Guest-Transparent Instruction Authentication for Self-Patching Kernels

Purdue University / CERIAS
Guest-Transparent Instruction Authentication for Self-Patching Kernels

- Context
- Problem
- Approach
- Evaluation
- Summary
THE CONTEXT:
VMM-based prevention of kernel rootkits.
Kernel rootkits operate with kernel-level permissions
  - Full control over the system

In-kernel protection mechanisms are vulnerable to kernel rootkits
  - Ex. memory permission restrictions can be turned off by a rootkit
Security mechanisms have been created to prevent kernel rootkits by leveraging VMM capabilities such as introspection and memory protection.

- For our work, we build-upon one such mechanism: NICKLE.
- Other similar systems exist: hvmHarvard and SecVisor.

Though each system has its own unique contributions we refer to NICKLE, hvmHarvard, and SecVisor as “NICKLE-like systems.”
Context: VMM Enforced $W \oplus KX$

- How does NICKLE prevent kernel rootkit infection?

- Run vulnerable system in a virtual machine
- Enforce $W \oplus KX$ kernel memory permissions from VMM

| Vulnerable VMM + $W \oplus KX$ | Hardware |
NICKLE Guarantee:

- No unauthorized code can be executed at the kernel level

Kernel rootkits typically need to introduce their own malicious code into a running kernel to gain control.

Because NICKLE does not allow memory to be both writable *and* kernel-executable:

- The malicious code introduction will be prevented, or
- The malicious code will be introduced but will not be executable
• The system must allow:
  – an authorized kernel to get loaded into place at boot
  – authorized kernel modules to get loaded during run-time
• Code introduced into the kernel is marked non-executable until it is authenticated
• If the code is authenticated:
  – The code is set to executable and read-only
To authenticate kernel code, previous works have used cryptographic hashes

- Offline: a cryptographic hash is calculated for each piece of authorized code that may get loaded into the guest kernel
- Online: the VMM intercepts each guest attempt to load new kernel code and calculates a hash for the code
- If the online hash matches an offline hash, the load is allowed
THE PROBLEM:
How do we authenticate self-patching kernels?
• Some kernels are self-patching; they modify their own code at runtime
  – CPU optimizations, multiprocessor compatibility adjustments, and advanced debugging
• In a NICKLE-like system:
  – If the patch is applied after hash verification
    • The memory will be read-only and the patching will fail
  – If the patch is applied before hash verification
    • The code authentication will fail
Problem: Self-Patching Kernels

Host Memory Layout

Host Kernel Space

... 

Stack ↓

Host User Space

Heap ↑
Problem: Self-Patching Kernels

VMM Reserves Heap Memory for Virtual Machine
Problem: Self-Patching Kernels

Guest Memory Layout

Guest Kernel Space

Stack ↓

Guest User Space

Heap ↑
Problem: Self-Patching Kernels

Kernel Module to be Loaded

Host Kernel Space

Host User Space

Heap ↑

Stack ↓

Guest Kernel Space

Guest User Space

Stack ↓

Heap ↑
Problem: Self-Patching Kernels

Module Allowed, Code Set to Non-Executable (NX)
Problem: Self-Patching Kernels

Hash\textsubscript{B} Compared to Hash\textsubscript{A} Stored in VMM

Hash\textsubscript{B} Calculated for Module Code
Problem: Self-Patching Kernels

If Hash\textsubscript{B} Matches Hash\textsubscript{A}, Set Module Code Read-Only & Executable
Problem: Self-Patching Kernels

Guest Kernel Tries to Patch Module Code

Patch Fails Because Module Code is Read-Only
Example from the Linux kernel
  – Alternative Instructions (altinstructions)
Altinstructions enable the kernel to optimize code at run-time based on the capabilities of the CPU
  – At compile time, a list of alternative instructions may be stored in special ELF headers
  – At run-time, if an altinstruction is defined for the current CPU, the alternative replaces the default instruction
Why do this?
  – Linux distributors can ship just one binary that will be optimized for multiple CPUs
Problem: Ex: Altinstructions

- Altinstructions Example: “run-time memory barrier patching” (RTMBP)
  - Default Linux memory barrier instruction sequence:
    - lock; addl $0,0(%esp)
  - Memory barrier instruction sequence for Pentium 4:
    - lfence
RTMBP in the context of NICKLE-like system
- Offline a hash would be calculated over the unoptimized code
- Two options for online authentication:
  - Unoptimized code would pass authentication and guest would run without the RTMBP optimization
  - Optimized code would fail authentication
OUR APPROACH:
Verify each patch at run-time.
Approach: Definitions

- **Patch**: Each valid replacement instruction sequence
- **Patch Definition**:
  - patch-location: where the patch may get applied
  - patch-length: size of the replacement instruction
  - patch-data: holds the replacement instruction
- **Patch Set**: whitelist of valid patch definitions
- **Patch Site**: location in code which may or may not be patched at run time
Approach: Patch Example

Source Code ➔ x86 Assembly ➔ Raw Hex Dump

rcu_idle_enter();        call 0xc04bb940
while (!need_resched()) {
    lea 0x0(%esi),%esi
    mov %esp,%eax
    and $0xffffe000,%eax
    mov 0x8(%eax),%eax
    test $0x8,%al
    jne 0xc0402366
    lock addl $0x0,(%esp)
    bt %ebx,(%esi)
    sbb %eax,%eax
    test %eax,%eax
    je 0xc0402366
    call 0xc04063b0
check_pgt_cache();
local_touch_nmi();

while (!need_resched()) {
    check_pgt_cache();
    rmb();
    if (cpu_is_offline(cpu))
        play_dead();
    local_touch_nmi();

Example patch:
lock addl $0x0,(%esp)

<table>
<thead>
<tr>
<th>location</th>
<th>length</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xc040232e</td>
<td>6</td>
<td>f0 83 04 24 00 0f</td>
</tr>
<tr>
<td>lfence</td>
<td>6</td>
<td>0f ae e8 90 90 90</td>
</tr>
</tbody>
</table>
Approach: Patch Verification

• Offline:
  – Generate cryptographic hashes, as before
    • Except: skip patch sites for hash calculation
  – Generate a whitelist of patch definitions
    • At least one definition for each patch site

• Online
  – At code load-time:
    • Verify code hashes, again skipping patch sites
    • Verify contents of each patch site using whitelist
  – At write-fault (caused by $W\oplus KX$ protection):
    • If writing to patch site, allow patch site to be overwritten by valid patch
Approach: Patch Set Creation

• Patch set creation is challenging!
  – Requires deep knowledge of guest kernel (or kernel vendor participation)
• Example: Linux Kernel (v 2.6) has six\(^1\) different mechanisms that may patch the kernel at runtime:
  – Alternative Instructions
  – SMP Locks
  – Jump Labels
  – Mcounts\(^1\)
  – Paravirtual Instructions
  – Kprobes

\(^1\) We identified a sixth mechanism after paper submission.
Approach: Ex. Altinstructions

- Generating a patch definition for an altinstruction

```
struct alt_instr
(from ELF headers)

patch definition
```
OUR EVALUATION:
Add patch verification to a NICKLE-like system.
Our patch-level verification procedure is implemented as a subsystem of NICKLE-KVM.

**NICKLE-KVM**: NICKLE-like system based on KVM
- KVM is a Linux-based VMM that takes advantage of hardware-assisted virtualization
- Uses the page-level redirection technique introduced by hvmHarvard (rather than instruction-level technique used by NICKLE)

Generated patch set for four of the six Linux kernel patching facilities (two weren’t used by our guest)

We implemented the load-time patch verification procedure for NICKLE-KVM.
To evaluate our system we generated patch sets for the Linux kernel\(^2\) (vmlinux) and 3308 kernel modules
- The kernel contained 31,643 patch sites
- The 11 modules needed by our guest contained 639 patch sites

After adding patch-level verification, NICKLE-KVM correctly verified the integrity of all 32282 patch sites

\(^2\) After paper submission we added kernel patch-site verification
Crafted synthetic attacks to test robustness:

- Attacker modifies code outside of patch-site
  - Load fails due to cryptographic hash mismatch
- Attacker modifies code within patch site
  - Load fails if patch site does not receive valid patch
  - Load succeeds if patch site receives valid patch
- Attacker modifies candidate patch code (ex. altinstructions ELF header)
  - Load fails if the spurious patch is selected
Evaluation: Performance

- Our load-time patch verification procedure incurs no additional NICKLE-KVM VM exits for patch-verification
  - Adds time for patch verification (lookup and string compare) proportional to the number of patches
- A write-fault-triggered patch verification procedure (not implemented) would incur one additional VM exit per patch that triggers a write-fault
  - Additional VM exit required for restoring read-only permission
Summary

- Previous NICKLE-like systems were not able to authenticate code introduced by self-patching kernels.
- Our kernel code authentication procedure accommodates self-patching kernels by verifying each patch.
- Our implementation is able to authenticate a self-patching guest Linux kernel and its modules (32,282 patch sites).
THANK YOU!
### At load-time, verify kernel code and set to read-only

### Run-time patch will fail due to read-only permissions
Problem: Self-Patching Kernels

NICKLE Protects Guest Kernel Memory