Detecting and Surviving Intrusions

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

Ronny Chevalier [1,2]

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: Overview and Concepts

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: Overview and Concepts

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

Confidentiality

Integrity

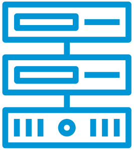
Availability

Computing Platforms Rely on Preventive Security Mechanisms

Preventive security mechanisms aim at enforcing a **security policy** on our devices

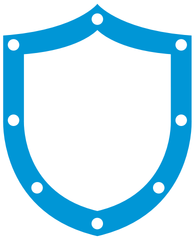
Laptop

Printer



Server

Preventive Security is not Sufficient



Examples of preventive security mechanisms

- Access control
- Cryptography
- Firewalls

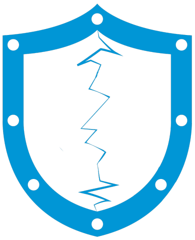
Preventive Security is not Sufficient

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

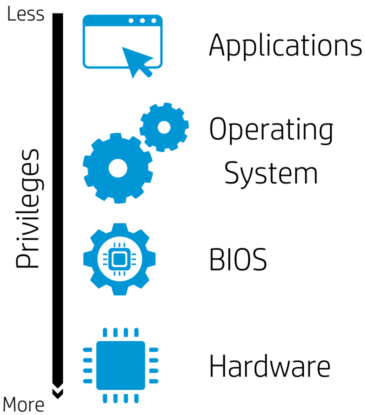
Detecting Intrusions

Surviving Intrusions

Computing Platforms Are Made of Multiple Layers

Computing platforms

Abstraction layers



Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level

Detecting Intrusions at the Firmware Level

Conclusion and Perspectives

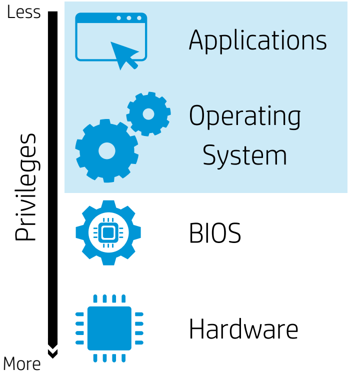
Commodity Operating Systems Can Detect Intrusions

Intrusion Detection Systems (IDSs)[1]

Knowledge-based vs anomaly-based

IDSs exist in commodity OSs

e.g., Antivirus software share many aspects of host-based IDSs[2]



[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]

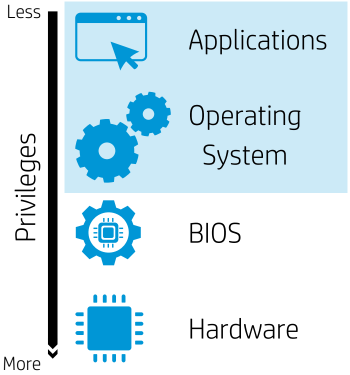
Knowledge-based vs anomaly-based

IDSs exist in commodity OSs

e.g., Antivirus software share many aspects of host-based IDSs[2]

What can we do after a system has been compromised?

Eventually we want to patch the system



[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

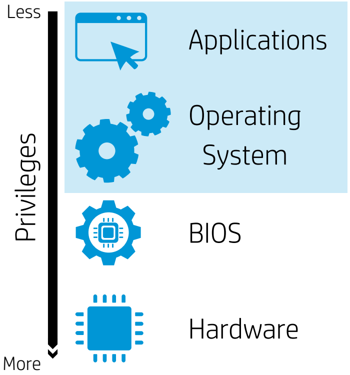
e.g., Antivirus software share many aspects of host-based IDSs[2]

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? → system unavailable for a long time
- Restore to a previous state? → 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".

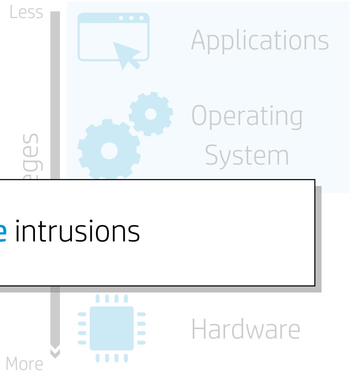
Commodity Operating Systems Can Detect Intrusions

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]



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

---

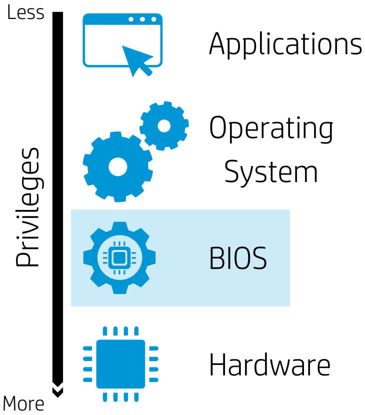
[1] Anderson, *Computer Security Threat Monitoring and Surveillance*; Denning, "An Intrusion-Detection Model".

[2] Morin and Mé, "Intrusion detection and virology: an analysis of differences, similarities and complementariness".

Computing Platforms Are Made of Multiple Layers

Computing platforms

Abstraction layers



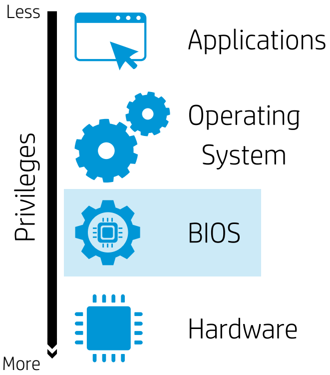
Low-Level Components Are Increasingly Targeted

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]



[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

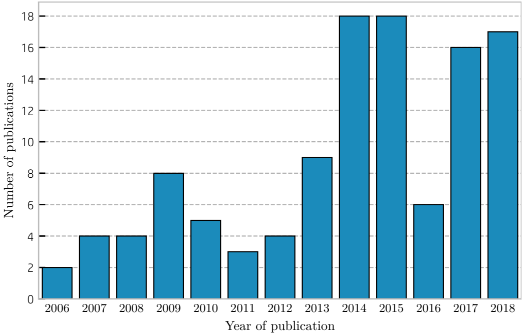
OS and Application Security Improved Nonetheless

It is more difficult to compromise systems stealthily

Less

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

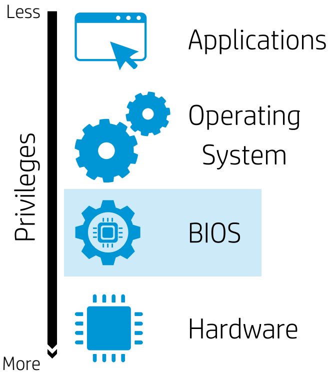
It is more difficult to compromise systems stealthily

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.*

[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*.

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[6] and ACSAC’19[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[8]

---

[6] Chevalier, Plaquin, and Hiet, “Intrusion Survivability for Commodity Operating Systems and Services: A Work in Progress”.

[7] Chevalier, Plaquin, Dalton, et al., “Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems”.

[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

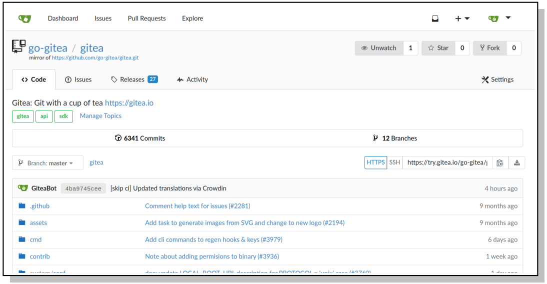
Surviving Intrusions at the Operating System Level

- 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[9]

Trade-off between the availability and the security risk

[9] Knight and Strunk, “Achieving Critical System Survivability Through Software Architectures”; Ellison et al., *Survivable Network Systems: An emerging discipline*.

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

[9] Knight and Strunk, “Achieving Critical System Survivability Through Software Architectures”; Ellison et al., *Survivable Network Systems: An emerging discipline*.

[10] Goel et al., “The Taser Intrusion Recovery System”; Xiong, Jia, and P. Liu, “SHELF: Preserving Business Continuity and Availability in an Intrusion Recovery System”.

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

Intrusion Response[11]

Limit the impact of an intrusion on the system

[9] Knight and Strunk, “Achieving Critical System Survivability Through Software Architectures”; Ellison et al., *Survivable Network Systems: An emerging discipline*.

[10] Goel et al., “The Taser Intrusion Recovery System”; Xiong, Jia, and P. Liu, “SHELF: Preserving Business Continuity and Availability in an Intrusion Recovery System”.

[11] Balepin et al., “Using Specification-Based Intrusion Detection for Automated Response”; Shameli-Sendi, Cheriet, and Hamou-Lhadj, “Taxonomy of Intrusion Risk Assessment and Response 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

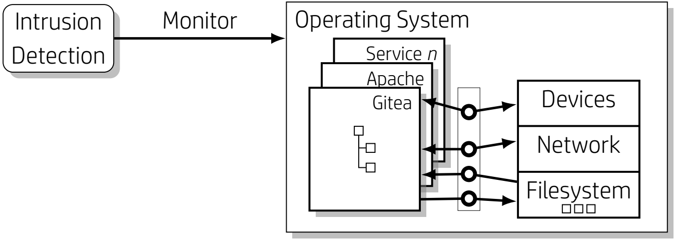
If the ransomware reinfects the service $\rightarrow$ cannot compromise the files

Degraded Mode

Users can no longer push to repositories $\rightarrow$ trade-off between availability and security risk

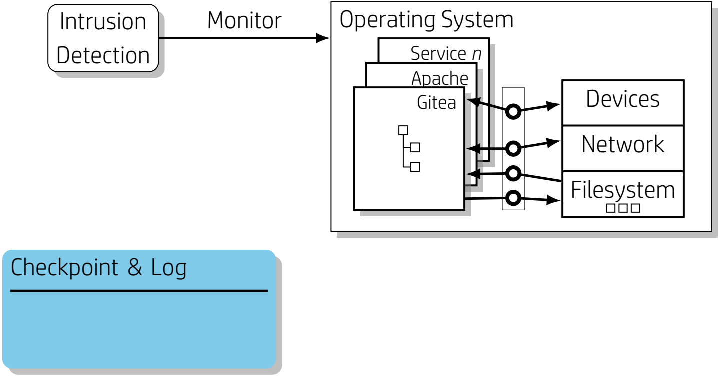
Approach Overview

During the normal operation of the system



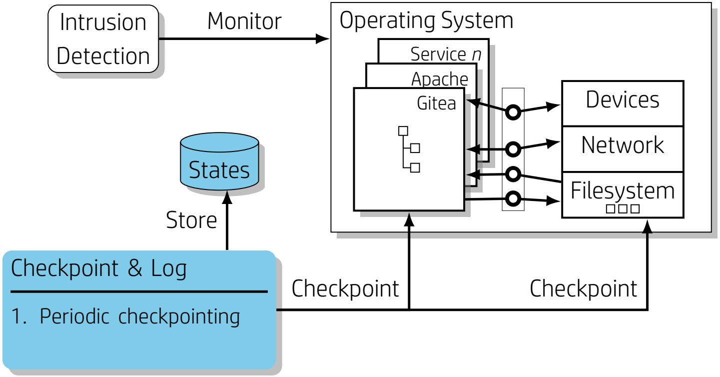
Approach Overview

During the normal operation of the system



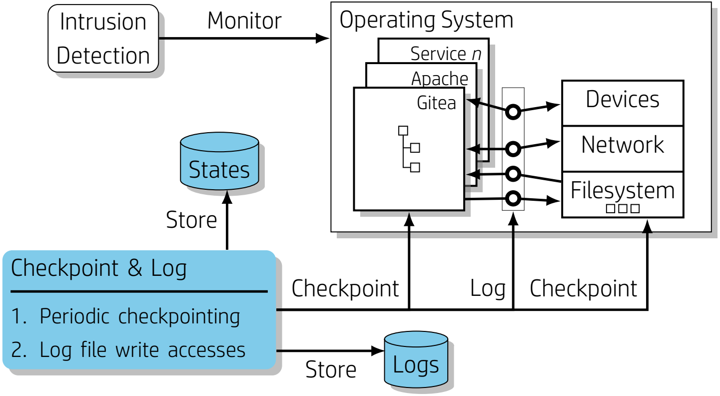
Approach Overview

During the normal operation of the system



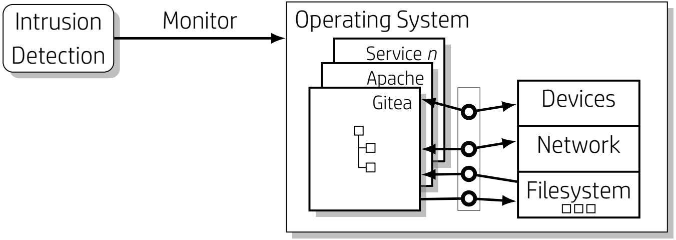
Approach Overview

During the normal operation of the system



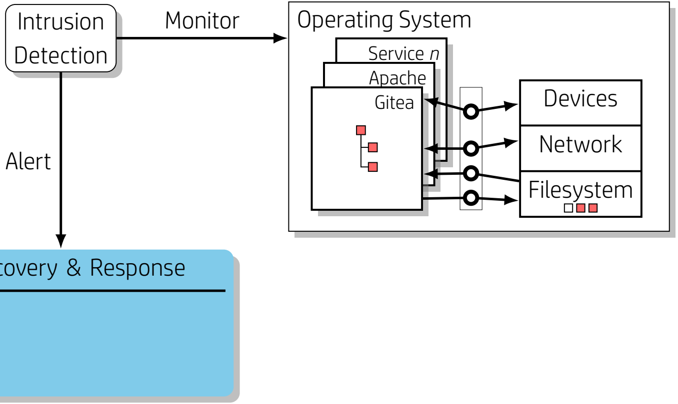
Approach Overview

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



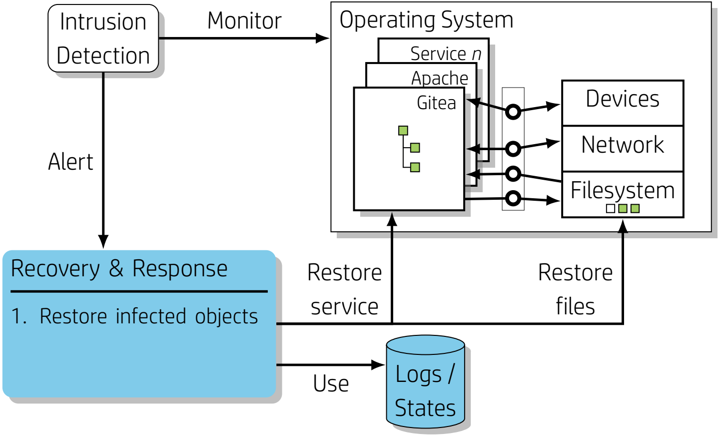
Approach Overview

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



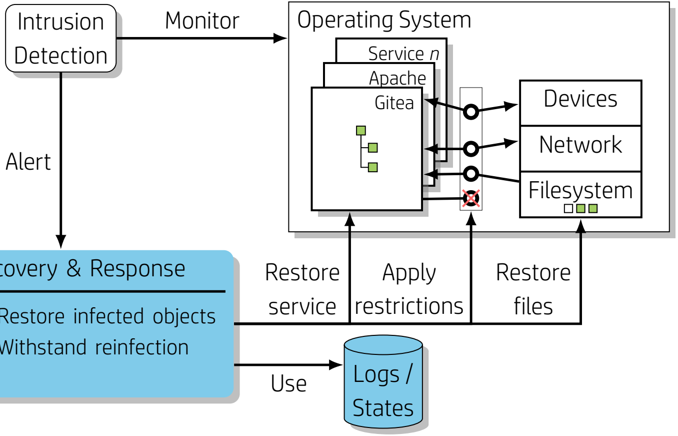
Approach Overview

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



Approach Overview

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

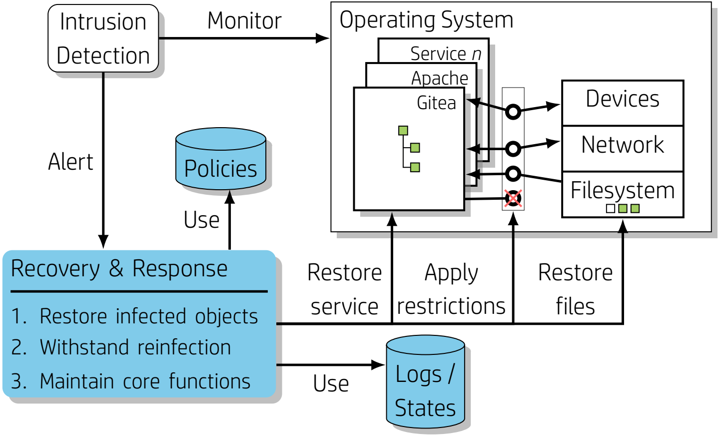


Remove privileges and decrease resource quotas

Per-service responses to prevent attackers to achieve their goals

Approach Overview

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

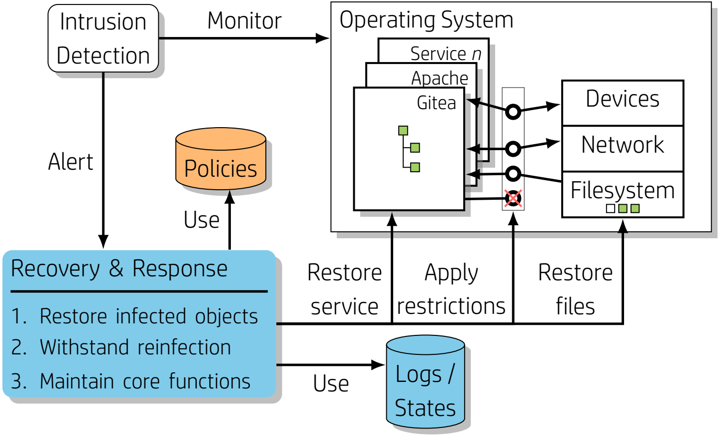


Potential Degraded Mode

The degraded mode maintains core functions **while waiting for patches**

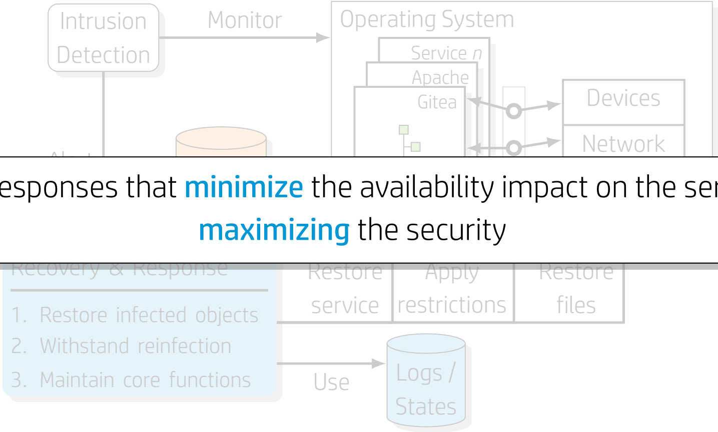
Approach Overview

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



Approach Overview

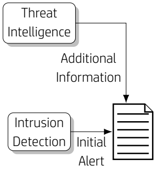
How our approach allows the system to survive intrusions after their detection?



We select responses that **minimize** the availability impact on the service while **maximizing** the security

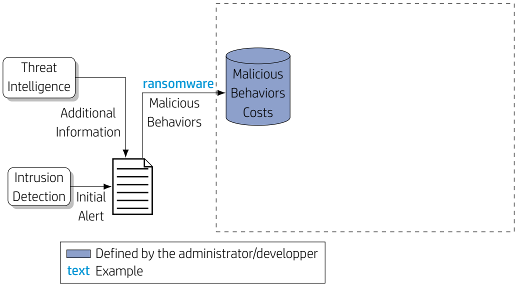
Cost-Sensitive Response Selection

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



Cost-Sensitive Response Selection

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



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
- …

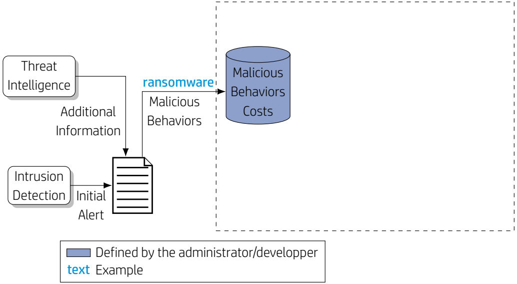
Example of malicious behaviors

Costs

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

Cost-Sensitive Response Selection

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



- 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
    - …
  - …

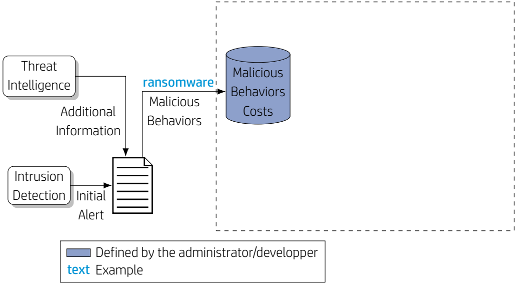
Example of a non-exhaustive malicious behavior hierarchy (Source: MAEC of the STIX project)

Costs

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

Cost-Sensitive Response Selection

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



Malicious behaviors
- Availability violation=**moderate**
  - 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
  - ...
- ...

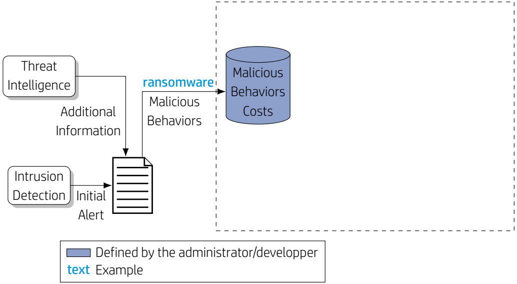
Example of a non-exhaustive malicious behavior hierarchy (Source: MAEC of the STIX project)

Costs

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

Cost-Sensitive Response Selection

understand the intrusion → find possible responses → **assign costs** → 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
  - ...
- ...

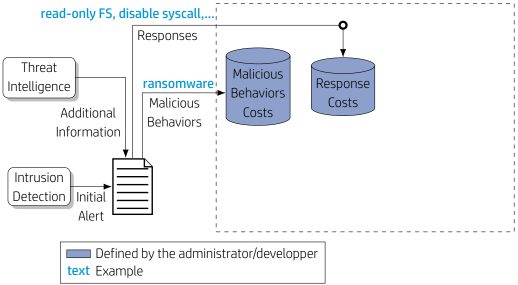
Example of a non-exhaustive malicious behavior hierarchy (Source: MAEC of the STIX project)

Costs

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

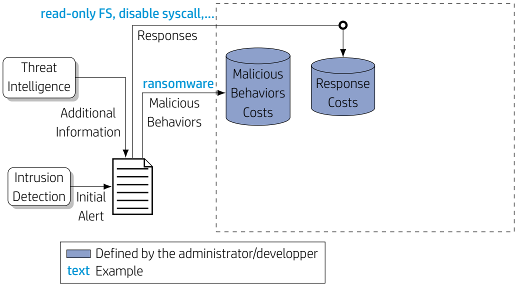
Cost-Sensitive Response Selection

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



Cost-Sensitive Response Selection

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



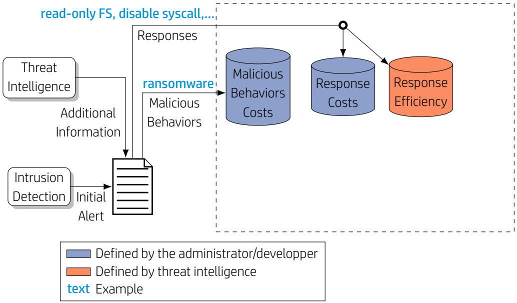
Per-service responses
- File system
  - Read-only file system
    - Read-only path
  - Inaccessible path
- System calls
  - Blacklist any system call
    - Blacklist a list or a category of system calls
- Network
  - Disable network
    - Blacklist IP addresses
    - Blacklist ports
    - ...
- Resources
  - CPU quota
  - ...
- ...

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)

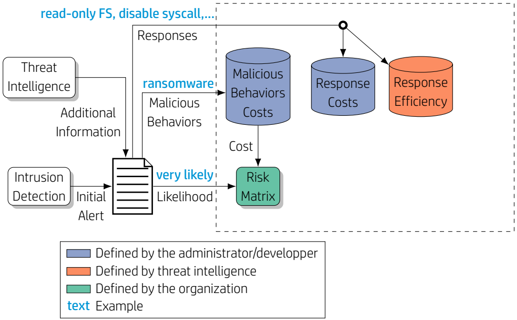
Cost-Sensitive Response Selection

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



Cost-Sensitive Response Selection

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

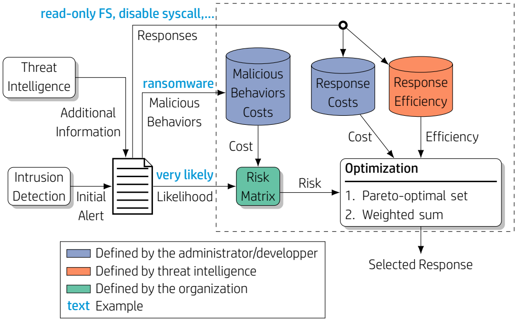


Risk Matrix

Confidence (Likelihood)	Malicious Behavior Cost				
	Very low 0 – 0.2	Low 0.2 – 0.4	Moderate 0.4 – 0.6	High 0.6 – 0.8	Very high 0.8 – 1
Very likely 0.8 – 1	L	M	H	H	H
Likely 0.6 – 0.8	L	M	M	H	H
Probable 0.4 – 0.6	L	L	M	M	H
Unlikely 0.2 – 0.4	L	L	L	M	M
Very unlikely 0 – 0.2	L	L	L	L	L

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

$$\max\left(Risk \times Efficiency + (1 - Risk) \times (1 - Cost)\right)$$

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

read-only FS, disable syscall,...

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

Defined by threat intelligence
Defined by the organization
text Example

$$\max\left(Risk \times Efficiency + (1 - Risk) \times (1 - Cost)\right)$$

Prototype Implementation for Linux-Based Systems

Projects Used or Modified

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

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,…)
- Linux.BitCoinMiner, Linux.Rex.1, Hakai, Linux.Encoder.1, GoAhead Exploit

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

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

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

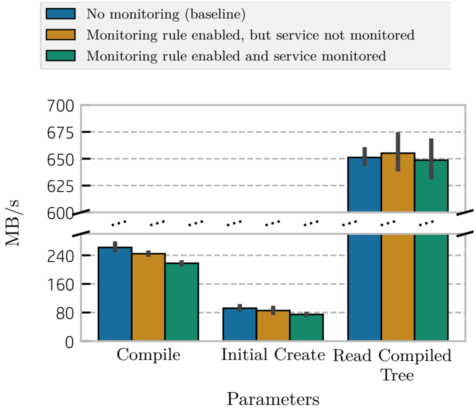
Performance Evaluation

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)

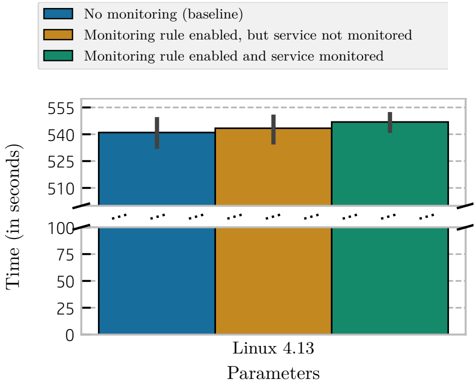
Performance Evaluation

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 %)



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

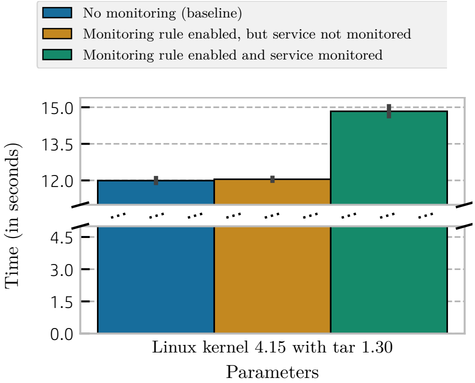
Performance Evaluation

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[12] has 8 % and 67 % overhead

[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

Ronny Chevalier, David Plaquin, Chris Dalton, and Guillaume Hiet. “Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems”. Dec. 2019

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,...

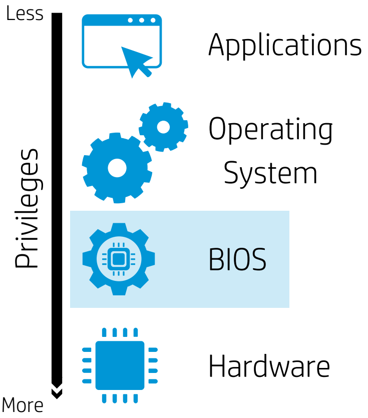
Here, we focus on BIOS/UEFI-compliant firmware

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[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 $\rightarrow$ some platforms are rarely rebooted

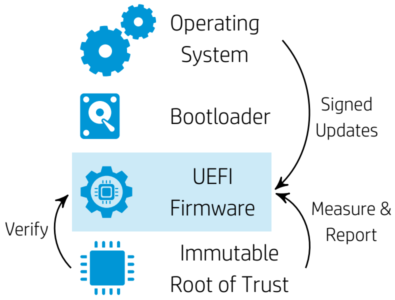
---

[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

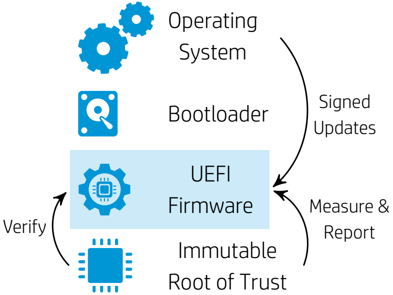
- Signed updates
- Signature verification before executing
- Measurements and reporting to a TPM chip
- Immutable hardware root of trust



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) $\rightarrow$ needs kernel privileges
- SMIs code & data are stored in a protected memory region: System Management RAM (SMRAM)

BIOS code is not exempt from vulnerabilities affecting SMM[14]

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[15]

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

Event-Based Approaches[16]

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

---

[15] Petroni et al., "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor"; Bulygin and Samyde, "Chipset based approach to detect virtualization malware".

[16] Lee et al., "KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object"; Z. Liu et al., "CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM".

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

Few solutions were designed to monitor the SMM at runtime

Snapshot-Based Approaches[15]

- Periodic snapshot of the target's state

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

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

[15] Petroni et al., "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor"; Bulygin and Samyde, "Chipset based approach to detect virtualization malware".

[16] Lee et al., "KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object"; Z. Liu et al., "CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM".

Agenda

Introduction: Preventing, Detecting, and Surviving Intrusions

Surviving Intrusions at the Operating System Level

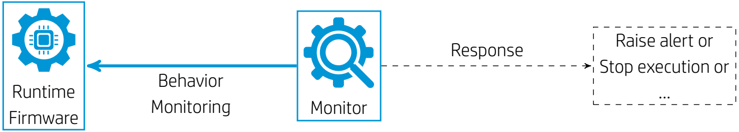
Detecting Intrusions at the Firmware Level

- Background, Use Case, and State of the Art
- Approach and Prototype
- Evaluation
- Conclusion

Conclusion and Perspectives

Our objective

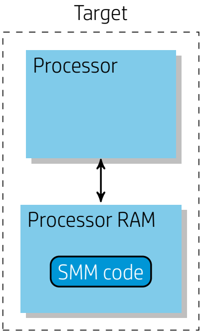
Our goal is to detect attacks that modify the **expected behavior** of the SMM by **monitoring** its behavior **at runtime**.



Such a goal raises the following questions:

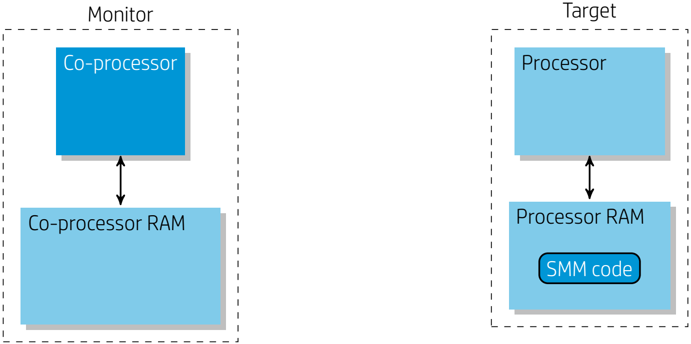
- How to ensure the integrity of the monitor?
- How to define a correct behavior?
- How to monitor?

Approach overview



Approach overview

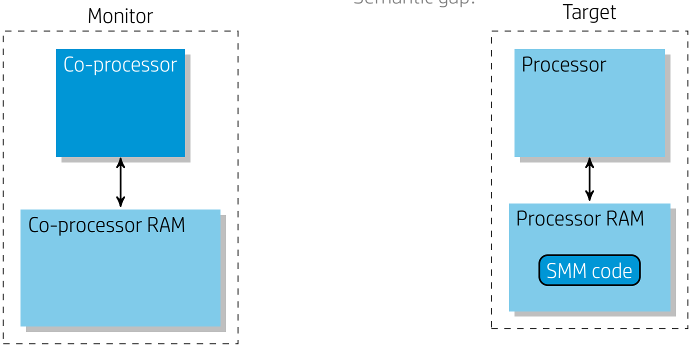
How to ensure the integrity of the monitor?



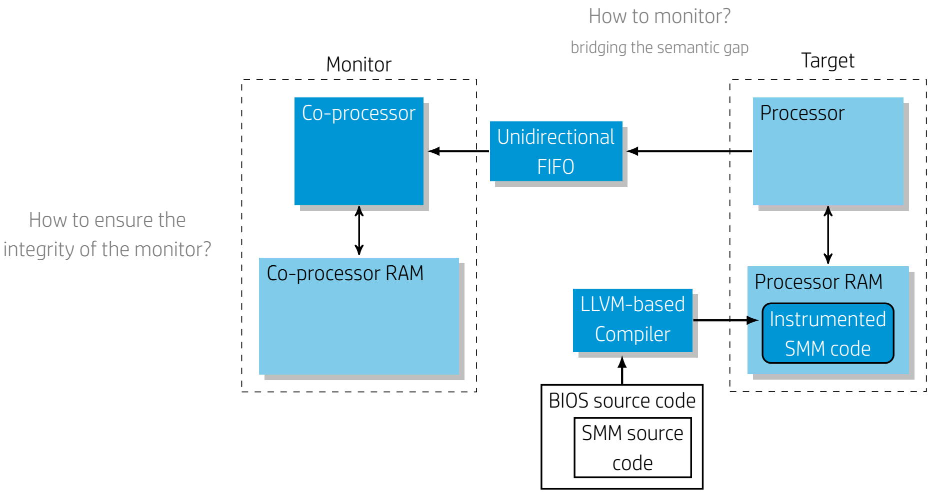
Approach overview

How to ensure the integrity of the monitor?

Semantic gap?



Approach overview

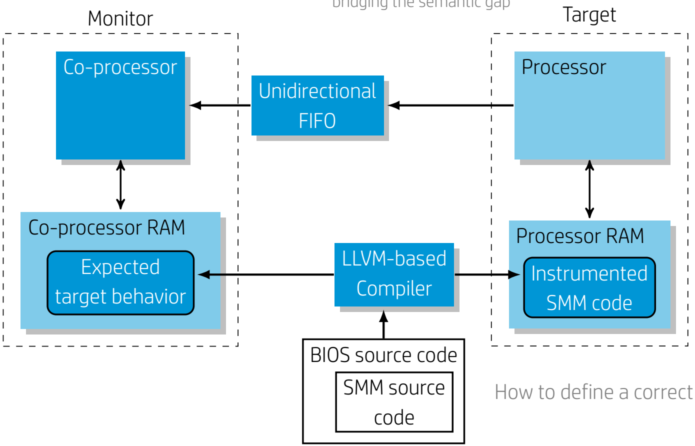


Approach overview

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

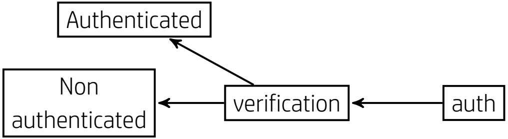
Example

```
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

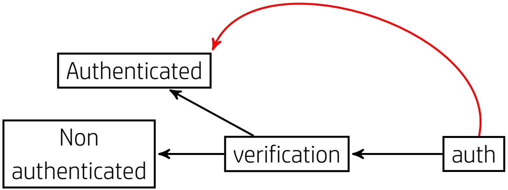
Example

```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



How to define a correct behavior?

Control Flow Integrity (CFI): principle

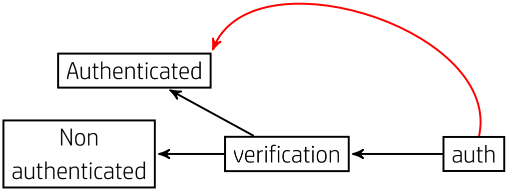
Example

```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



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

```c
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

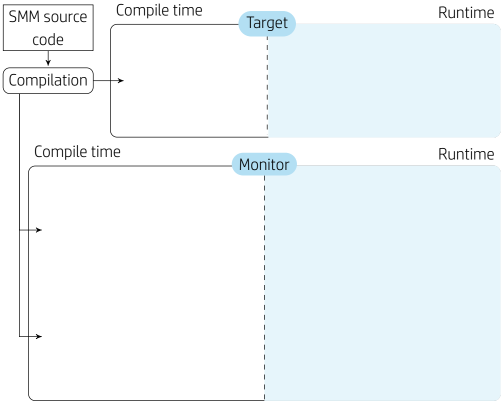
We focus on indirect branches integrity

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

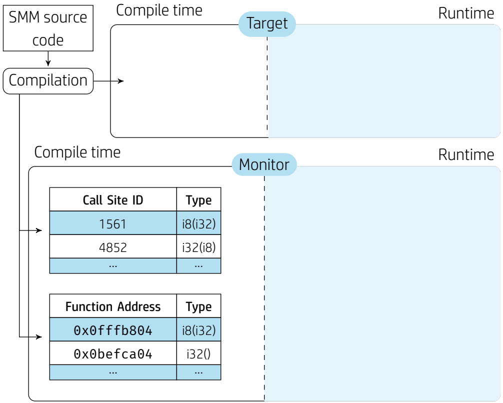
We focus on indirect branches integrity

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 */
  [...]
}
```



Call Site ID	Type
1561	i8(i32)
4852	i32(i8)
...	...

Function Address	Type
0x0fffb804	i8(i32)
0x0befca04	i32()
...	...

How to define a correct behavior?

Control Flow Integrity (CFI): type-based verification

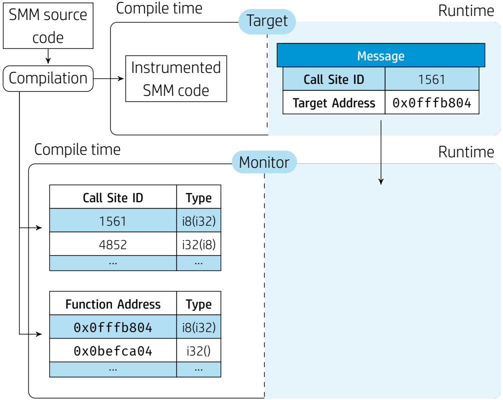
We focus on indirect branches integrity

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

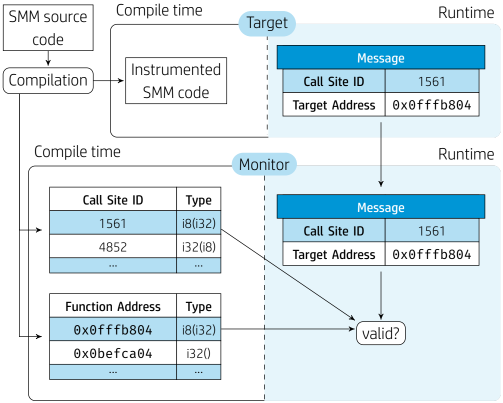
We focus on indirect branches integrity

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 */
  [...]
}
```

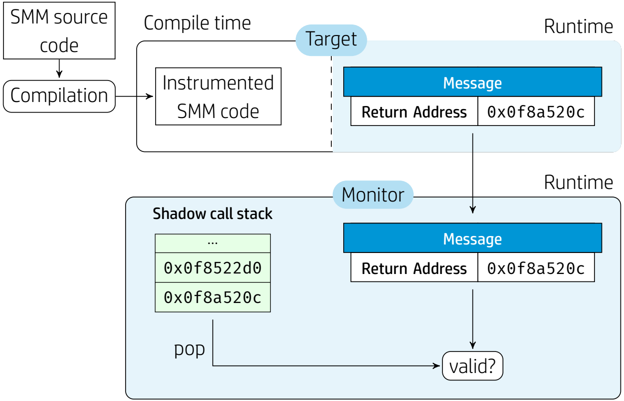


How to define a correct behavior?

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

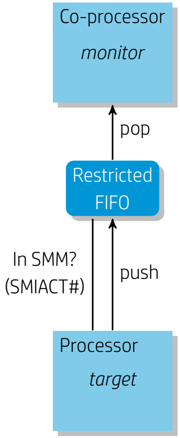
- Acceptable latency of an SMI as defined by Intel BIOS Test Suite: 150 µs
- More than 150 µs per SMI handler leads to degradation of performance or user experience

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 (SMIACT# signal)
- Performance
  → Use a low latency interconnect



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 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

Security evaluation

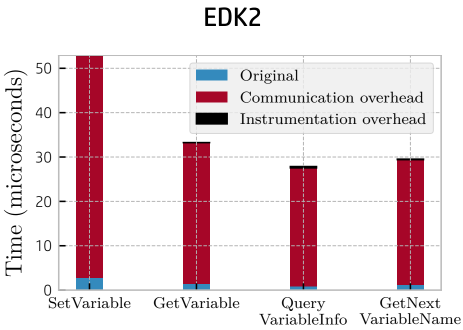
We simulated attacks that exploited vulnerabilities similar to those found in real-world BIOSes

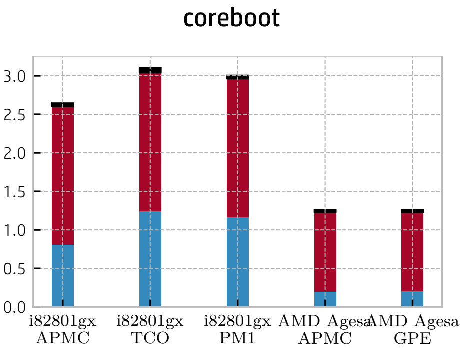
Vulnerability	Attack Target	Security Advisories	Detected
Buffer overflow	Return address	CVE-2013-3582	Yes
Arbitrary write	Function pointer	CVE-2016-8103	Yes
Arbitrary write	SMBASE	LEN-4710	Yes
Insecure call	Function pointer	LEN-8324	Yes

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





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

Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. "Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode". Dec. 2017

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

ACSAC'19, RESSI'18

- The system survives while waiting for the patches
- Maintains availability while maximizing security
- Linux-based prototype implementation

Detecting Intrusions at the Firmware Level

ACSAC'17

- 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

ACSAC'19, RESSI'18

- The system survives while waiting for the patches
- Maintains availability while maximizing security
- Linux-based prototype implementation

Detecting Intrusions at the Firmware Level

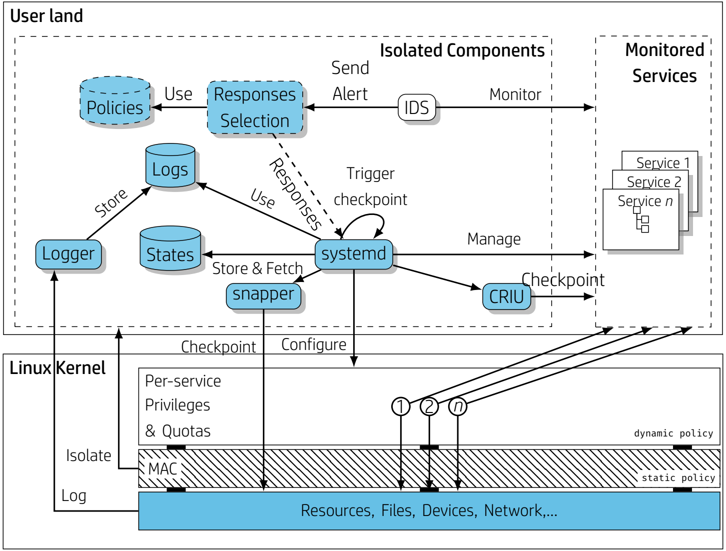
ACSAC'17

- 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

Backup: Surviving

Prototype Implementation for Linux-Based Systems

Architecture Overview



Attack Graphs, Attack Trees, Attack-Defense Trees,...

Models That Depends on Vulnerabilities

Various approaches rely on knowledge about vulnerabilities[17]

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)”[18]

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

---

[17] Foo et al., “ADEPTS: Adaptive Intrusion Response Using Attack Graphs in an E-Commerce Environment”; Kheir et al., “A Service Dependency Model for Cost-sensitive Intrusion Response”; Shameli-Sendi, Louafi, et al., “Dynamic Optimal Countermeasure Selection for Intrusion Response System”.

[18] Ablon and Bogart, *Zero Days, Thousands of Nights: The life and Times of Zero-Day Vulnerabilities and Their Exploits*.

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[19]

---

[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

Service	Checkpoint Size
Apache	26.2 MiB
nginx	7.5 MiB
mariadb	136.0 MiB
beanstalkd	130.1 KiB

Memory pages took at least 95.3 % of the size of their checkpoint

Availability Cost Details

Checkpoint

Checkpoint Operation		Mean	Standard deviation	Standard error of the mean
Service-independent operations				
Initialize	(μs)	643.20	90.75	14.35
Checkpoint service metadata	(μs)	51.47	8.45	1.33
Snapshot file system	(ms)	98.95	1.38	2.19
Checkpoint processes (CRIU)				
httpd	(ms)	199.24	11.05	3.49
nginx	(ms)	51.59	3.99	1.26
mariadb	(ms)	171.77	8.52	2.69
beanstalkd	(ms)	16.25	1.37	0.43
Total				
httpd	(ms)	298.88		
nginx	(ms)	151.24		
mariadb	(ms)	271.41		
beanstalkd	(ms)	115.89		

Time to perform the checkpoint operations of a service

Availability Cost Details

Restore

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

Time to perform the restore operations of a service

Backup: Detecting

Security evaluation

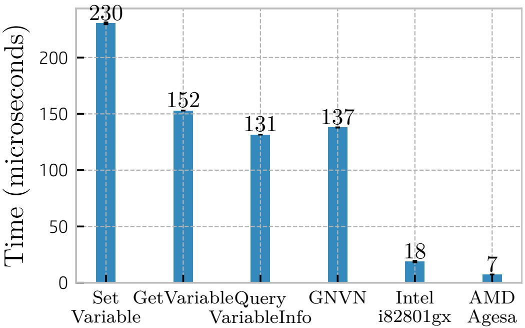
Number and size of equivalence classes for the type-based verification

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



Performance evaluation

Number of packets sent due to the instrumentation

SMI Handler	Number of packets sent: Shadow stack (SS)	Indirect call (IC)	SMBASE & CR3 (SC)	Total number of packets
EDK II				
VariableSmm				
SetVariable	384	4	4	392
GetVariable	240	4	4	248
QueryVariableInfo	299	4	4	208
GetNextVariableName	212	4	4	220
coreboot				
Intel i82801gx				
APMC/TCO/PM1	8	2	4	14
AMD Agesa Hudson				
APMC/GPE	4	0	4	8

Figure 1: Number of packets sent during one SMI handler (Number of packets per message type: SS=2, IC=2, SC=4)

Threat model & assumptions

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

$\rightarrow$ 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)

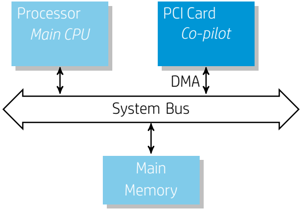
What kind of attack?

Runtime attack by triggering memory corruption issues in an SMI handler (e.g., ROP)

Related work

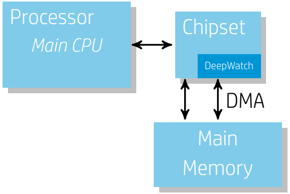
Snapshot-based approach

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



✓ Flexible ✗ Cannot monitor SMM code
✗ Semantic gap ✗ Transient attacks
✗ Additional hardware

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

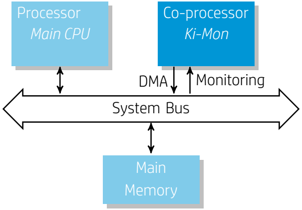


✓ Flexible ✓ Can monitor SMM code
✗ Semantic gap ✗ Transient attacks
✓ No additional hardware

Related work

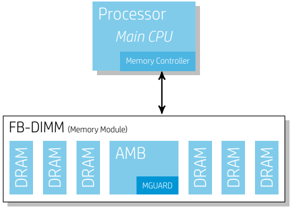
Event-driven approach

Ki-Mon [Lee et al., "KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object"]



✓ 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

[21] http://www8.hp.com/h20195/v2/GetPDF.aspx/4AA6-9339ENW.pdf

References i

- Ablon, Lillian and Andy Bogart. *Zero Days, Thousands of Nights: The life and Times of Zero-Day Vulnerabilities and Their Exploits*. RAND Corporation, 2017. DOI: 10.7249/RR1751.
- Anderson, James P. *Computer Security Threat Monitoring and Surveillance*. Tech. rep. James P. Anderson Co., Fort Washington, PA. Apr. 1980. URL: http://seclab.cs.ucdavis.edu/projects/history/papers/ande80.pdf.
- Balepin, Ivan et al. "Using Specification-Based Intrusion Detection for Automated Response". In: *Recent Advances in Intrusion Detection*. 2003, pp. 136–154. DOI: 10.1007/978-3-540-45248-5_8.
- Bazhaniuk, Oleksandr et al. "A new class of vulnerabilities in SMI handlers". (Vancouver, B.C., Canada). CanSecWest. 2015. URL: https://cansecwest.com/slides/2015/A%20New%20Class%20of%20Vulnin%20SMI%20-%20Andrew%20Furtak.pdf.

References ii

- Bulygin, Yuriy, Oleksandr Bazhaniuk, et al. “BARing the System: New vulnerabilities in Coreboot & UEFI based systems”. REcon Brussels. 2017. URL: `https://www.c7zero.info/stuff/REConBrussels2017_BARing_the_system.pdf`.
- Bulygin, Yuriy and David Samyde. “Chipset based approach to detect virtualization malware”. Black Hat USA. 2008. URL: `http://www.c7zero.info/stuff/bh-usa-08-bulygin.ppt`.
- 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`.

- CERT C Coding Standard. *EXP12-C. Do not ignore values returned by functions*. Aug. 30, 2019. URL: `https://wiki.sei.cmu.edu/confluence/display/c/EXP12-C.+Do+not+ignore+values+returned+by+functions`.
- Chevalier, Ronny, David Plaquin, Chris Dalton, et al. "Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems". In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACSAC'19. ACM, Dec. 2019. DOI: `10.1145/3359789.3359792`.
- Chevalier, Ronny, David Plaquin, and Guillaume Hiet. "Intrusion Survivability for Commodity Operating Systems and Services: A Work in Progress". In: *Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. RESSI'18. May 2018. URL: `https://ressi2018.sciencesconf.org/190500/document`.

- Chevalier, Ronny, Maugan Villatel, et al. "Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode". In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACSAC'17. ACM, Dec. 2017, pp. 399–411. DOI: `10.1145/3134600.3134622`.
- Cooper, David et al. *BIOS protection guidelines*. Tech. rep. NIST Special Publication 800-147. National Institute of Standards and Technology, Apr. 2011. DOI: `10.6028/NIST.SP.800-147`.
- Denning, Dorothy E. "An Intrusion-Detection Model". In: *Proceedings of the 1986 IEEE Symposium on Security and Privacy* (Oakland, CA, USA). IEEE Computer Society, Apr. 1986, pp. 118–131. DOI: `10.1109/SP.1986.10010`.
- Ellison, Robert J. et al. *Survivable Network Systems: An emerging discipline*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, Nov. 1997. URL: `https://apps.dtic.mil/dtic/tr/fulltext/u2/a341963.pdf`.

References v

- Foo, Bingrui et al. “ADEPTS: Adaptive Intrusion Response Using Attack Graphs in an E-Commerce Environment”. In: *Proceedings of the International Conference on Dependable Systems and Networks*. DSN ’05. 2005, pp. 508–517. DOI: `10.1109/DSN.2005.17`.
- Goel, Ashvin et al. “The Taser Intrusion Recovery System”. In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom). SOSP ’05. 2005, pp. 163–176. DOI: `10.1145/1095810.1095826`.
- HP Inc. *HP Sure Start: Automatic Firmware Intrusion Detection and Repair*. Tech. rep. HP Inc., Jan. 2019. URL: `http://h10032.www1.hp.com/ctg/Manual/c06216928`.
- Intel Corporation. “Control-flow Enforcement Technology Specification”. May 2019. URL: `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`.

- Kallenberg, Corey et al. “Defeating Signed BIOS Enforcement”. EkoParty, Buenos Aires. 2013. URL: `https://www.mitre.org/sites/default/files/publications/defeating-signed-bios-enforcement.pdf`.
- Kheir, Nizar et al. “A Service Dependency Model for Cost-sensitive Intrusion Response”. In: *Proceedings of the 15th European Conference on Research in Computer Security* (Athens, Greece). ESORICS’10. 2010, pp. 626–642. DOI: `10.1007/978-3-642-15497-3_38`.
- Knight, John C. and Elisabeth A. Strunk. “Achieving Critical System Survivability Through Software Architectures”. In: *Architecting Dependable Systems II*. Ed. by Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky. 2004, pp. 51–78. ISBN: 978-3-540-25939-8.

Lee, Hojoon et al. “KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object”. In: *Proceedings of the 22th USENIX Security Symposium* (Washington, D.C., USA). USENIX Association, 2013, pp. 511–526. URL: `https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_lee.pdf`.

Liu, Ziyi et al. “CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM”. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel). ISCA ’13. ACM, 2013, pp. 392–403. ISBN: 978-1-4503-2079-5. DOI: `10.1145/2485922.2485956`.

Morin, Benjamin and Ludovic Mé. “Intrusion detection and virology: an analysis of differences, similarities and complementariness”. In: *Journal in Computer Virology* 3.1 (Apr. 1, 2007), pp. 39–49. DOI: `10.1007/s11416-007-0036-2`.

- Petroni Jr., Nick L. et al. “Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor”. In: *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, USA). USENIX Association, Aug. 2004, pp. 179–194. URL: `https://www.usenix.org/legacy/events/sec04/tech/full_papers/petroni/petroni.pdf`.
- Pujos, Bruno. *SMM unchecked pointer vulnerability*. May 2016. URL: `http://esec-lab.sogeti.com/posts/2016/05/30/smm-unchecked-pointer-vulnerability.html` (visited on 08/05/2019).
- Regenscheid, Andrew R. *Platform Firmware Resiliency Guidelines*. Tech. rep. Special Publication 800-193. National Institute of Standards and Technology, Apr. 2018. DOI: `10.6028/NIST.SP.800-193`.
- Researchers, ESET. *LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group*. Tech. rep. ESET, Sept. 2018. URL: `https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf`.

- Shameli-Sendi, Alireza, Mohamed Cheriet, and Abdelwahab Hamou-Lhadj. “Taxonomy of Intrusion Risk Assessment and Response System”. In: *Computers & Security* 45 (Sept. 2014), pp. 1–16. DOI: `10.1016/j.cose.2014.04.009`.
- Shameli-Sendi, Alireza, Habib Louafi, et al. “Dynamic Optimal Countermeasure Selection for Intrusion Response System”. In: *IEEE Transactions on Dependable and Secure Computing* 15.5 (2018), pp. 755–770. DOI: `10.1109/TDSC.2016.2615622`.
- Trusted Computing Group. *TPM Main, Part 1 Design Principles*. Trusted Computing Group. Mar. 2011. URL: `https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf`.
- UEFI Forum. *Unified Extensible Firmware Interface Specification*. Version 2.8. Mar. 2019. URL: `https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf`.

References x

Xiong, Xi, Xiaoqi Jia, and Peng Liu. "SHELF: Preserving Business Continuity and Availability in an Intrusion Recovery System". In: *Proceedings of the 25th Annual Computer Security Applications Conference*. ACSAC '09. IEEE Computer Society, 2009, pp. 484–493. DOI: `10.1109/ACSAC.2009.52`.

Images Credits

URLs provided

Image	Name	Author	License
	Rollback	Gyorgy Hunor-Arpad	CC BY 3.0 US
	Application	Christopher	CC BY 3.0 US
	Chip Settings	Luis Rodrigues	CC BY 3.0 US
	Gear	Jonathan Higley	CC0 1.0 Universal
	Harddrive	Creaticca Creative Agency	CC BY 3.0 US
	Microchip	Creative Stall	CC BY 3.0 US
	Research	Gregor Cresnar	CC BY 3.0 US