Lares: An Architecture for Secure Active Monitoring Using Virtualization

Bryan D. Payne, Martim Carbone, Monirul Sharif, Wenke Lee

2008 IEEE Symposium on Security and Privacy
Motivation

Host-based security tools are not adequately protected.

- Antivirus
- IDS

We need active monitoring for prevention

Lares - Applies VM techniques for active monitoring

Presents an architecture and sample implementation
Active vs. Passive Monitoring

Passive Monitoring
- Run external scanning or polling
- **Detect** a malicious access (e.g. memory corruption) and **Correct**
- Example: VM Introspection

Active Monitoring
- Trigger a handler when an event happens
- **Prevent** a malicious access
- Example: Antivirus, IDS
Major Challenges

Malware monitoring requires

- The analyzer and monitor must be hidden from malware

Critical requirements:

- For active monitoring we need to place hooks on the running system
- Hooks are in the untrusted OS
- How to protect the hooks and its comm. to VM?
System Layout

- Untrusted
  - Guest VM
- Trusted
  - Security VM

- Hypervisor
- Trusted

- Hardware
Adversary Model

- \( N_e \) is triggered if and only if \( e \) occurs legitimately
- \( I_e \) is not modifiable between occurrences of \( e \) and \( N_e \)

3. \( B(I_e) \) of the security application is not altered maliciously
4. The effects of \( R(I_e) \) on the system are enforced
Design Goals

- **Protect Monitoring Components**
  - Meet formal requirements

- **Flexible Hook Placement**
  - Difficult since adversary controls the host system

- **Acceptable Performance Overhead**
Solution Approach

Split security solution over two VMs

Guest VM
- VM running the host OS
- Controlled by adversary
- Contains hooks

Security VM
- Contain security application (antivirus, IDS)
- Trusted
- Runs at same privilege level as hypervisor

Use hypervisor-based memory protection to secure the hooks
Assumptions

- The hypervisor and Security VM form a trusted code base (TCB)
- The machine can undergo secure boot
- Guest VM undergoes an initialization after boot
  - Start the security components
  - Protect them
  - Add other security configurations
Architecture Overview
Guest VM Components

Hooks
- JUMP in program code
- Redirection in jump tables
- Any other technique to transfer control

Trampoline
- Bridge between the hooks and security drivers in security VM
- Passes arguments to the security driver
Requirements for designing trampoline

- Should be completely self-contained
- Execute atomically at each round
- Usage of data elements must be completely non-persistent
Security VM Components

- Security application
- Introspection API
- Security driver
Hypervisor Components

- Guest OS component protection
  - Mark the memory location of hooks and trampoline as read-only

- Inter-VM communication
  - Delay returning to Guest OS until security VM gives a response
Implementation

Uses hook on Windows syscall `NtCreateSection`

Wrote a driver to install hook and trampoline

- Initied during Guest OS initialization
- Never swapped to disk
- Hypervisor activate memory protection
- 324 SLOC
Inter-VM Communication
- Added a new hypercall
- Uses a shared memory to send (receive) event request (response) to (from) hypervisor
- 127 SLOC

Security Driver
- Imparts modularity/flexibility
- Interfaces with security application
- Built as Linux Kernel Module (LKM)
- 182 SLOC
Security application

- Checks the file handle that was passed to `NtCreateSection`
- Allow execution if permitted
- 298 SLOC
Memory Protection

- Takes advantage of shadow paging in Xen
- Modify Xen’s page fault handler to enable byte-sized granularity
- 78 SLOC
Performance

Settings:
- Intel Core Duo, 2GB RAM
- Hypervisor: Xen
- Guest VM: Windows XP, 384MB
- Security VM: Fedora

Hook processing time:
- Traditional Hook (security application in same VM as host)
  - 17 micro-sec
- Lares Hook (best case, minimal processing in security application)
  - 28 micro-sec
- Lares Hook (with processing)
  - 150 micro-sec
Results

- Traditional Hook
- Lares Hook (best case)
- Lares Hook (with processing)
Adversary Model

- $N_e$ is triggered if and only if $e$ occurs legitimately
- $I_e$ is not modifiable between occurrences of $e$ and $N_e$

3. $B(I_e)$ of the security application is not altered maliciously
4. The effects of $R(I_e)$ on the system are enforced

Legend of attacks:
A1: Bypass hook
A2: Modify event context
A3: Tamper with security application
A4: Tamper with dependencies
A5: Tamper with response
Security Analysis

- A3, A4 are met as per trusted VM assumptions
- A2, A5 are met by disabling interrupts by the trampoline
- A5 may be circumvented by non-maskable interrupts (NMI) [hardware fatal errors]
- But one VM uses a single core only
  - Malware cannot use the other core to send NMI
A1.5 and A1.6 handled by memory protection
Write protect IDT to prevent A1.2 and A1.4
A1.1 cannot be handled by Intel VT-x
AMD SVM can trap changes to IDTR
A1.3 is more difficult to handle

- Changes to GDTR, GDT, Page Tables
- Memory Introspection has similar problems
- Monitor shadow PT entries to check that they point to known good locations

Preventing bogus event notification

- Mark the trampoline memory region as non-executable (NX)
- On fault, check origin of branch
  - If valid hook, proceed
  - Else attacker is making bogus call to trampoline
Conclusion

- Enable active monitoring with VM security
- Takes advantages of secure VM
- Can place hooks at arbitrary location in Kernel
- Low overhead
- Works on production systems
- Strong adversary model
- No special hardware needed