HyperTP: Mitigating vulnerability windows with hypervisor transplant

Tu Dinh Ngoc\textsuperscript{1}  Boris Teabe\textsuperscript{1}  Alain Tchana\textsuperscript{2,3}  Gilles Muller\textsuperscript{3}  Daniel Hagimont\textsuperscript{1}

\textsuperscript{1}University of Toulouse  \textsuperscript{2}ENS Lyon  \textsuperscript{3}Inria
Motivation

- Need to reduce datacenter’s window of vulnerability
  - Alibaba reports 15 days to upgrade one cluster
- Different hypervisors have different attack surfaces and vulnerability windows
  - Ideally select from a “pool” of hypervisors for the safest one
- Solution: hypervisor transplant
  - Switching from one hypervisor to another
- Combine with existing virtualization management tools (libvirt, OpenStack)
Vulnerability timeline

a) Timeline of a classic flaw handling

- Hypervisor deployment
- Flaw discovered
- Flaw reported
- Patch released
- System patched

Unpatched → Patched

b) Timeline of a flaw handling with hyperTP

- Hypervisor deployment
- Flaw discovered
- Flaw reported
- Patch released
- System patched

Unpatched → Mitigated → Patched

Hypervisor switching with HyperTP
Key contribution: HyperTP

- Combination of 2 approaches: in-place hypervisor transplant ("InPlaceTP") and migration-based transplant ("MigrationTP")
- Built around two main principles: memory separation and unified intermediate VM representation
HyperTP general workflow

1. Save VM states and memory
2. Suspend running VMs
3. Transplant new hypervisor
4. Restore VM states
5. Resume VM
Memory separation

- We identified four categories of hypervisor memory:
  
  most VM-specific...

  - Guest state (i.e. guest-only memory)*
  - VM state (GPA-HPA maps, vCPU state, etc.)*
  - VM management state (list of running VM, hypervisor scheduler)
  - Hypervisor state (non-VM related)
    
    ...most hypervisor-specific
  
  *saved/translated by HyperTP
Unified VM state representation

- Intermediate representation of guest and VM states
  - vCPU
  - Memory
  - I/O devices
- Functions to translate to/from native hypervisor state
- Towards uniform “virtual machine API” across hypervisors
Prototype

- Implemented on Linux 5.3.1, kvmtool and Xen 4.12.1 HVM
- ~8.5 KLOC (90% userspace-based)
- InPlaceTP for Xen ↔ KVM, Xen → Xen
- MigrationTP for Xen → KVM
Saving guest devices

- Saving and restoring VM hardware state already possible in most hypervisors
  - Xen: reuse existing HVM state saving code
  - kvmtool: write saving code matching Xen HVM state format
- Incompatible devices (e.g. PV NICs): unplug device before transplanting
Saving guest memory (InPlaceTP)

- Saving guest memory to disk is too time consuming
- Keep guest memory in host RAM
- Manage VM memory in file table
  - Modified Vladimir Davydov’s PRAM patchset and in-memory format
  - Supports 4K and 2M pages
PRAM table construction

1. Allocate temporary memory
2. Relocate VM memory regions
3. Construct PRAM table

*Non-contiguous regions
PRAM in-memory format
Transplanting (InPlaceTP)

- Need to preserve memory contents
- Solution: Use Kexec
  - Fast boot without going through BIOS
  - Supported for Xen ↔ Linux transition
  - Keeps RAM relatively intact
- Pass PRAM table address through kernel parameter
- Compatibility fixes for Kexec with Xen ↔ KVM
## Restoring guest devices

- Modified kvmtool to read/write unified state:

<table>
<thead>
<tr>
<th>State</th>
<th>Xen</th>
<th>kvmtool</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU regs</td>
<td>CPU</td>
<td>KVM (S)REGS, MSRS</td>
</tr>
<tr>
<td>Local APIC</td>
<td>LAPIC</td>
<td>KVM MSRS</td>
</tr>
<tr>
<td>Local APIC regs</td>
<td>LAPIC_REGS</td>
<td>KVM LAPIC_REGS</td>
</tr>
<tr>
<td>MTRR</td>
<td>MTRR</td>
<td>KVM MSRS</td>
</tr>
<tr>
<td>XSAVE</td>
<td>XSAVE</td>
<td>KVM XCRS, XSAVE</td>
</tr>
<tr>
<td>IO-APIC</td>
<td>IOAPIC</td>
<td>KVM IRQCHIP</td>
</tr>
<tr>
<td>PIT</td>
<td>PIT</td>
<td>KVM PIT2</td>
</tr>
<tr>
<td>TSC</td>
<td>TSC</td>
<td>KVM TSC</td>
</tr>
<tr>
<td>RTC</td>
<td>RTC</td>
<td>kvmtool RTC</td>
</tr>
</tbody>
</table>
Restoring memory (InPlaceTP)

- Mount PRAM structure as virtual filesystem at boot time
- Each VM’s memory regions together represented as one file
- KVM: VM memory file directly mmap()ed
- Xen: Implement hypercall to assign memory file to restored domain
Evaluations: InPlaceTP Xen → KVM

- Two different machines: M1/M2
- Downtime: 1.70 seconds on M1; 3.01 seconds on M2

The VM is running after 1.7 sec and 3.01 sec

- Networking card takes additional time to restart
InPlaceTP (left) offers low VM disruption and acceptable VM downtime
MigrationTP (right) performs similarly to live migration: minimal downtime but long migration time comes with performance impact
Cluster-level upgrade times

- Used BtrPlace to simulate cluster upgrade event
- Cluster contains a mix of InPlaceTP-compatible and non-compatible VMs
- The more InPlaceTP-compatible VMs, the less time it takes to upgrade the cluster
Thank you for listening!