I Heard It through the Firewall: Exploiting Cloud Management Services as an Information Leakage Channel

Hyunwook Baek,
Eric Eide, Robert Ricci,
Jacobus Van der Merwe

University of Utah
Motivation

- Information leakage in cloud has concerned cloud users from the beginning of cloud computing.
- Existing cloud information leakage channels:
  - Cache [Ristenpart et al. 2009, Liu et al. 2015]
  - Memory [Zhang et al. 2011, Meltdown, Spectre]
  - Network device [Bates et al. 2012]

→ Hardware-level Shared Resources

- How about Software-level Shared Resources?
Motivation
Motivation

The two users’ requests shared:
- Processes
- Threads
- Variables
- Queues
- Execution paths
- ...
Goal

- Demonstrating exploitability of software-level shared resources as an information leakage channel
- Especially, focusing on Shared Execution Paths (i.e., cross-tenant batch-processing)
- Using OpenStack Network Management Service (similar mechanism can be applied to other systems)
Background: polling_interval

```python
def rpc_loop(self):
    while True:
        start = now()

        # update OVS changes
        # update Iptables changes
        # update conntrack changes

        elapsed = now() - start  # job_done
        if elapsed < polling_interval:
            sleep(polling_interval - elapsed)
```
Background: polling_interval

polling_interval (2 sec)
Basic Idea

- The `rpc_loop()` is shared by requests of VMs running in the host.
- The total size of the load of requests ∝ elapsed.

```
polling_interval (2 sec)

rpc_loop()  rpc_loop()

elapsed  sleep()

#job_done
```
Basic Idea

- Observing elapsed times to distinguish infrastructure level events – Side Channel
Basic Idea

- Manipulating elapsed times to send messages
  – Covert Channel
Problem

- Cloud users (and VMs) cannot directly observe the elapsed times.
- Something \(\approx\) elapsed and observable by users?
  \[\rightarrow\] Virtual Firewall Epoch
Epoch

```
rpc_loop()
rpc_loop()
rpc_loop()
```

```
iptables_restore
```

```
Epoch
Epoch
```

```
iptables_restore
```
Epoch

- Epoch \approx \max(\text{elapsed, polling\_interval})

```
 rpc_loop() rpc_loop() rpc_loop()
```

```
 elapsed elapsed elapsed
```

```
 iptables_restore iptables_restore iptables_restore
```

Epoch ▪ Epoch ▪ Epoch
Epoch

No security group is changed, so this loop does not execute `iptables_restore`

- Epoch ≠ elapsed if there is no change on the iptables.
Solution

- Observing *Epochs* to distinguish infrastructure level events – Side Channel
Solution

- Manipulating **Epochs** to send messages
  - Covert Channel
Epoch

- To monitor Epochs:
  1. The virtual firewall should be updated in every RPC loop iteration so that the Iptables is also updated.
  2. The update result should be observable by the attacker.
  3. The update request should have small impact on the elapsed to minimize noise.
Epoch

- To manipulate Epochs:
  1. There should be a request that can make a clearly distinguishable impact on elapsed.
  2. The request should be processed at the targeted RPC loop iteration.
Impact of Requests: One-time Impact

- Property 0) Some requests bring the same result but their load sizes are different
Impact of Requests: One-time Impact

- Property 1) Some requests introduce nearly no additional load

- Useful for monitoring Epochs
Impact of Requests: One-time Impact

- **Property 1)**  
  Some requests introduce nearly no additional load

- Useful for monitoring Epochs
Impact of Requests: One-time Impact

- Property 2) Some other requests introduce clearly distinguishable additional load

- Useful for manipulating Epochs
Impact of Requests: Long-term Impact

- Property 3) Some requests may permanently increase the loads of other requests.

- Useful for manipulating Epochs
Epoch Patterns
Monitoring Epoch: **UPDATE+PROBE**

**Update:** add a new rule to its virtual firewall.
E.g., Allow ICMP type:8 code:4 ingress

![Diagram showing the process of adding a new rule to a virtual firewall](image)
Monitoring Epoch: UPDATE+PROBE

Physical Machine  
| Neutron-OVS-agent  |  Allow p  
|-------------------|-----------
| Iptables          |           

Virtual Machine  
| Probe Monitor     |           

Cloud Controller  
| Request Sender    | Allow p   

Probe Sender  
|                  |           

ICMP type: 8 code: 4 ingress

Probe: generate a series of probe packets
Monitoring Epoch: UPDATE+PROBE

Estimated arrival time of dropped packet

$t_{up}(x_1)$

$t'(p_1^6)$  $t(p_1^7)$  $t(p_1^8)$  $t(p_1^9)$

$t_{req}(x_1)$  $t'_{up}(x_1)$

Probabilistic bound of iptables update time

Arrival time of the firstly arrived probe packet
Continuous Monitoring

- **Iterative UPDATE+PROBE method**
  - Monitoring modules are independent

- **Reactive UPDATE+PROBE method**
  - The number of requests: 1 / epoch

- **n-Reactive UPDATE+PROBE method**
  - can dynamically adjust the number of requests
Practical Epoch Monitor

- **EpochMonitor**
  - A stand-alone architecture for epoch monitoring.
  - Can easily support any of the previously introduced methods.
Deployment: Boomerang Packets

• Layer 3 Boomerang with Single Interfaces

srcMAC: VM-MAC
dstMAC: Router-MAC
srcIP: VM-IP
dstIP: VM-IP

srcMAC: VM-MAC
dstMAC: Router-MAC
srcIP: VM-IP
dstIP: VM-IP
Single-node Covert Channel

- **Covert Channel**
  - Both VMs keep monitoring the epochs using EpochMonitor.
  - SVM also reactively send message to RVM by manipulating the duration of epochs.
  - E.g., to send 0: do nothing
  - to send 1: attach/detach SG
Single-node Covert Channel – Evaluation

- Error rate: 0
- Bandwidth: 0.21 bps
Multi-node Covert Channel

- Covert Channel
  - SVM send message by sending the same message for \( n \) seconds.
  - This can be done by manipulating the duration of epoch of medium VMs, using the long-term impacting requests.
Multi-node Covert Channel – Evaluation

- Error rate: 0
- Bandwidth: 0.1 bps
Infrastructure Event Snooper

- Snooping on the host level events
- Any network-related requests can leave their mark on Epoch
- The attacker keep monitoring Epochs and extract event information
Infrastructure Event Snooper

- VM creation / termination
- # of virtual interfaces per VM
Infrastructure Event Snooper

- Continuously monitor Epochs
- Classify events using LSTM Model

Output:
  - If any VM was created / terminated during an Epoch
  - The number of virtual NIC attached to the VM
Infrastructure Event Snooper – Evaluation

- Training Data
  - Two types of Host Machines
  - Four types of VMs
    each of which has different # of virtual NIC
  - Two types of events: VM creation / VM termination
  - 100 data points for each class
  - 75% for training, 25% for validation
Infrastructure Event Snooper – Evaluation

- Test Data
  - For each different type of Host Machine
  - Created and terminated 100 VMs in a random order
  - Each VM was configured to have
    - random number of virtual NIC between 1 and 4
  - 478 labeled data points
### Infrastructure Event Snooper – Evaluation

#### Accuracy:

83.1%

<table>
<thead>
<tr>
<th>Ground Truth</th>
<th>VM Creation</th>
<th>VM Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>72</td>
<td>6</td>
</tr>
<tr>
<td>I</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>50</td>
<td>12</td>
</tr>
<tr>
<td>III</td>
<td>35</td>
<td>3</td>
</tr>
<tr>
<td>IV</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>VM Termination</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>II</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>III</td>
<td>9</td>
<td>34</td>
</tr>
<tr>
<td>IV</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

**Note:** The numbers in the table represent the count of classified events against ground truth categories.
## Infrastructure Event Snooper – Evaluation

### Accuracy:
- **83.1%**

### Accuracy ignoring vNIC:
- **93.3%**

<table>
<thead>
<tr>
<th>Ground Truth</th>
<th>VM Creation</th>
<th>VM Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>46</td>
<td>6</td>
</tr>
<tr>
<td>I</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>II</td>
<td>35</td>
<td>3</td>
</tr>
<tr>
<td>III</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IV</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VM Creation</th>
<th>VM Termination</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>31</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>4</td>
<td>34</td>
<td>10</td>
</tr>
<tr>
<td>III</td>
<td>1</td>
<td>33</td>
<td>4</td>
</tr>
<tr>
<td>IV</td>
<td>1</td>
<td>11</td>
<td>42</td>
</tr>
</tbody>
</table>
Evaluation – EpochMonitor

- Root Mean Square Error
  - 1.54 milliseconds

- Maximum Error
  - 25.5 milliseconds
  - Sufficient for distinguishing different requests
    (differences are larger than 100 milliseconds)
Mitigation – Refactoring

- Don’t use Cross-tenant Batch

```python
... 
req_batch = aggregate_requests() 
...
...
update_something(req_batch)  # observable event
...
```
Mitigation

- Increasing Polling Interval
  - Pros: simple and may work for some cases
  - Cons: increases the system delay by order of seconds

- Introducing Random Delay
  - The same as above...
Mitigation

- Rate Limiting (Request Delaying)
  - Request pattern is different from Dos-style attack
    - e.g., 0.5 request per second
  - If combined with a tailored policy, may effectively mitigate the probing.
    - e.g., if avg(# of requests for VM1 per sec) > 1 and std(# of requests for VM1 per sec) < 0.1:
      delay future requests by 5 seconds
Conclusion

- Showed software-level shared resources can be exploited as an information leakage channel.
- Designed covert / side channels exploiting shared execution paths.
- Demonstrated attacks using OpenStack Network Management Service.
Possible Application

- Cooperative co-residency detection
  - Detecting co-residency of the attacker’s own VMs.
  - A VM keeps sending detectable signal through the control plane (e.g., keep creating/deleting SG with many rules)
  - If another VM successfully co-reside with the VM, it can read the signal through the Update+Probe
  - Trivially doable
Possible Application

- Un-cooperative co-residency detection
  - Detecting co-residency with victim VMs.
  - E.g., when load increases, the auto-scaling service launches new VMs in the same physical machine (e.g., affinity group in OpenStack)
  - The attacker change the load on the victim VM and monitors Epochs to detect when VMs come/leave
Possible Application

- Infrastructure Profiling
  - E.g., a cloud provider launches large number of ‘spot instances’ in night time for specific type of machines.
  - E.g., a cloud provider launches ‘High-end VMs’ with large number of virtual interfaces only in specific types of machines.