Remote Direct Memory Introspection

Hongyi Liu, Jiarong Xing, Yibo Huang, Danyang Zhuo, Srinivas Devadas, Ang Chen
Problem: Memory introspection

• Memory introspection is a critical security task
Problem: Memory introspection

- Memory introspection is a critical security task
  - It can detect kernel-level attackers (i.e., rootkits)
  - Agent snapshots raw memory for further forensics
  - Hypervisor-based introspection is widely used
    - E.g., Livewire (NDSS’03), ImEE (SEC’17), LibVMI
Limitations of hypervisor-based introspection

- Hypervisor-based introspection has inherent limitations
  - It causes performance interference with local workloads
  - It contains a large trusted computing base inducing vulnerabilities
  - It is not capable to support baremetal installations
Insight: Dom(-1) security offloading

- Can we solve the problem by moving one layer below?
  - Dom(-1) security is enforced in widely-deployed hardware
  - Dom(-1) substrate enforces efficient security execution
Insight: Dom(-1) security offloading

• Can we solve the problem by moving one layer below?
  • Dom(-1) security enforced in widely-deployed hardware
  • Dom(-1) substrate enforces efficient execution
  • Dom(-1) security supports baremetal installation
Opportunities: Remote Direct Memory Access

- RDMA enables reading/writing remote memory with CPU bypassed
  - RNIC (RDMA NIC) can perform DMA to remote memory over network
  - RDMA has been widely deployed in cloud datacenters

🌟 RDMA can serve as memory datapath for Dom(-1) introspection
Opportunities: Programmable switches

- Programmed with high-level language, e.g., P4.
  - Parse RDMA headers, enforce match-action tables and stateful operations
- Run at line speed (Tbps) and are commercially available
- Have been widely used for network security
  - E.g., PortCatcher-CCS’22, IMAP-NSDI’22, Bedrock-Security’22
- This work is the first to use programmable switches for kernel security

Programmable switches can efficiently enforce control logics
RDMI: Remote Direct Memory Introspection

- RDMI: A new paradigm for memory introspection
  - DSL: Introspection abstractions hiding low-level programming details
  - AIM: Instruction set for better resource sharing and live deployment
  - Runtime: Reconfigurable engines to instantiate AIM instructions
RDMI benefits

- Baremetal support
- Remote execution
- Efficient introspection
- COTS deployment
- Programmable DSL policies

• RDMI offers protections with new benefits
Outline

- Motivation: Better memory introspection
- Opportunity: Dom(-1) security execution
- Approach: Remote Direct Memory Introspection
  - RDMI design:
    - Design #1: Introspection DSL design
    - Design #2: Abstract introspection machine
    - Design #3: Reconfigurable introspection engines
- Evaluation
- Conclusion
Introspection is a “graph processing” task

struct task_struct {
    int pid;
    struct task_struct *nxt_task;
    struct mm_struct *mm;
};

• Memory introspection shares similar execution model with graph processing.

Follow nxt_task and get me each PID!
Introspection is a “graph processing” task

Memory introspection shares similar execution model with graph processing.

- Operators
  - `kgraph(addr)`
    - Initialize traversal at addr
  - `in(ptr)`
    - Dereference ptr into another DS
  - `traverse(ptr_nxt, ptr_end, type)`
    - Traverse `ptr_nxt` until `ptr_end` with type
  - `values(f1, ..., fn)`
    - Acquire values from current address

- Descriptions

```c
struct task_struct {
    int pid;
    struct task_struct *nxt_task;
    struct mm_struct *mm;
};
```

- Gremlin Lang

Follow `nxt_task` and get me each PID!
RDMI query example

// Go through PSlist and retrieve PID

Sample query: Process descriptor list traversal
RDMI query example

```
// Go through PSlist and retrieve PID
// Initialize introspection at init_task
kgraph(init_task)
```

Sample query: Process descriptor list traversal
RDMI query example

// Go through PSlist and retrieve PID
// Initialize introspection at init_task
kgraph(init_task)
// Traverse task linked list
.traverse(tasks.next, &init_task.tasks, task_struct)

Sample query: Process descriptor list traversal
RDMI query example

// Go through PSlist and retrieve PID
// Initialize introspection at init_task
kgraph(init_task)
// Traverse task linked list
.traverse(tasks.next, &init_task.tasks, task_struct)

Sample query: Process descriptor list traversal
RDMI query example

// Go through PSlist and retrieve PID
// Initialize introspection at init_task
kgraph(init_task)
// Traverse task linked list
.traverse(tasks.next, &init_task.tasks, task_struct)

Sample query: Process descriptor list traversal
RDMI query example

```c
// Go through PSlist and retrieve PID
// Initialize introspection at init_task
kgraph(init_task)
// Traverse task linked list
.traverse(tasks.next, &init_task.tasks, task_struct)
// get each pid value while traversing
.values(pid)
```

Sample query: Process descriptor list traversal
RDMI introspection queries

<table>
<thead>
<tr>
<th>Policy</th>
<th>LoC</th>
<th>Policy</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1. Task list traversal</td>
<td>3</td>
<td>P7. Process memory map check</td>
<td>7</td>
</tr>
<tr>
<td>P3. VFS hook detection</td>
<td>4</td>
<td>P9. Module list traversal</td>
<td>4</td>
</tr>
<tr>
<td>P4. Netfilter hijacking detection</td>
<td>7</td>
<td>P10. Afinfo operation check</td>
<td>6</td>
</tr>
<tr>
<td>P5. TTY keylogger check</td>
<td>11</td>
<td>P11. Open file list</td>
<td>11</td>
</tr>
<tr>
<td>P6. Syscall check</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- RDMI can express a range of useful introspection queries with a few LoC
DSL compilation: Naïve solution

Deploying policies incurs downtime

Sharing resources across policies is hard
Abstract introspection machine

- AIMs are underlying building block for DSL primitives
- AIMs are implemented in a master switch program
- AIMs are further compiled to configure the match action table
Reconfigurable introspection engines

- Reconfigurable introspection engines instantiate the AIM instructions
- Engines are implemented as MATs reconfigurable for different AIM streams

<table>
<thead>
<tr>
<th>table</th>
<th>rdma.qpn</th>
<th>meta.pred</th>
<th>param</th>
</tr>
</thead>
<tbody>
<tr>
<td>NxtPC_tab</td>
<td>7</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>NxtPC_tab</td>
<td>8</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>NxtPC_tab</td>
<td>8</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Match action tables (MAT):

```c
table NxtPC_tab {
    key = {
        rdma.qpn: exact;
        meta.pred: exact;
    }
    actions = {compute_NxtPC;}
}
...```

AIM engine

- Endianness engine
- Address trans. engine

Reconfigurable engine sets
Experiment setup

• RDMI prototype:
  • Runs with Tofino Wedge 100BF-32X switch and Mellanox CX-4 NIC
  • 2500 LoC of P4 + 2700 LoC of C++

• Baseline:
  • LibVMI based introspection

• Real world threats:
  • Adore-ng and 5 other rootkits

• Real world applications:
  • Redis and Nginx workloads
Evaluation: Introspection effectiveness

- RDMI detects real-world rootkits in baremetal machine
- RDMI’s policy deployment won’t affect normal traffic
Evaluation: Workload interference

- RDMI’s interference to guest workload is negligible
Summary

• **Motivation:** Better memory introspection
• **Insight:** Dom(-1) security offloading
  • Supported by widely-deployed hardware
• **RDMI: Remote direct memory introspection**
  • DSL support for introspection queries
  • AIM for resource sharing and live deployment
  • Runtime for supporting executions
• RDMI improves cloud security on several aspects
  • E.g., Baremetal support, higher attack detection rates
• **Source code:** [https://github.com/aladinggit/RDMI/](https://github.com/aladinggit/RDMI/) hl87 @ rice dot edu