Security in CORD

Larry Peterson and Sapan Bhatia
Open Networking Lab

May 24, 2016

Introduction

CORD adopts a service-centric architecture—functionality is organized as a collection of services, with applications built by composing those services—but it does so in a way that adheres to best practices in building a secure system. This note outlines the key aspects of CORD that contribute to its security model.

Adopting a service-centric architecture is a good first step in building a secure system. This is because a system constructed from fine-grain components naturally lends itself to minimizing the trusted code base. But CORD goes beyond this foundation to also (1) explicitly mediate trust, and (2) consistently apply the principle of least privilege. Both are a consequence of layering a common *service control plane* on top of a collection of micro-services, and in doing so, provides explicit support for multiple domains of trust.

Our observation is that every service has security features implemented in a component-specific way, but implementing a coherent strategy across a portfolio of services involves intelligently composing these policies. Naively concatenating the policies of two services does not necessarily result in a coherent global policy. The service control plane must account for potential conflicts and dependencies.

Service Control Plane

XOS is the component of CORD that implements the service control plane, thereby providing a means to express and enforce policies on a collection of services. XOS defines a concrete representation for such policies. It includes a language and a runtime system for writing service control programs (policy statements) that one can reason

about individually, and when combined with other such programs, makes it possible to reason about the system as a whole.

The XOS service control plane consists of a core framework and a collection of service-specific plugins called *Service Controllers*. The core framework includes a declarative *Data Model* that defines the authoritative state of the control plane, an interface that can be used to "program" the control plane, and a *Synchronizer* that keeps the operational state of the underlying services in sync with the CORD's authoritative state. (In a reference implementation of CORD, the data model is implemented in Django, there is both a RESTful and a TOSCA-based programmatic interface, and the Synchronizer leverages Ansible [1, 2].)

Each Service Controller, in turn, has two parts. The first is a model that defines the authoritative state for the service (this service-specific model extends the core data model). The second is an imperative program that synchronizes this state with the scalable set of instances that implement the service (this program adheres to a prescribed template that includes four methods: *sync_record*, *delete_record*, *check_sync* and *check_delete*).

Taken as a whole, the Service Controller represents the "control" aspects of the service's API, including how to provision the service and what it means to acquire tenancy in the service. XOS assumes the rest of the service implementation—a scalable set of service instances that implement the service's "data plane"—is essentially the same as in any micro-services architecture.

The CORD data model implemented in XOS defines a global resource container, called a *Slice*, that hosts the service instances, where a Slice consists of a set of *Compute Instances* interconnected by a set of *Virtual Networks.* (In the reference implementation, Compute Instances are implemented as either Docker containers or KVM virtual machines, and the Virtual Networks are implemented by a control application called VTN running on ONOS [3].)

Security Implications

The XOS-provided service control plane is the cornerstone of CORD's approach to security. It prescribes how individual services are organized so they are able to plug into that framework, and in doing so, applies two best practices of secure information systems to CORD: (1) it makes it possible to mediate trust, including the ability to verify a chain of trust through a sequence of components, and (2) it enables a design that requires least privilege, including the ability to support a fine-grain separation of privilege.

1. XOS supports minimizing the trusted code base by running as many services as possible in isolated IaaS-provided *Slices*. This is in contrast to “monolithic” cloud platforms like OpenStack, where many of the management subsystems run as privileged processes on a management node (e.g., OpenStack head node).

2. XOS mediates trust by requiring all service-to-service control operations (e.g., one service acquiring tenancy in another service) and all operator-to-service control operations (e.g., the operator specifying that a service should scale up or down) to pass through a logically centralized XOS control point, where the security policy is enforced.

3. XOS supports least privilege on the control plane by providing an extensible role-based access control mechanism. In contrast to clouds that distinguish only between “operator” and “tenant”, XOS associates fine-grained privileges with a range of intermediate roles, including global (root) operators, infrastructure-specific operators, service-specific operators, service developers, and service tenants (including both end-users and other services).

4. XOS supports least privilege on the data plane by allowing services to control the network(s) through which their instances are accessed. Unlike commodity clouds in which users access services through the public Internet, XOS allows services to restrict access by their tenants (other services) to private “access” networks. This is in addition to purely private networks used by peer instances *within* a service. XOS enforces this restriction in the control plane by mediating what instances can connect to which virtual networks, but XOS does not interject itself on the data plane.

5. XOS makes it possible to anchor a chain of trust by providing a *secure boot* mechanism that services can use to distribute the private keys needed by their instances. This mechanism, which leverages a cryptographically secure blockchain library configured into each instance image, frees each service from building an its own key distribution mechanism.

6. XOS makes it possible to verify an end-to-end chain of trust by modeling all services as multi-tenant, with tenants corresponding to either authenticated users or other services. By isolating tenants from each other, it possible to follow a sequence of tenancy relationships through the collection of micro-services configured into the system.

That XOS provides this support with an explicit and programmable framework is important for two reasons. First, security is not a one-off, but it is a perpetual concern.

Security policies can be expected to evolve over time, and so there is value in a unified framework in which security policies can be expressed as service control programs, and these programs can be updated and evolved as new security considerations arise. Hardcoding such policies in individual components is not a practical alternative.

Second, providing this support in a shared framework, rather than on a component by component basis, makes it possible to address shared problems in one place. This is true for verifying an end-to-end chain of trust (as outlined above), but also in providing explicit and formal error recovery. Specifically, the Synchronizer handles errors as first class events and recovers from them gracefully. Therefore, errors cannot accumulate, which also means they do not dangle as anomalous insecure states that attackers can exploit.

Finally, while XOS provides mechanisms to support best practices in building secure systems, it does not strictly force operators to follow these practices. For example, XOS mediates trust that flows through the service control plane, but it does not require operators to establish more than one domain of trust. Similarly, XOS supports fine-grained privileges and preserves tenant identify across a chain of services, but it does not prescribe what roles or tenant abstractions services should define, nor does it dictate a particular implementation strategy for isolating tenants within a service. Finally, XOS supports inter-service access networks that are more restrictive than the public Internet, but it does not prohibit services from granting access to any client that connects through a publically routable network interface.

References

[1] CORD Reference implementation. *CORD Design Notes* (March 2016).

[2] Service Assembly and Composition in CORD. *CORD Design Notes* (February 2016).

[3] CORD Fabric, Overlay Virtualization, and Service Composition. *CORD Design Notes* (March 2016).