g., can trigger as many SMI as necessary)

**What kind of attack?**
Runtime attack by triggering memory corruption issues in an SMI handler (e.g., ROP)
Related work

Snapshot-based approach

Copilot [Petroni et al., “Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor”]

- Processor
- Main CPU
- PCI Card
- Co-pilot
- System Bus
- Main Memory
- DMA

- Flexible
- Cannot monitor SMM code
- Semantic gap
- Transient attacks
- Additional hardware

DeepWatch [Bulygin and Samyde, “Chipset based approach to detect virtualization malware”]

- Processor
- Main CPU
- Chipset
- DeepWatch
- Main Memory
- DMA

- Flexible
- Can monitor SMM code
- No additional hardware
Related work

**Event-driven approach**


- Flexible
- Could monitor SMM code
- Semantic gap
- Detect transient attacks
- Additional hardware

MGuard [Z. Liu et al., “CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM”]

- Flexible
- Can monitor SMM code
- Semantic gap
- Detect transient attacks
- Requires FB DIMM Memory
Related work

Hardware-based CFI approach

Future CFI technology in Intel processors? [Intel Corporation, “Control-flow Enforcement Technology Specification”]

Advantages
✓ Can monitor SMM code
✓ Efficient
✓ No semantic gap
✓ Detect transient attacks

Limitations
✗ Precision loss
✗ Not flexible (i.e., one detection method)
✗ Requires to modify the processor
Communication channel

Mailboxes
High latency

Need to design an intermediate hardware component
Restricted FIFO to store temporarily messages

PCIe
- Designed to maximize I/O throughput
- Not suited to send many small packets (coarse-grained interaction)

CPU Interconnects (QPI, HyperTransport)
- Designed to minimize latency
- Suited to exchange small packets (fine-grained interaction)
SMBASE integrity

Save State Area
The processor stores its context at SMI entry and restores it at SMI exit

SMBASE
Location of the SMRAM in RAM, stored in the save state area

What if an attacker overwrites the SMBASE?
• Need to exit the SMI and retrigger a SMI
• The new SMBASE is used
• Arbitrary code execution in SMM

Solution
• At boot time: Send the expected value to the monitor
• At runtime: Send the current value at each SMI exit
Performance evaluation

Firmware size

Size of firmware code is limited by the amount of flash (e.g., 8MB or 16MB)

EDK2

- +17 408 bytes in firmware code
- +0.6% increase in size for the compressed firmware

coreboot

- Could not compile the whole firmware with our LLVM toolchain (clang not supported by coreboot)
- AMD Agesa Hudson SMI handlers: +568 bytes
- Intel i82801gx SMI handlers: +3448 bytes
Code integrity at runtime

Multiple options

Page tables
Recent BIOSes can enable write protection for SMM code pages\(^{20}\)

HP Sure Start Gen3\(^{21}\)
Detects attempts to modify SMM code
Notifies and takes actions per a predefined policy

\(^{20}\)https://lists.01.org/pipermail/edk2-devel/2016-November/004185.html


CERT C Coding Standard. ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy. Aug. 30, 2019. URL: https://wiki.sei.cmu.edu/confluence/display/c/ERR00-C.+Adopt+and+implement+a+consistent+and+comprehensive+error-handling+policy.


References


## Images Credits

### URLs provided

<table>
<thead>
<tr>
<th>Image</th>
<th>Name</th>
<th>Author</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://example.com/rollback" alt="Rollback" /></td>
<td>Rollback</td>
<td>Gyorgy Hunor-Arpad</td>
<td>CC BY 3.0 US</td>
</tr>
<tr>
<td><img src="https://example.com/application" alt="Application" /></td>
<td>Application</td>
<td>Christopher</td>
<td>CC BY 3.0 US</td>
</tr>
<tr>
<td><img src="https://example.com/chip_settings" alt="Chip Settings" /></td>
<td>Chip Settings</td>
<td>Luis Rodrigues</td>
<td>CC BY 3.0 US</td>
</tr>
<tr>
<td><img src="https://example.com/gear" alt="Gear" /></td>
<td>Gear</td>
<td>Jonathan Higley</td>
<td>CC0 1.0 Universal</td>
</tr>
<tr>
<td><img src="https://example.com/harddrive" alt="Harddrive" /></td>
<td>Harddrive</td>
<td>Creaticca Creative Agency</td>
<td>CC BY 3.0 US</td>
</tr>
<tr>
<td><img src="https://example.com/microchip" alt="Microchip" /></td>
<td>Microchip</td>
<td>Creative Stall</td>
<td>CC BY 3.0 US</td>
</tr>
<tr>
<td><img src="https://example.com/research" alt="Research" /></td>
<td>Research</td>
<td>Gregor Cresnar</td>
<td>CC BY 3.0 US</td>
</tr>
</tbody>
</table>