Hardening OpenStack Cloud Platforms against Compute Node Compromises*

Wai Kit Sze
Stony Brook University
wsze@cs.stonybrook.edu

Abhinav Srivastava
AT&T Labs - Research
abhinav@research.att.com

R. Sekar
Stony Brook University
sekar@cs.stonybrook.edu

ABSTRACT

Infrastructure-as-a-Service (IaaS) clouds such as OpenStack consist of two kinds of nodes in their infrastructure: control nodes and compute nodes. While control nodes run all critical services, compute nodes host virtual machines of customers. Given the large number of compute nodes, and the fact that they are hosting VMs of (possibly malicious) customers, it is possible that some of the compute nodes may be compromised. This paper examines the impact of such a compromise.

We focus on OpenStack, a popular open-source cloud platform that is widely adopted. We show that attackers compromising a single compute node can extend their controls over the entire cloud infrastructure. They can then gain free access to resources that they have not paid for, or even bring down the whole cloud to affect all customers. This startling result stems from the cloud platform's misplaced trust, which does not match today's threats.

To overcome the weakness, we propose a new system, called SOS , for hardening OpenStack. SOS limits trust on compute nodes. SOS consists of a framework that can enforce a wide range of security policies. Specifically, we applied mandatory access control and capabilities to confine interactions among different components. Effective confinement policies are generated automatically. Furthermore, SOS requires no modifications to the OpenStack. This has allowed us to deploy SOS on multiple versions of OpenStack. Our experimental results demonstrate that SOS is scalable, incurs negligible overheads and offers strong protection.

CCS Concepts

•**Security and privacy** → *Intrusion detection systems; Distributed systems security;*

1. Introduction

Cloud platforms such as OpenStack [17] manage infrastructures that consist of two main types of nodes — *control nodes* and *compute nodes*. Control nodes manage resources for cloud providers and customers using well defined interfaces. Compute nodes run virtual machines (VMs) of customers, and constitute the bulk of the cloud infrastructure. Compute nodes host VMs prepared by cloud customers that cloud providers have no control over. These VMs can be controlled by attackers, and they in turn can exploit vulnerabilities in hypervisors. While hypervisor-breakout vulnerabilities are rare, they are being discovered almost every year since 2007 [11]. Indeed, hypervisor-escape vulnerability has been a major concern to the OpenStack security group [1], and has affected their OpenStack design [30].

In a cloud platform such as OpenStack, it is clear that the whole cloud infrastructure is compromised if attackers have control over control nodes. Since control nodes are small in quantity, cloud providers can harden their security by employing heavy protection mechanisms (such as taint-tracking approaches [22, 9]) at the cost of deploying a few more control nodes. On the other hand, the number of compute nodes can be several orders of magnitude larger, and hence securing all of them can be a challenge. We therefore posed the following question: *What is the extent of damage that attackers can cause when they have gained control over a single compute node?* We found that attackers can easily expand their control to the entire infrastructure. We present several attacks that enable attackers to control and access any VMs on any compute node, create or destroy resources of arbitrary cloud customers, or disable the entire cloud.

Attacks are possible because compute nodes are part of the TCB (Trusted Computing Base) in the OpenStack's design, which trusts compute nodes entirely. Attackers controlling a compute node can therefore exploit this trust. To address this weakness, we propose SOS (Secure OpenStack), a system to limit trust on compute nodes. SOS consists of a framework that can enforce a wide range of policies. Specifically, we applied mandatory access control (MAC) and capabilities to regulate interactions between compute nodes and controller nodes. This limits the damage that a compromised compute node can inflict.

One of the main challenges with security policies (such as SELinux [10]) is to develop effective policies against attacks while reducing false-positives. This is particularly challenging for OpenStack as it is an active project. Its modular design allows it to be deployed with different optional components. All of these factors make hand-crafting policies impractical. We have therefore developed an approach to generate policies automatically based on training and static analysis. Our goal is to minimize the false positives while remain effective against attacks.

Access control policies alone do not solve the confused

*This work was supported in part by NSF (grant and 1319137) and DARPA (contract FA8650-15-C-7561).

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

ISBN 978-1-4503-4233-9/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2897845.2897851

deputy problem [8], where an intermediary is fooled by an attacker into misusing its authority. In particular, compute nodes in OpenStack have a legitimate need to access customers' secret-tokens in order to manage resources on their behalves. Attackers controlling compute nodes can hence sniff the tokens and impersonate the customers. SOS prevents tokens from being abused by limiting privileges, forming a hierarchical structure similar to the capabilities in operating systems. Instead of giving the customer tokens to compute nodes, SOS gives compute nodes tokens with reduced privileges to limit what operations the compute nodes can perform. A major challenge is to identify what capabilities are needed for each request. SOS cannot rely on customers because they do not have details about the cloud infrastructures. Without specifying these details in capabilities, attackers can easily replay the tokens. SOS solves this problem by identifying and reducing token privileges progressively *within the infrastructure*. By the time tokens reach compute nodes, token privileges are limited to the point that they can only be used for their intended purpose.

To show the efficacy and generality of SOS , we have deployed it to protect three different releases of OpenStack: Havana [13], IceHouse [14], and Juno [15]. SOS incurs only a small overhead during resource provisioning, and no overhead at all when VMs are running.

1.1 Contributions

This paper makes the following contributions:

- *Investigation of possible attacks from compute nodes.* We present several classes of attack from a compromised compute node that enable an attacker to create VMs at the expense of other cloud customers, control, destroy, or inject backdoors into their VMs, or bring down the entire infrastructure.
- *Policy enforcement framework for OpenStack.* We propose a policy enforcement framework for OpenStack. The framework supports a wide range of policies, including capability-like policy, mandatory access control, isolation or information flow policies (e.g., Chinese-Wall, Bell and LaPadula, or Biba). The framework can also transparently alter OpenStack behaviors such as controlling how VMs are scheduled.
- *Capabilities for limiting trust on compute nodes.* SOS uses capability to confine interactions between compute nodes and control nodes. SOS also applies capabilities to reduce token privileges such that compute nodes cannot abuse stolen tokens.
- *Transparent and efficient design and implementation.* SOS requires no modifications to OpenStack, and hence is readily deployable to multiple OpenStack versions. SOS incurs negligible overhead and protects against all attacks presented in this work.

1.2 Paper organization

We begin with some background on OpenStack in Section 2. This lays the foundation for the attacks described in Section 3. Following this, we present the design of our policy enforcement framework in Section 4. The framework supports a wide range of policies. Specifically, we focus on security by illustrating how the framework can enforce a capability-like policy to protect intra-module communication. In Sections 5, we extend the framework to protect inter-module communication. Experimental evaluation is presented in Section 6 . Related work is discussed in Section 7, followed by concluding remarks in Section 8.

2. Background

OpenStack embraces a modular design, allowing different modules to plug-and-play. The most important OpenStack modules include `Keystone` for user authentication, `Nova` for managing compute resources, `Glance` for managing (disk) images, and `Neutron` for managing networking. Other modules like `Cinder` for block storage and `Ceilometer` for telemetry can also be installed to provide additional functionality. Figure 1 illustrates an OpenStack deployment that uses three modules — `Keystone`, `Nova`, and `Cinder`.

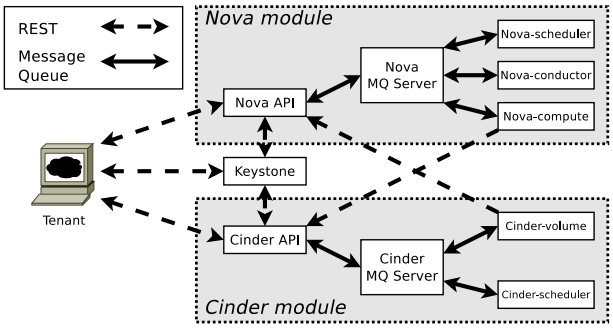

Figure 1: Intra-module interactions between services (RPC/Message Queue) and Inter-module interactions between modules (REST/HTTP)

2.1 Interactions between modules

Each OpenStack module provides one or more *services*. Services can either be exposed to customers and other modules, or internal to a module. Exposed services handle REST (REpresentational State Transfer) APIs (*inter-module communication*). Internal services communicate using RPCs (Remote Procedure Calls) (*intra-module communication*).

Inter-module communication.

Modules expose REST APIs for other modules or customers. Therefore, each module authenticates requests independently. Authentication relies on customer secrets, called tokens, presented along with the REST requests. A token is nothing but a string returned to customers by `Keystone` upon successful authentication. It can be some shared secret or information signed with public keys of `Keystone`. Anyone presenting a token will have all the privileges of the token owner.

Customers and services from different modules interacts with a module using exposed services, called API-services. API-services translate REST requests into requests to internal services. They serve as entry points of modules. Authentications are therefore performed by these API-services. Upon authenticated by API-services, internal services will not perform additional authentication.

Ideally, tokens can be distracted after authenticated with API-services. But this is not possible in general when a request involves multiple modules: Consider a create-VM request. This request will first reach `nova-api`, the API-service in `Nova`. Upon successful authentication, `nova-api` will invoke RPCs within the `Nova` module to create VMs. Following this, it may be necessary to interact with the `Cin-`

der and Neutron modules to provision storage and networking respectively. To access these services, Nova needs to store the token it received from the customer, and forward it to Cinder and Neutron.

Intra-module communication.

Services within a module communicate through RPCs over Advanced Message Queuing Protocol (AMQP). Each service has a set of procedures that other services can invoke. AMQP is a publisher/subscriber communication model: publishers send messages to a message-queue server. Based on the *routing-keys* specified on the messages, messages are routed, and copied if necessary, to different queues. Subscribers can create or listen to existing queues by specifying the messages that they are interested in, based on routing-keys.

A key advantage of using AMQP for RPC is scalability: publishers can invoke RPCs by specifying just the services they need without identifying a particular receiver. Multiple receivers can listen to the same queue to handle requests. This allows automatic load balancing, fault-tolerance and seamless scaling.

The flexibility of message-queue (MQ) comes at the cost of security. By using MQ as a RPC mechanism, services need to be able to send and receive messages to and from other services. AMQP does not support fine-grained access-control primitives: Once services are granted permissions to access a MQ server, they can send and receive any messages. This means *any service can invoke any arbitrary RPC or sniff messages intended for others.* We present in Section 3.2 some of the attacks based on this property.

3. Threat Model, Attacks, and Overview

3.1 Threat model for attacks

We consider software and services running in control nodes to be secure. Services such as Keystone, Nova-conductor, Nova-scheduler[1], and other API-services (e.g., Nova-api, Cinder-api) are all secure as they run inside control nodes.

In contrast, we assume that compute nodes are vulnerable. These vulnerabilities may be present in compute services and/or hypervisors. By exploiting these vulnerabilities, an attacker can compromise the compute node hosting his/her VM. The attacks presented in this section rely on the ability to compromise just one of the compute nodes. We assume that *once a compute node is compromised, the attacker has complete control over the node:* he/she can access (a) all credentials stored on the node, (b) all messages intended for the node, etc.

We assume that cloud providers configure their infrastructures according to the recommended security-practices [18], including encrypting network-traffic and applying SSL on message-queues so that compromised nodes cannot sniff or tamper with network packets intended for other nodes.

We also assume that compute nodes are configured with least privileges, i.e., they contain only the information that is strictly necessary for them to function. For example, database credentials are not stored in the compute nodes.

[1] Nova-conductor is a RPC service for compute node to update the database entries. It is introduced since the OpenStack Grizzly release to prevent compute nodes from directly accessing the database. Nova-scheduler is a service for deciding which compute nodes to run what VMs.

3.2 Attacks on OpenStack

Attack setup.

We deployed OpenStack on 4 machines using the 3-node architecture in [16], with a controller, a network, and two compute nodes: compute1 and compute2. (See Figure 6). We configured OpenStack with Keystone, Nova, Glance, Neutron, and Cinder. We used QEMU as the hypervisor. The testbed has two customers: *tenant1* and *tenant2*.

With the above threat model, we launched several attacks from a compromised compute node, compute1. Our attacks have been successful on multiple OpenStack versions: Havana, IceHouse, and Juno.

Sniffing tokens from message-queue.

In this attack, we sniffed tokens from the MQ-server to invoke REST requests. Each compute node stores its MQ credentials inside its OpenStack configuration files. With these credentials, we created a new queue inside the MQ-server with a wildcard routing-key "#". All messages were then copied to compute1. We then extracted customer credentials from the messages— tokens and customer-ids were located in message fields named _context_auth_token and _context_tenant.

With *tenant1*'s tokens, we constructed REST requests to list, create, delete, and modify *tenant1*'s resources (including VMs, volumes, networking, etc.). Note that we had only compromised compute1, but we were able to control resources managed by non-compromised nodes and non-Nova resources.

With this attack, attackers can procure resources that would get billed to another customer (*tenant1*). More alarming, we were also able to capture the cloud administrator tokens. This had allowed us to create another cloud administrator account, control and access resources of all customers.

Invoking arbitrary RPCs.

OpenStack realizes RPC using message-queue. Messages are self-describing key-value pairs represented using JSON (JavaScript Object Notation) format. The key method indicates the procedure to invoke. The message itself also contains the parameters for the procedure.

Anyone with the MQ credentials can send arbitrary messages. We crafted a message on compute1 with routing-key=compute.compute2 and method=reboot_instance, containing parameters referring to a VM on compute2. Without specifying any token in the message, we had successfully rebooted the VM.

Recall that tokens are authenticated only at API entry-points— There is no authentication beyond API-services. Hence, these crafted messages were considered as legit. Furthermore, internal services do not use tokens. We were therefore able to perform actions like *stop*, *pause*, *resume* or *rebuild* on any VMs. Furthermore, we had invoked the set_admin_password method to change the root-password of the VMs.

More interestingly, we were also able to invoke cloud administrator APIs such as those for migrating-VMs after simply setting the key _context_is_admin to true. This has allowed us to invoke admin-only methods *without* knowing an administrator token.

Manipulating RPC parameters.

Apart from invoking arbitrary RPCs, we also manipulated RPC parameters. This had allowed us to deny resources, as

well as hijack an ongoing operation.

Compute nodes update the status of VMs that they host and report resource utilization using `instance_update` and `compute_node_update` procedures. `Nova-conductor` handle these RPCs to update databases that track resource status. The databases serve as a directory service: `Nova-scheduler` uses the database to decide where to schedule new VMs.

We launched an attack to falsify VM status and resource utilization of any VMs or any compute nodes. Without specifying any tokens, we had successfully invoked `instance_update` from `compute1` to update status of *tenant1*'s VM running on `compute2`. `Nova-conductor` accepted the RPC and updated the database accordingly. The status of the VM was then marked as "deleting." Similarly, we invoked `compute_node_update` to claim that `compute1` has 100 VCPUs and 1000GB memory. As a result, `Nova-scheduler` scheduled most VMs to run on `compute1`.

3.3 SOS overview and threat model

The above attacks highlight the fact that in OpenStack, it is straightforward for attackers to expand their footprint once they control one compute node. Through these attacks, attackers can acquire additional resources at the expense of other tenants, launch denial-of-service attacks, cause billing-havoc, and possibly bring down the whole infrastructure.

The goal of SOS is to prevent attackers from expanding their footprint, and limit the scope of damage they can cause. However, SOS does not attempt to stop activities that take place entirely within the compromised compute node, e.g., creating new VMs on that node, destroying existing running VMs, mounting and unmounting volumes, and so on. These activities affect customers having resources on that node. In addition, SOS does not focus on protecting compute nodes from getting compromised. SOS also does not protect VMs from attacking another VMs within the same compute node. These are important but orthogonal problems to the ones addressed by SOS . Existing solutions such as SELinux and its extension SVirt [24] address these problems specifically. SOS focuses on protecting the infrastructure when these existing mechanisms are not deployed or failed (e.g., due to configuration errors).

In the rest of the paper, we present SOS to stop all these attacks. SOS consists of a policy enforcement framework that can enforce a wide range of policies. We present a capability-based solution to limit compute nodes' RPC interactions. We also proposed embedding privileges in tokens to confine computes' REST interactions. SOS mediates all interactions to and from compute nodes.

We assume that SOS runs inside control nodes. In order for SOS to function properly, we assume a benign cloud platform to begin with, and remains attack-free until the end of the training-phase. This allows SOS to generate policies for enforcement. Any cloud platform is thoroughly tested prior to deployment. We rely on this period to generate policies, thus avoiding the need to rely on well-behaved cloud tenants.

4. Policy Enforcement on Message-Queue

In this section, we describe the first component of SOS , a policy enforcement framework to limit trust on compute nodes. It can enforce policies to protect against the intra-module attacks described in Section 3.2. However, the framework can also enforce a wide range of policies.

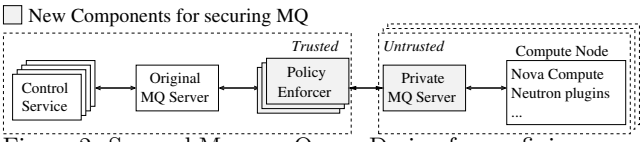

Figure 2: Secured Message Queue Design for confining compute nodes

4.1 Secure MQ architecture

Intra-module communication happens through RPC over AMQP. However, AMQP does not support fine-grained access-control to enforce useful policies at the RPC level. Any one with access to the message-queue (MQ) can send and receive any message, impersonate other nodes, and invoke any RPCs with arbitrary parameters.

The very first step of SOS is to identify publishers and perform filtering on messages. Since AMQP neither distinguishes publishers nor discriminates subscribers, SOS introduces a *MQ-proxy* for each compute node. MQ-proxy is a MQ-server with filtering capability. It consists of a private MQ-server and a policy enforcer. In this Secure MQ architecture (Figure 2), each compute node interacts with its own dedicated MQ-server, but not with the original MQ-server. SOS ensures this by replacing the original MQ credentials stored on the compute nodes with MQ-proxy credentials. Since the security of SOS does not depend upon the MQ-proxies, they can be located on any machine, including the compute nodes.

Messages in private MQ-server are isolated from other MQ-servers. SOS bridges the MQ-server and private MQ-servers with *policy enforcers*. These enforcers retrieve messages from private MQ-servers, examine the messages, and forward the messages to the original MQ-server only if the messages do not violate the enforcement policy. Policy enforcers also examine messages from the other direction to enforce stateful policies. Enforcers can run on dedicated machines or along with existing control nodes. Enforcers are designed to be scalable: SOS can introduce additional policy enforcers dynamically to scale up with message-queue traffic.

Policy enforcers retrieve messages from the MQ-server using routing-keys specific to a compute node. They subscribe to messages that a legitimate compute node would subscribe to. This ensures that *only messages intended for the compute node will reach its own MQ-proxy*. Although attackers can still create wildcard queues inside the private MQ-server, messages not intended for the compute node will never be accessible. Therefore, SOS is effective in preventing attackers from sniffing tokens that were not intended for the compute node.

This architecture acts as a framework to enforce RPC policies. It can enforce different type of policies transparently by altering and dropping RPC messages. For instance, it can enforce capability-like policy, mandatory access control (by controlling what nodes can send), isolation or information flow policies (e.g., Chinese-Wall, Bell and LaPadula, or Biba) to restrict communications between nodes. This framework can also transparently alter the OpenStack behaviors such as controlling how VMs are scheduled (e.g., running all VMs with same security labels on same com-

pute nodes). We focus our discussion on developing security policies to stop the attacks described in Section 3.2.

4.2 Security policies

SOS enforces multiple policies to limit trust on compute nodes and block the attacks. Each level aims to address a specific class of attack:

RPC procedure policy.

SOS classifies each RPC procedure into *callable* and *non-callable* by compute nodes. Callability of a procedure indicates whether compute nodes can legitimately invoke it. Policy enforcers only forward messages invoking callable procedures. Messages that invoke non-callable procedures are dropped and flagged as attacks.

SOS classifies all RPC procedures from both control and compute services. This ensures that compromised compute nodes cannot invoke arbitrary RPC procedures on any services. We have developed a tool to analyze OpenStack source code to extract the set of callable procedures. Table 1 shows a subset of the callable and non-callable procedures. Identifying compute procedures such as `terminate_instance` as non-callable has the effect of enforcing high-level policy such as "a compute node should not terminate VM on another compute node".

RPC parameter policy.

Blocking compute nodes from invoking non-callable procedures does not stop all attacks. Indeed, attackers can still launch attacks by simply modifying parameters in callable procedures. For example, compute nodes legitimately need to invoke reporting RPCs on control nodes to report their resource utilization and update VM status. Instead of faithfully reporting its own status, a malicious compute node can impersonate other compute nodes and/or falsify status.

SOS addresses this with *RPC parameter policy*. There are two types of parameters: static and dynamic. Static parameters do not change their values across RPC invocations. The main purpose of static parameters is for callee to identify callers. Recall messages in AMQP do not carry publisher information. This information is therefore encoded as a RPC parameter and hence subjected to manipulation. The impersonation attacks described above involve modifying static parameters. To detect these attacks, SOS generates policies based on training. SOS infers parameters as static if their values do not change during the training phase. In the enforcement phase, SOS makes sure static parameters have the same values as observed during the training.

Unlike static parameters, dynamic parameters can have different values across invocations. Callers use dynamic parameters to either (1) report values to callee (e.g., amount of free RAM or number of free VCPU) or (2) reference to resources (e.g., instance-uuid). For parameters that report values, SOS builds a data model to detect abnormal values. Currently, our prototype simply applies a simple range based approach to detect abnormalities. This works well in our testbed. Parameters that refer to existing resources raise more concerns: they appear to be random across RPC invocations. Attackers can replace them to manipulate resources owned by other compute nodes. RPC message itself does not contain sufficient information for SOS to decide if a referencing-resource parameter has been manipulated. SOS addresses these attacks with a higher-level policy described below.

Transactional policy.

Majority of the RPCs are triggered from API-services when handling requests from customers or other OpenStack modules. SOS considers all RPCs serving the same request as a *transaction*. RPCs belonging to the same transaction carry the same *request-id*.

Transactions are not random: They start with API-services or other control services invoking RPCs on compute nodes. SOS calls these RPCs *triggering RPCs*. With these triggering RPCs, compute nodes then invoke a subset of RPCs on control services to serve the requests. Instead of allowing compute nodes to invoke any callable procedures at anytime, SOS restricts compute nodes to invoke only a subset of callable procedures based on triggering RPCs. For example, a `terminate_instance` transaction would allow a different set of callable procedures than a `run_instance` transaction. Conceptually, the start of a transaction (a triggering RPC) grants compute nodes capabilities to invoke a subset of callable procedures. When the transaction ends, the capabilities will be revoked.

Similarly, RPC parameters within a transaction are not random: They concern with the same resource. As such, SOS considers the use of resource-referencing parameters as capabilities. When control nodes invoke triggering RPCs with references to a resource, the policy enforcer would update its internal state to allow compute nodes to invoke RPCs with these parameters inside the transaction.

Reducing the set of callable procedures alone works only when compute nodes serve one transaction at a time. When a compute node serves multiple concurrent transactions, it can easily aggregate the capabilities to invoke a superset of callable procedures and compromise the transactions. SOS maintains a one-to-one correspondence between resource-referencing parameters and transactions: compute nodes can only invoke callable procedures on specific resource-referencing parameters within a transaction. Similarly, a transaction can only reference to a set of resource-referencing parameters. Deviating from a transaction provides no additional access.

This policy requires transaction modeling. There are several techniques to model transactions. We can statically derive a set of callable procedures for each transaction. This is an over-approximation of the actual set because some callable RPCs would not be exercised due to different OpenStack deployment options. Alternatively, we can rely on training to get an under-approximation of the set. Our prototype relies on both training and static analysis to reduce noises in training data and provide higher confidence in flagging attacks. We evaluate in Section 9 the false positive rate of the approach.

4.3 MQ-proxy design and policy enforcer

Instead of modifying OpenStack to incorporate MQ-proxy, we reuse the existing message queue interfaces that OpenStack is already using. As such, configuring OpenStack with SOS simply involves replacing the original MQ-server credentials with the MQ-proxy credentials.

In our prototype, MQ-proxy is implemented using a dedicated MQ-server. Instead of creating a MQ-server for each compute node, SOS uses *virtual hosts*. Virtual hosts provide strong namespace isolation as if running multiple MQ-servers but with less resource overhead.

Callability	RPC Procedure
Callable	compute_node_update, get_by_compute_host, get_by_host, get_by_host_and_node, get_by_id, get_by_uuid, instance_update, ping, report_state, service_get_all_by, service_update, ...
Non-Callable	backup_instance, change_instance_metadata, check_can_live_migrate_destination, external_instance_event, get_console_output, get_host_uptime, get_vnc_console, inject_network_info, pause_instance, prep_resize, reboot_instance, rebuild_instance, reserve_block_device_name, reset_network, resume_instance, run_instance, shelve_instance, snapshot_instance, start_instance, stop_instance, suspend_instance, terminate_instance, unpause_instance, unshelve_instance, unshelve_instance, validate_console_port, ...

Table 1: Callable and Non-Callable RPC Procedures

Parameter Type	Policy	Examples
Static	Remain constant	Node id/ name, Hypervisor type, Node IP address
Dynamic (Value reporting)	Modeling data values	Number of free CPUs, Amount of free RAM, ...
Dynamic (Resource-Referencing)	Capability based on triggering RPCs	Instance-uuid, Volume-uuid, Image-uuid, ...

Table 2: Callable and Non-Callable RPC Procedures

5. Policy Enforcement on REST

In OpenStack, tokens authenticate cloud customers. Presenting tokens is sufficient to obtain full-access to resources in all modules. It is therefore important to protect the secrecy of tokens. However, OpenStack circulates tokens among all services, including services running on compute nodes.

The message-queue architecture described in Section 4 is effective in confining what RPCs compute nodes can invoke, and hence can stop compute nodes from launching attacks within a module. However, the architecture does not prevent compute nodes from abusing customer tokens through the REST interfaces (inter-module communication). MQ-proxy only reduces token-exposure to need-to-know compute-nodes. If these compute-nodes are compromised, attackers can steal the tokens and abuse privileges of the token owners.

This problem is challenging because compute nodes rely on tokens for inter-module-authentication. Without sending tokens to compute nodes, compute nodes cannot manage resources on behalf of customers.

Instead of removing tokens from compute nodes, SOS tackles this problem by constraining what tokens sent to compute nodes can do. SOS embraces the principle of least privilege to grant just enough privileges to compute nodes to complete customer requests, and nothing more.

5.1 Finer granularity token privileges

Tokens in OpenStack are designed for authentication. They do not carry any authorization information about what operations a request can perform. Anyone presenting a token will have full-privilege of the token owner to invoke any REST APIs. SOS introduces authorization to tokens by overloading them with privilege information. SOS breaks token privileges into finer-granularity such that SOS can give tokens with less privileges to less-trusted nodes. Tokens in SOS form a hierarchy as in Figure 3.

Token generation follows the standard rule: A token can only create equal or less-privileged token. As illustrated in Figure 3, a full-privileged customer token can create tokens concerning only volume operations, or specifically attach volume operation. SOS supports tokens at fine-grained levels down to specific resources.

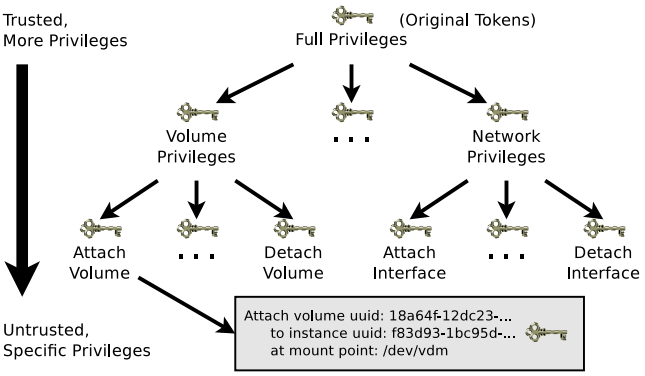

Figure 3: Hierarchical Token design in SOS , illustrating a token with specific privilege to attach volume.

5.2 Supporting fine-grained tokens in OpenStack

One of the challenge in applying fine-grained tokens in OpenStack is to identify what privileges a request needs. One approach is to let customers generate less-privileged tokens directly. This makes sense because customers know what requests they want to make. Unfortunately, cloud providers do not expose details about the cloud infrastructure for customers to generate least-privilege tokens. Consider a VM creation request. Customers know only the VM specification, but neither the new VM-uuid nor which compute node the new VM will run on. As a result, attackers stealing the token can replay the tokens to create additional VMs with the same specification.

Instead of identifying all privileges needed when making a request, SOS solves the challenge by generating less-privileged tokens progressively within the OpenStack. The key observation is that all resource assignment and allocation decisions are made by control nodes. Compute nodes do not make any decision. When control nodes invoke compute node RPCs to handle requests, RPC parameters already have all the information required to fulfill the request, including what resources the compute nodes need.

In Section 4.2 we discussed how SOS uses triggering RPCs (requests originating from control nodes) for granting capabilities for compute nodes to invoke callable procedures and reference to resources. The same idea can be used here. SOS considers the triggering RPCs as granting capabilities to invoke a certain set of REST APIs with specific parameters. Unlike in the secure MQ architecture where policy enforcers keep state information and mediate RPC requests from compute nodes, REST requests do not use message queue and hence capabilities have to be granted to compute nodes explicitly. Upon receiving a REST request, API-services will verify if the capability presented in the request matches with the request. Figure 4 illustrates how different components in SOS interact to protect REST interfaces.

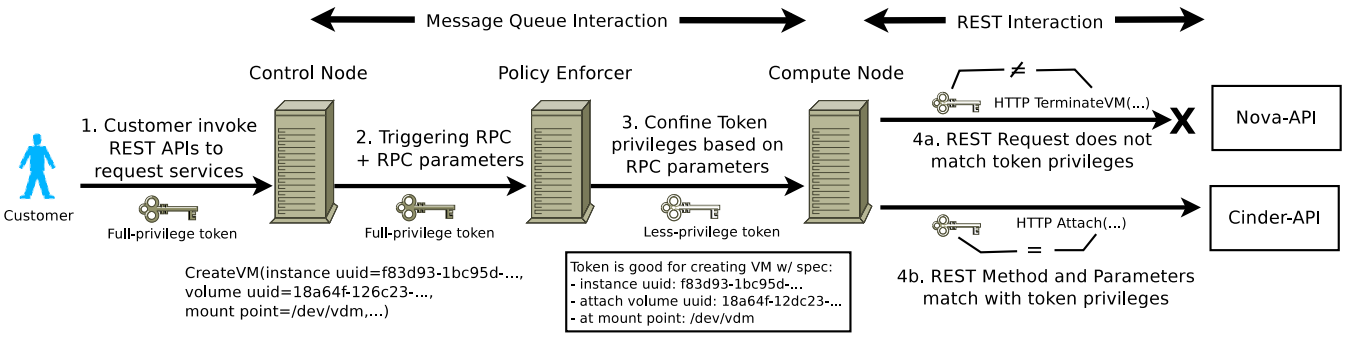

Figure 4: Generation of privilege-specific token from MQ to confine REST requests

5.3 REST API and parameter policy

There are two questions need to be solved in order to use capabilities to protect REST interfaces: (1) How to encode capabilities? (2) How to correlate triggering RPCs with REST requests?

Recall that triggering messages dictate what resources compute nodes need in order to fulfill the requests. Specifically, they encode the set of inter-project resources that compute nodes need access to. Naturally, SOS encodes triggering messages as fine-grained tokens. These tokens become the capabilities for API-services authorization.

To correlate triggering RPCs with REST requests, our prototype relies on both static and dynamic analysis. SOS statically identifies the sets of REST APIs (callable REST APIs) that each of the triggering RPC can lead to. This ensures that compute nodes can invoke REST APIs only when instructed by control nodes. However, attackers can still invoke callable REST APIs with arbitrary parameters. To protect against manipulating REST parameters, SOS checks if the REST parameters are authorized by the triggering RPC parameters. SOS relies on finding correlations between parameters. API-service will handle a REST request only if the request is a callable REST API and there exists a correlation between the REST parameters and the RPC parameters. These parameters are mainly resource uuid and hence SOS uses a string matching to correlate parameters. This works well in our experiments.

SOS can further optimize the process by avoiding encoding the entire triggering message. The purpose of encoding the triggering messages is to allow API-services to check for correlations between RPC messages and REST requests. If SOS can know what RPC parameters would be used in REST requests, SOS can simply encode these values. This knowledge can be derived based on training using (Algorithm 1). API-services receiving these "stripped" capabilities can first check if the REST parameters are authorized in the capabilities (Algorithm 2) (Fast Path). If not, SOS falls back to the original approach by performing correlation checking with the entire triggering message stored in a least recently used buffer (Slow Path).

5.4 Implementation

Instead of modifying the token generation mechanism in OpenStack to support hierarchy and encode privileges, SOS borrowed the idea of fat pointer to embed privilege information into the original customer tokens. Since both privilege information and original tokens are encoded inside fat tokens, SOS applies authenticated encryption to protect the secrecy of the original token and integrity of the privilege information. This design is stateless and avoided the need to modify OpenStack. Fat tokens contain all the authorization information. This allows efficient token authorization without referencing to databases to retrieve either the original tokens or the privilege information.

SOS uses the secure MQ-architecture in Section 4 to intercept triggering RPC messages. Policy enforcers replace original tokens with fat tokens. As tokens are not used once authenticated by API-services, fat tokens do not affect compute node operations. When compute nodes make REST requests, these fat tokens are sent along with the requests automatically for authorization in addition to authentication.

When handling REST requests, SOS needs to decrypt fat tokens and performs authorization. Instead of modifying API-services, SOS inserts additional components (paste-filters) into the REST request-handling pipeline by modifying the configuration files. API-services are already using these paste-filters to provide features like authentication, DoS prevention and supporting for multiple API versions. SOS inserts a paste-filter to decrypt fat tokens before authentication, and another paste-filter for authorization before the API-services serving the requests. Figure 5 illustrates the paste-filter pipeline in SOS . Note that customer

Algorithm 1: Identifying correlation between triggering RPC message and REST request

Input : REST Request R, Message M that triggered R
Result: Key-value pairs P_R for policy enforcement
$P_R \leftarrow \{\}$;
foreach *field* f_R *specifying resource in* R **do**
  foreach *field* f_M *specifying resource in* M **do**
    if $R[f_R] == M[f_M]$ **then**
      $P_R[f_R] = f_M$

Algorithm 2: REST API authorization checking (Fast Path)

Input : REST Request R, Message M that triggered R, Policy P_R for R
foreach *field* $f \in P_R$ **do**
  if $R[f] \neq M[P_R[f]]$ **then**
    goto SlowPathChecking;

tokens are not fat tokens. Therefore, customer tokens will fail at the decryption. SOS will simply continue with the pipeline for authentication. For tokens failed to decrypt but successfully authenticated by OpenStack, SOS will not enforce any policy.

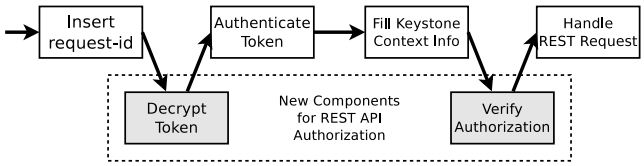

Figure 5: Pipeline arrangement in API-services

6. Evaluation

We deployed SOS on OpenStack Juno on 4 Dell PowerEdge 1750 servers. Each server has 4 Physical Intel(R) Xeon(TM) CPU 2.80GHz and 3 GB Ram. The testbed consists of a `controller` node, a `network` node, and two `compute` nodes. We deployed `Keystone`, `Nova`, `Neutron`, `Glance`, and `Cinder`. Figure 6 shows the services configured on each node. MQ-policy enforcers can be at any control nodes or dedicated machines. We deployed them at the `network`. For larger deployment, we can deploy it across multiple dedicated machines. The SOS -paste-filters are deployed along with the API-services at the `controller`.

SOS incurs overheads when examining messages and REST requests. This overhead is independent of the testbed size. Policy enforcers examine each message to and from compute nodes exactly once. SOS does not create new messages. Therefore, SOS adds a constant overhead per message. Similarly, SOS -paste-filters add a constant overhead for per REST request. In OpenStack, the number of RPC messages and REST requests is dominated by the number of provisioning operations, which depends on concurrency rather than the number of compute nodes in the testbed. We evaluate the performance impact of SOS on OpenStack by considering different concurrency level. These overhead measurements are expected to remain constant on different testbed size. Furthermore, SOS is designed to scale with OpenStack. Like many OpenStack services, cloud providers can always increase the number of policy enforcers and API-services to compensate for the overhead.

To run as many VMs as possible, we use *Cirros OS* as VM-image. Each VM has 1 VCPU and 32 MB memory. Other VM-images show similar results except they take longer time to spawn due to larger image size. Since the number of RPC messages and REST requests are independent of VM image, the overhead becomes less significant for larger VM-images.

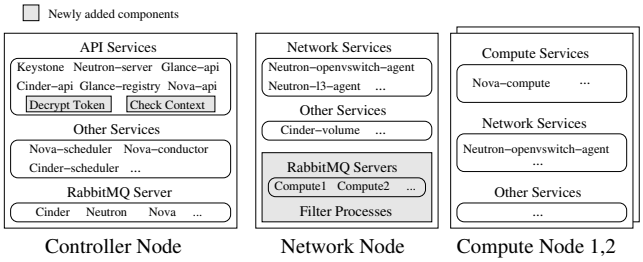

Figure 6: Experiment Setup

6.1 MicroBenchmarks

SOS uses paste-filters to validate token authorization at API-services. `Nova`, `Cinder`, `Glance` and `Neutron` API-services handle both fat tokens from compute nodes, as well as normal requests from cloud customers. In this benchmark, we study the overheads on REST requests. We used the OpenStack benchmarking tool *Rally* [6] to benchmark the time required to validate `Cinder` tokens using 1 `Cinder` API-service. We configured Rally with different number of concurrent token validation operations: 1, 5, 10, 20 and 40— each experiment lasted for 3600 seconds, with more than 10000 token validation operations completed in each experiment.

Table 3 shows the time required in SOS (Protected) compared to native OpenStack (Unprotected). SOS does not change the API-service latency characteristics of OpenStack across different concurrency levels.

Concurrency	1	5	10	20	40
Unprotected (s)	0.065	0.173	0.423	0.795	1.011
Protected (s)	0.071	0.176	0.430	0.802	1.013

Table 3: API-service token validation overhead

6.2 MacroBenchmarks

We also used Rally to evaluate the end-to-end overhead of SOS . We specifically looked at two predefined Rally *scenarios*: `NovaServers.snapshot_sever` and `CinderVolumes.create_and_attach_volume`. Each scenario consists of multiple operations. For example, `NovaServers.snapshot_sever` consists of *(a) booting a VM, (b) creating a snapshot, (c) destroying the VM, (d) booting a new VM from the snapshot, (e) destroy the new VM*, and *(f) deleting the snapshot.* We picked these two scenarios because they involve multiple resources (`nova`, `neutron`, `cinder` and `glance`) and common VM operations such as booting VM, attaching volume to VM or snapshotting VM. This can provide a good understanding about the performance impact of SOS at both message queue and REST request level.

We tested each scenario with concurrency level 1, 2, 4, 8, and 16. Concurrency corresponds to the number of simultaneous requests for performing the operations. Each experiment lasted 3600 seconds. Figure 7 and 8 show the results under concurrency level 1, 4, and 8 respectively. The bars cover the 25-percentile to 75-percentile of the measurements. The lines in the bar represent medians. The two ends of the bars covers the rest of the measurements. Dots represent outliers.

Again, SOS does not modify the latency behavior of OpenStack across different workloads or operations. Note that Rally focuses on VM-provisioning operations. Once an operation is completed, the subsequent operation will start immediately— The results do not consider the time to boot the guest-OS or the time to run tasks inside VMs. In reality, VM running times are significantly longer than the VM provisioning times. We can therefore consider the results presented here as the worst case performance for SOS .

While the graphs look similar across different concurrency levels, the actual time for performing the operations differ a lot— operations that took 10 seconds at no-concurrency could take 50 seconds to complete when concurrency is 8. As SOS scales linearly with the number of compute nodes, performance of SOS at a concurrency level 8 with 2 compute

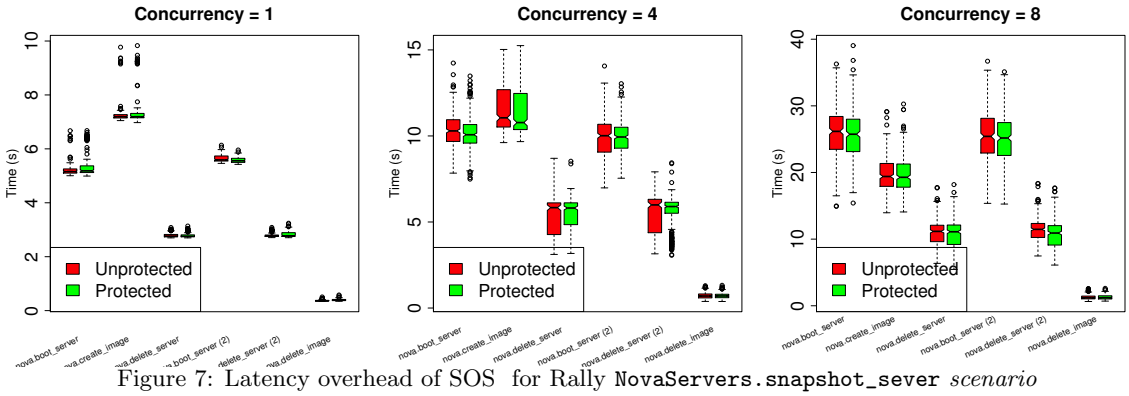

Figure 7: Latency overhead of SOS for Rally `NovaServers.snapshot_sever` *scenario*

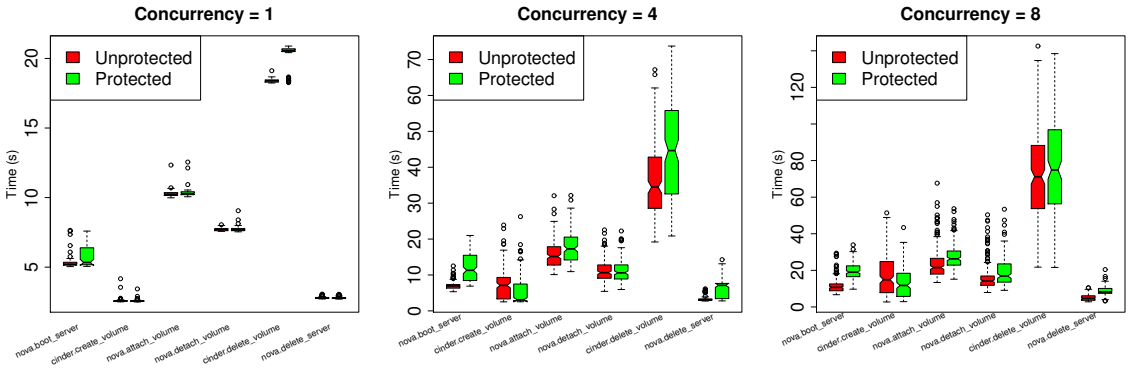

Figure 8: Latency overhead of SOS for Rally `CinderVolumes.create_and_attach_volume` *scenario*

nodes is equivalent to concurrency level 64 with 16 compute nodes.

Table 5 shows the number of scenarios completed in an hour. The system reached its maximum throughput at around 4 concurrent *operations*, with the `controller` being the bottleneck. Again, these are purely VM-provisioning operations.

6.3 Security analysis

We tested the attacks described in Section 3.2 on a SOS -protected deployment. SOS detected and stopped all attacks. Table 4 summarizes how SOS defeated the attacks. We generalize the attacks and discuss them below.

6.3.1 *Defending against attacks*

Sniffing tokens from message-queue.
SOS does not prevent compute nodes from declaring queues in private MQ-server. However, SOS ensures that compute nodes cannot sniff messages. The policy enforcers ensure that only messages intended for the compute node will reach its private MQ-server. As a result, attackers can learn nothing more than what the compute node already has access to by sniffing message-queue.

Invoking arbitrary RPCs.
OpenStack does not identify RPC callers. Attackers can therefore invoke RPCs that compute nodes are not supposed to invoke. SOS 's secure MQ architecture identifies RPC callers and enforce policies to constrain messages that compute nodes can send/receive. With the RPC Procedure Policy, SOS restricts compute nodes to only invoke RPCs that compute nodes can legitimately invoke. Non-callable RPCs such as `reboot_instance` are blocked.

Manipulating RPC parameters.
Attacks that modified node name and resources such as total number of CPUs on other compute nodes were defeated using static parameters. SOS makes sure that one compute node cannot call `compute_node_update` to update other compute nodes by ensuring that the compute node id specified in the message matches with the node's id. Attacks that reported falsified resource updates such as having 100 free VCPUs are also detected by SOS 's simple range based policy.

We also launched an attack which hijacked a transaction by replacing the VM-uuid to other VMs. SOS 's transactional policy and its resource-referencing parameters detected that the transaction referenced to parameters not authorized and reported as an attack.

SOS uses triggering RPCs to grant compute nodes capabilities to reference to resources. Without first obtaining a capability, attackers cannot reference to any resources. Attackers may trick control nodes to invoke triggering RPCs as to grant them capabilities, but triggering RPCs are always originated from control nodes and are not callable by compute nodes. Furthermore, attackers do not have the capabilities to reference to the resources in the first place.

Abusing tokens to invoke REST requests.
Compromised compute nodes can abuse tokens to perform operations not intended by cloud customers. SOS restricts the set of REST requests that compute nodes can invoke based on triggering RPCs parameters. This greatly reduces

Attacks	Protected by component	Policy
Sniffing RPC-messages	private MQ-server	Message-Queue Server Isolation (Need-to-know messages)
Invoking arbitrary RPCs	MQ Policy Enforcer	Callable RPCs (within transaction)
Manipulating RPC parameters	MQ Policy Enforcer	RPC Parameter Policy + Parameter-referencing Capability
Invoke arbitrary REST requests	REST authorization-validation	Callable REST + Parameter-referencing capability

Table 4: SOS defenses for attacks discussed in Section 3.2

Concurrency	1	2	4	8	16
`NovaServers.snapshot_sever`					
Unprotected	150	253	330	300	255
Protected	148	244	332	307	253
`CinderVolumes.create_and_attach_volume`					
Unprotected	77	129	178	191	169
Protected	73	111	147	167	150

Table 5: Number of scenarios completed in 1 hour

privileges of tokens that compute node can access. Any attempts to invoke different REST requests or with different parameters will result in inconsistency in the capability. API-services can therefore detect such attacks.

However, SOS does not protect tokens from being replayed to invoke the same REST requests with the same parameters. Attackers can continuously invoke the same REST requests with the same tokens until the tokens expire. As a result, attackers having a token for detaching a specific volume for a VM can prevents the VM from attaching the volume by keep invoking the detach volume REST request. SOS can address the problem by limiting the tracking token usage. Alternatively, SOS can also shorten the lifetime of OpenStack tokens.

6.3.2 *False-positives and false-negatives*

SOS uses static analysis or training to generate RPC transactional and REST request callable policies. Static analysis has the advantage of covering all possible code paths. It can model all potential RPC messages and REST requests that a legitimate deployment may occur. Therefore, it has low false-positive. However, not all code paths derived in static analysis can be exercised. So, it can lead high false-negatives. On the other hand, SOS can use training to generate deployment specific policies. It achieve good security by allowing only known benign messages and REST requests. However, the quality of the policies depend highly on the training data. It can lead to false-positives (due to insufficient coverage) or false-negatives (if attacks exist in the training data).

SOS can combine both approaches to generate policies. Static analysis can improve quality of training data by identify attacks in training data. Combining both approaches can also yield better confidence in identifying attacks: If a message is flagged as an attack by static analysis, it is very likely to be an attack. Similarly, a message accepted by training approach is likely to be legit. On the other hand, messages flagged as attack by the training approach but not by the static analysis could either be legit or real attacks. Cloud providers can decide on how to balance between false positives and false negatives.

Figure 9 shows the number of new policy violations observed in SOS based on training. We ran OpenStack Tempest [19], a test suite for OpenStack, to simulate legitimate usage behaviors. We do not consider static analysis here because it detected no violation. Based on the traces, most policy violations have been identified in the first 5 rounds of Tempest executions. If we generate policies using first 25 rounds of Tempest traces, we would have identified 950 policies and expected to see less than 10 violations in the following 20 rounds of Tempest traces. The number of new policy violations decreased substantially as more training data is used.

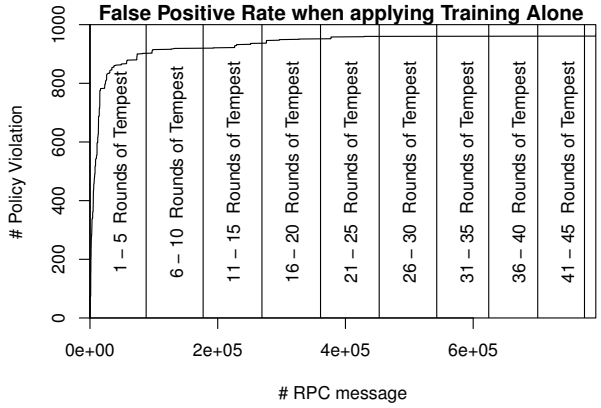

Figure 9: False Positive Rate when SOS applies only dynamic analysis (Based on Tempest [19] traces)

As SOS detected all attacks described in Section 3.2, SOS has no observed false-negative.

7. Related Work

7.1 Defenses in OpenStack

Although OpenStack includes `nova-compute` in the TCB, the OpenStack community has started developing several measures to limit the trust placed on its services. We summarize these measures below and identify their limitations.

Encrypting/Signing messages.

There is an OpenStack blueprint [12] that proposes to *sign and encrypt RPC messages*, preventing spoofing and sniffing of messages. Existing AMQPs already support using SSL to encrypt connections between services and message-queue servers. This blueprint requires two communicating services, say `nova-compute` and `nova-conductor` to share a key for signing and encrypting the communication between them. By doing so, attackers controlling one compute node cannot sniff messages meant for another compute node.

This blueprint yields the same effect as our MQ-proxy, which ensures compute nodes can only access to messages intended to them. As we demonstrated, attackers can send messages to update databases falsify their resources. This can increase the likelihood of becoming the intended message-recipients. By claiming they have tons of resources available, `nova-scheduler` is more likely to select the compromised compute node more often to run VMs, and thus receive the token information.

The second part of the proposal is to authenticate and protect message integrity with signing. At first glance, signing identifies request-origins to protect against spoofing. However, note that a compromised compute node does not have to spoof another compute node. Instead, all it wants to do is to make requests related to VMs on other compute nodes, or make resource requests on behalf of customers that did not initiate those requests. Signing alone does not prevent these attacks. What is needed is a more fine-grained policy to limit the requests and parameters that can be made by any given compute node.

Scoping tokens.

The basic management unit in OpenStack is a *project*. Any cloud customer of a project can control all resources (`cinder`, `nova`, `neutron`, etc) within that project. A cloud customer can be in multiple projects. Upon successful authentication of a customer, `keystone` returns a token that can access any resource in any project owned by that customer. As a result, leaking of the token can lead to compromise of resources across all the projects.

Scoped token is a simple extension that reduces the scope of tokens: instead of exposing the "master tokens" in every REST request, customers can scope tokens down to a single project by explicitly specifying the project. Leakage of a scoped token can therefore only compromise resources in that particular project.

Compromising a scoped token can still compromise all resources in a single project. One may further reduce the privileges associated with the token down to particular resources or OpenStack modules. However, this is hard to achieve because of the complex interactions between different OpenStack modules. It is not always possible to know in advance what modules will be involved to fulfill a given request. Furthermore, some tokens inherently cannot be scoped when they need to control resources across multiple projects.

SOS took the idea of scoped tokens to the extreme. SOS scopes tokens not only to a particular OpenStack module, but down to specific operations on specific resources. Instead of predicting what operations will be involved, SOS scopes down tokens within the OpenStack infrastructures. It supports much finer granularity privilege control and can support tokens that access resources across multiple projects.

7.2 Cloud security

Prior work in cloud security have focused providing security and privacy to cloud customers to protect their code, data, and computation [25, 2, 3]. Some focused on protecting secrecy or integrity of VMs against other VMs running on the same compute node, or against the cloud provider who have complete control over the underlying hypervisor and hardware [32, 20, 31]. Self-service cloud [4] proposes an architecture to redesign hypervisor, giving more controls to cloud customers to manage their VMs. Cloud Verifier [21] (CV) focuses on integrity of VMs. Users can specify integrity criteria that a communicating VM must satisfy. CV will then continuously monitor the integrity of the VM on behalf of the users. Communication to the VM is interrupted immediately when a violation of the criteria is detected. Srivastava et al. [26] created a service that generates trusted VM snapshot in the cloud, assisting cloud customers to check for the presence of malware.

While there has been some work done showing the problems with cloud infrastructures and how attackers can take advantage of it [5, 28], there is not much work done on protecting the cloud infrastructure. SCOS [27] addresses the problem that services are vulnerable. They studied various MAC mechanisms to confine cloud services at node level and local system level. They also designed an architecture to enforce MAC policies to protect services. However, details about SCOS has not been developed yet. While both our work and SCOS aim to protect the cloud infrastructure, we have a different threat model. We assumed controller services are trusted and focused on restricting compute nodes to prevent damages.

7.3 Host based security

Our secure MQ architecture provides a framework for enforcing various types of policy. It can enforce mandatory access control to confine compromised nodes. MAC has been well studied on end host systems. SELinux [10] can enforce sophisticated policies to confine applications running inside a machine. Instead of focusing on end hosts, our framework confines interactions between cloud services. We studied how various services interact and proposed an enforcement framework to confine compute nodes.

A majority of the service interactions happen through RPCs over message queues. RPCs are similar to function calls and a lot of work has been done on system-call based confinement [7, 23]. Many techniques from the system-call defenses are applicable to our system, too. Our system can rely on automaton to detect anomalous behavior as in [23]. With a framework to enforce polices, the question is what policies to enforce? Our system does share the same problem as system-call based defenses on generating effective policy against attacks. To solve the problem, we both exploit invariants in not only the set of methods/system calls that can be invoked, but also the parameters. Effective policies against mimicry attacks [29] are generated by correlating RPC messages and REST requests with triggering RPCs.

7.4 Distributed system security

Our hierarchical privilege for tokens shares some spirits with Kerberos. In Kerberos, users successfully authenticated with an Authentication Server (AS) will receive a ticket. Users can then present the ticket to a Ticket-Granting Server (TGS) to get another ticket which is good for a service. Users can then present the ticket to the server and get services. In our design, `Keystone` serves as the AS which generates a full-privileged token for authenticated users. Users can then present the token to API services (similar to TGS), which then generates another tokens that are good for specific API requests. However, there is an important difference: The goal of Kerberos is scalability. Users do not need to request for another ticket as long as all requests are in the same service session. On the other hand, our goal is to achieve fine grained access control, granting only the accesses a compute node needs to fulfill a customer request. Hence, our goal is to prevent reusing a token. These two are conflicting goals. The more fine-grained control it is, the less likely a ticket/token can be reused.

8. Conclusion

In this paper, we challenge the trust assumptions that OpenStack placed on compute nodes. We show that attackers compromising a single compute node can spread their footprints to bring down the entire infrastructure, or abuse

the resources in the infrastructure. To solve this problem, we proposed SOS . SOS consists of a secure message-queue architecture that can enforce a wide range of policy. We developed capability-based policies to confine trusts on compute nodes at both intra-project and inter-project communication. SOS enforces policies without requiring any changes to OpenStack. It supports multiple OpenStack version, protects against all the attacks discussed in the paper with little overhead.

9. References

[1] Openstack-security mailing list, http://lists.openstack.org/cgi-bin/mailman/listinfo/openstack-security.

[2] Bleikertz, S., Bugiel, S., Ideler, H., Nürnberger, S., and Sadeghi, A.-R. Client-controlled Cryptography-as-a-service in the Cloud. In *ACNS* (2013).

[3] Brown, A., and Chase, J. S. Trusted Platform-as-a-service: A Foundation for Trustworthy Cloud-hosted Applications. In *CCSW* (2011).

[4] Butt, S., Lagar-Cavilla, H. A., Srivastava, A., and Ganapathy, V. Self-service Cloud Computing. In *CCS* (2012).

[5] Farley, B., Juels, A., Varadarajan, V., Ristenpart, T., Bowers, K. D., and Swift, M. M. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *SoCC* (2012).

[6] Foundation, O. Rally - OpenStack, , https://wiki.openstack.org/wiki/Rally.

[7] Giffin, J. T., Dagon, D., Jha, S., Lee, W., and Miller, B. P. Environment-Sensitive Intrusion Detection. In *RAID* (2006).

[8] Hardy, N. The Confused Deputy: (or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev. 22*, 4 (Oct. 1988).

[9] Le, M., Stavrou, A., and Kang, B. B. Doubleguard: Detecting Intrusions in Multitier Web Applications. In *TDSC* (2012).

[10] Loscocco, P., and Smalley, S. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Ottawa Linux symposium* (2001).

[11] Luft, M. Analysis of Hypervisor Breakouts, http://www.insinuator.net/2013/05/analysis-of-hypervisor-breakouts/. Online; accessed November 13, 2014.

[12] OpenStack Foundation. MessageSecurity - OpenStack, https://wiki.openstack.org/wiki/MessageSecurity. Online; accessed November 13, 2014.

[13] OpenStack Foundation. OpenStack Havana Release - OpenStack Open Source Cloud Computing Software, http://www.openstack.org/software/havana/. Online; accessed November 13, 2014.

[14] OpenStack Foundation. OpenStack Icehouse Release - OpenStack Open Source Cloud Computing Software, http://www.openstack.org/software/icehouse/. Online; accessed November 13, 2014.

[15] OpenStack Foundation. OpenStack Juno Release - OpenStack Open Source Cloud Computing Software, http://www.openstack.org/software/juno/. Online; accessed February 29, 2016.

[16] OpenStack Foundation. OpenStack Networking (neutron), http://docs.openstack.org/icehouse/install-guide/install/apt/content/basics-networking-neutron.html. Online; accessed November 13, 2014.

[17] OpenStack Foundation. OpenStack Open Source Cloud Computing Software, http://www.openstack.org/. Online; accessed November 13, 2014.

[18] OpenStack Foundation. OpenStack Security Guide, http://docs.openstack.org/security-guide/security-guide.pdf. Online; accessed August 4, 2014.

[19] OpenStack Foundation. Tempest Testing Project, http://docs.openstack.org/developer/tempest/. Online; accessed November 13, 2014.

[20] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *CCS* (2009).

[21] Schiffman, J., Sun, Y., Vijayakumar, H., and Jaeger, T. Cloud Verifier: Verifiable Auditing Service for IaaS Clouds. In *SERVICES* (2013).

[22] Sekar, R. An Efficient Black-box Technique for Defeating Web Application Attacks. In *NDSS* (2009).

[23] Sekar, R., Bendre, M., Dhurjati, D., and Bollineni, P. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *S&P* (2001).

[24] SELinux Project. SVirt - SELinux Wiki, http://selinuxproject.org/page/SVirt. Online; accessed November 6, 2014.

[25] Srivastava, A., and Ganapathy, V. Towards a Richer Model of Cloud App Markets. In *CCSW* (2012).

[26] Srivastava, A., Raj, H., Giffin, J., and England, P. Trusted VM Snapshots in Untrusted Cloud Infrastructures. In *RAID* (2012).

[27] Sun, Y., Petracca, G., and Jaeger, T. Inevitable Failure: The Flawed Trust Assumption in the Cloud. In *CCSW* (2014).

[28] Varadarajan, V., Kooburat, T., Farley, B., Ristenpart, T., and Swift, M. M. Resource-freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *CCS* (2012).

[29] Wagner, D., and Soto, P. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *CCS* (2002).

[30] Yun Mao. Understanding nova-conductor in OpenStack Nova, http://cloudystuffhappens.blogspot.com/2013/04/understanding-nova-conductor-in.html. Online; accessed May 13, 2015.

[31] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *S&P* (2011).

[32] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS* (2012).