Architectural Supports to Protect OS Kernels from Code-Injection Attacks

2016-06-18

Hyungon Moon, Jinyong Lee, Dongil Hwang, Seonhwa Jung, Jiwon Seo and Yunheung Paek
Seoul National University
Why to protect the OS kernels?

- Operating systems (and their kernels) are everywhere

- Applications rely on the OS kernels
Operating systems are vulnerable

- New vulnerabilities reported every year
  - CVE-2013-2094 (S. Vogl et al., 2014)
  - CVE-2014-3153 (TowelRoot)
  - CVE-2015-3636 (PingPongRoot)

- Adversaries may
  - **Read** from the memory regions for the kernel
  - **Write** to the memory regions for the kernel

- With the capabilities,
  - Hiding Processes, files, or network connections
  - Privilege escalation
  - Execute their code while the CPU in the kernel mode
A powerful type of attack: code-injection

Handling a read system call
- Supervisor call handler: `sys_read`
- The address of `sys_read` written in the system call table

Attackers can
- Write their code into the kernel’s memory
- Manipulate the system call table

Consequence
- `mal_sys_read` replaces `sys_read`
Existing mechanisms effective

- Privileged eXecute Never (PXN)
  - A flag in the page table entries
  - MMU prevents the execution of memory pages with PXN=1

- Page Table Protection $\Rightarrow$ No Code-Injection Attack
Kargos overview

Goal
- Mitigate the kernel code-injection attacks with minimal performance cost

Threat Model
- Adversaries can read from/write to the kernel memory arbitrarily

Mechanism
- Dedicated hardware support
  - Traffic Monitor
  - Trace Monitor
- Minimal kernel instrumentation
  - Special execution traces
  - Special register protection
The four rules to detect the attacks

R1. The physical code regions of the kernel should never be modified
The four rules to detect the attacks

R2. The CPU jumps to an address in the virtual code regions when entering the kernel.
The four rules to detect the attacks

R3. All indirect branch targets lie in the virtual code regions while the CPU is in the kernel mode.
The four rules to detect the attacks

R4. All virtual code regions are mapped to the physical code regions.
Why the four rules prevent the attacks

- **R1**: attacker’s code should be outside the physical regions
- **R2 & R3**: PC points to the virtual code regions
- **R4**: Virtual code regions never mapped to the attacker’s code

![Diagram showing virtual memory and physical memory with mappings]

1. Virtual Memory
   - Virtual Code Regions
2. Physical Memory
   - Physical Code Regions
   - Map
   - Attacker’s code
Trace monitoring

- Need to monitor the virtual addresses that the CPU jumps to

Our Implementation:
- Parses the ARM's PTM packets
Traffic monitoring

- Need to know the physical addresses that the CPU writes to

Our implementation:
- Examines the traffic complying with the AXI protocol
- Naturally detect the violations of R1
Rule 2: Kernel entrance

- The gateway code blocks
  - SCTLR
  - VBAR

- Vector table is inside the physical code regions

- Protection of the SCTLR and VBAR: Kernel Instrumentation
  - Check the values before executing the special instructions
Rule 3: Indirect branches

- **Challenge: Mode recognition**
  - In which CPU mode a trace is generated?
  - Jump to gateway code block indicates the kernel enter

- **Answer: special traces in the exit code blocks**

```
msr      SPSR_fsxc, r1
and      r3, r1, #31
cmp       r3, #16
subeq    pc, pc, #4
restore_context
movs    pc, lr
```
Rule 4: Mappings

- Memory management unit uses:
  - Partial page table protection
    - Small number of (<10) PGD entries for virtual code region translations
    - Traffic Monitor can detect the modifications
  - TTBR protection: Kernel Instrumentation
    - Check the PGD entries before updating the TTBRs
Prototype implementation details

- Implemented all hardware components in Verilog HDL
- Used Xilinx ZC702 evaluation kit to prototype

Operational frequency:
- Processor core: 222MHz
- Kargos hardware modules: 80MHz

Kernel instrumentations
- Six for SCTLR updates
- Four for TTBR updates
- Two exit code blocks
Evaluation: Security

- Implemented three Proof-of-Concept (PoC) attacks using a real-world vulnerability (CVE-2014-3153)
  - Kernel code modification
  - Virtual code region remapping
  - Redirecting the kernel execution to a attacker’s code block

- Targeting Linux kernel 3.8.0 for Android 4.2

- All these three attacks detected
addresses, using TrafficMonitor. Not require the entire page tables to be protected, as it can detect would corrupt the page tables to deceive the MMU and modify the integrity of all page tables in the system. Otherwise, attackers rely on the MMU to examine the accesses and detect malicious would accept as a genuine one. In Section 3.1.2 to be resilient to the fake reporting attack. Figure 3 code block is implemented to comply with the protocol presented provides TrafficMonitor with the physical addresses of the new entries the atomic code blocks that update the registers, the kernel also pro-

vides TrafficMonitor with the physical addresses of the new entries (address Translation Redirection Attack).

Although we could avoid protecting all page tables in the target system, Kargos has to protect the page table entries which map the privileged to user. Figure 4 shows how the signature for a Linux kernel considers as the sign of the mode change event. With this additional instruction only when the mode field of the SPSR is set to user. This additional instruction generates a trace that Mode Tracker can in detail, the kernel checks the value of the SPSR before execut-

user mode, whereas it follows the original control flow otherwise. For example, the exit code blocks of the Linux kernel for cause the execution of the block may not always cause such a mode change. For this reason, we augmented the register... ARM CPUs use change. For example, the exit code blocks of the Linux kernel for... ARM CPUs use change. For example, the exit code blocks of the Linux kernel for... ARM CPUs use change. For example, the exit code blocks of the Linux kernel for... ARM CPUs use change. For example, the exit code blocks of the Linux kernel for...

Evaluation: Performance 1

LMBench result to show the impact on OS services

<table>
<thead>
<tr>
<th>Name</th>
<th>Baseline</th>
<th>Kargos</th>
</tr>
</thead>
<tbody>
<tr>
<td>null syscall</td>
<td>0.98μs</td>
<td>1.07μs (0.92%)</td>
</tr>
<tr>
<td>open/close</td>
<td>18.39μs</td>
<td>18.15μs (-1.28%)</td>
</tr>
<tr>
<td>select</td>
<td>4.58μs</td>
<td>4.57μs (-0.11%)</td>
</tr>
<tr>
<td>sig. handler install</td>
<td>2.81μs</td>
<td>2.82μs (0.11%)</td>
</tr>
<tr>
<td>sig. handler overhead</td>
<td>9.91μs</td>
<td>10.55μs (6.42%)</td>
</tr>
<tr>
<td>pipe</td>
<td>40.89μs</td>
<td>43.23μs (5.72%)</td>
</tr>
<tr>
<td>fork+exit</td>
<td>2853.15μs</td>
<td>2838.60μs (-0.51%)</td>
</tr>
<tr>
<td>fork+execve</td>
<td>9279.8μs</td>
<td>9159.16μs (-1.3%)</td>
</tr>
<tr>
<td>page fault</td>
<td>4.34μs</td>
<td>4.45μs (3.63%)</td>
</tr>
<tr>
<td>mmap</td>
<td>84.7μs</td>
<td>84.9μs (0.24%)</td>
</tr>
</tbody>
</table>
Evaluation: Performance 2

Application benchmarks for the comparison

<table>
<thead>
<tr>
<th>Name</th>
<th>Baseline</th>
<th>Kargos</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlb</td>
<td>12097.99s</td>
<td>12121.52s (0.19%)</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>7284.54s</td>
<td>7274.29s (-0.14%)</td>
</tr>
<tr>
<td>403.gcc</td>
<td>2420.82s</td>
<td>2429.91s (0.38%)</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>13412.38s</td>
<td>13542.57s (0.97%)</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>15327.28s</td>
<td>15385.06s (0.38%)</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>17000.11s</td>
<td>17051.94s (0.3%)</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>42659.18s</td>
<td>42753.94s (0.22%)</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>18785.86s</td>
<td>18841.65s (0.3%)</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>10334.19s</td>
<td>10382.46s (0.47%)</td>
</tr>
<tr>
<td>473.astar</td>
<td>7717.71s</td>
<td>7684.35s (-0.43%)</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>11235.73s</td>
<td>11257.41s (0.19%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Baseline</th>
<th>Kargos</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL</td>
<td>607.90</td>
<td>610.82 (0.48%)</td>
</tr>
<tr>
<td>CF-Bench</td>
<td>531.80</td>
<td>527.80 (0.75%)</td>
</tr>
<tr>
<td>GeekBench</td>
<td>67.20</td>
<td>67.00 (0.30%)</td>
</tr>
<tr>
<td>Linpack-single</td>
<td>9.01</td>
<td>8.96 (0.64%)</td>
</tr>
<tr>
<td>Vellamo-metal</td>
<td>121.80</td>
<td>121.40 (0.30%)</td>
</tr>
</tbody>
</table>
Detection of kernel code injection attacks is not expensive
- With appropriate hardware supports

Hardware monitors can examine CPU states
- Mode of execution (privileged/user)
- Special register values

Can this mechanism also applied for the detection of the code-reuse attacks?
Thank you!