Technical Overview

Helping DevOps meet rapidly changing audit, security, and infrastructure needs.

Executive Summery 3

Why Conjur? 3

Use Cases 4

SSH Access Management 4

Secrets Management 4

Service-to-Service Authorization 5

Architecture and Deployment 6

Architecture Overview 6

Conjur Server 6

Encryption 6

Authentication 7

Audit 7

Authorization 7

High-Availability Design 8

Platforms Supported 9

Client Tools 9

What Is Conjur?

I. Executive Summary

Conjur is a cloud-native platform for directory services, authorization, and audit for development and operations teams and their entire infrastructure.

With 100% API coverage and a scalable, easily deployable, high-availability architecture, Conjur reduces the time, cost, and complexity associated with building authorization management compared to complex homegrown scripts and configuration management tools. Sometimes referred to as "Active Directory for the cloud", Conjur runs in either a virtual machine or container, and works alongside a wide range identity and access management (IAM) solutions to solve access and authorization challenges: machine-to-machine permissions, deployment and access rights to sensitive systems, and the auditing required to meet compliance requirements.

"Conjur is more than just an outstanding platform for managing authorization as a service: they are a partner that we can innovate with, and an essential part of achieving our vision and solving the hard problems that arise as we move our infrastructure into the cloud technologies that will drive our business forward."
- Mike Kail, VP of Operations, Netflix

Why Conjur?

Innovation moves fast. Systems are created, deprovisioned, and changed in real time; Conjur helps organizations keep up with this rapidly shifting landscape, adapting to and mitigating security risks and meeting compliance requirements without slowing down their continuous integration workflows, and keeping secrets, keys, certificates, and auth data out of repositories, off of hard drives, and secure.

Built with system administrators, DevOps professionals, and cloud architects in mind, Conjur natively supports a wide range of potential use cases. Some of the most common reasons why organizations select Conjur over building in-house solutions include:

- **Compliance**: auditable enforcement of both organizational and regulatory policies and rules, via existing reporting systems (e.g., SIEM platforms, SumoLogic, Splunk)
- **Risk Management**: reduction of the attack surface for sensitive data (credentials, SSL/SSH keys and certificates, secrets, etc.) by means of Conjur's policy and governance platform
- **DevOps Optimization**: integration of security & controls in upstream development and operations work, ensuring consistency across all production systems.
- **Access Intelligence**: unified control of identity (human & machine) and permissions across entire infrastructure (bare metal, private, public, cloud) helps prevent failures, without relying on legacy systems for policy governance and enforcement

II. Common Use Cases

While Conjur can be used for a wide variety of applications, three stand out as the most common. They are:

SSH access management

Conjur provides powerful, easy, standards-based SSH access to cloud servers and VMs and provides both authentication and authorization of SSH login.

• **authentication** by public key. Public keys do not need to be physically copied or otherwise distributed to each server or VM. Conjur makes the public keys available to SSH dynamically at login time

• **authorization** once an SSH connection is authenticated, it is then authorized. Authorization is a separate step from authentication, during which the login system determines if the authenticated user should be granted login access, and what their access level (via group membership) should be

Both authentication and authorization are performed in real-time, and both access grants and revocations take place immediately.

The server to which SSH access is granted is intended to be provisioned to use Conjur as an authentication and authorization provider, and must be assigned a "host" identity in Conjur.

In order to SSH to the server, a user needs to:

1. Have an associated Conjur user identity with registered personal public key. Note that no system user should be created in advance in target environment

2. Use an appropriate personal SSH key, with matches the registered public key

3. Be allowed to access the target host with appropriate level of privileges, typically assigned through group membership

Depending on Conjur permissions for the membership group, after successful login, user may have privileged ("sudo") or non-privileged access to the target system.

All authorization events on target system will be automatically sent to the Conjur and become part of the audit trail.

More details can be discovered at http://developer.conjur.net/tutorials/ssh

Secrets management

Automation scenarios, especially Continuous Integration and Deployment, often unavoidably include manipulation of secret data (encryption keys, api keys, passwords, etc.) The traditional approach is to keep secrets either in code repositories, configuration management tools, or inside of files on the server's filesystem.

These choices introduce two primary security risks:

• Secrets can be easily compromised via access to server filesystem or its snapshot

• It is difficult to track who accessed the secret data, and often cumbersome to implement fine-grained permissions control over it

Conjur addresses those issues, providing a solution for centralized manipulation of secrets, with comprehensive audit trail. Each secret is stored in an encrypted access-controlled container on Conjur server. The typical workflow for doing so is as follows:

1. A Conjur user who is authorized to manage particular secret stores "secrets" in the appropriate container – Conjur supports a wide variety of container types, from binary to ASCII text

2. The Conjur host authorized to fetch particular secrets for automation purposes uses the "conjur env" tool to obtain the relevant values from these containers, and then makes them available to the calling script via environment variables and/or temporary files, minimizing risk of their leak via persistent filesystem storage and/or execution logs

3. Where relevant, a Conjur user may be allowed to fetch particular secret, and can do so explicitly using the Conjur CLI toolset

4. (Optionally) Conjur host allowed to update secrets can run custom code to perform their rotation

All operations on secrets are recorded in the audit trail; more details can be discovered at http://developer.conjur.net/tutorials/secrets

Service-to-service authorization

In a service-oriented architecture it is often important to restrict some services from talking to others, and compliance requirements can require the ability to perform audits of both successful and unsuccessful service-to-service interactions.

Both needs can be met by way of Conjur's Role-Based Access Control (RBAC) capabilities:

1. Any service instance that needs to call other services should be assigned a Conjur host identity

2. The permissions model, presumably externally established, should be reviewed to ensure that it describes which hosts/layers are allowed to talk to which hosts/layers

3. Outgoing service calls should be performed through the authentication proxy (to which the Conjur authentication header can be added)

4. Protected services should be mapped to the authorization proxy, which in turn can recognize an authentication token in an incoming request, and perform a permission check via the Conjur Server to decide whether the request should be allowed

As a proof of concept, the following nginx plugins are available:

https://github.com/conjurdemos/forward-proxy-nginx -- authentication proxy

https://github.com/conjurdemos/service-to-service-nginx-lua -- authorization proxy

III. Architecture and Deployment

Conjur's design and architecture is built with two primary goals: (1) to provide an easy-to-use, well documented, extensible authorization system, and (2) to allow for non-disruptive integration into organizational workflows. This approach has resulted in a series of specific implementation benefits:

• **Ease of Integration:** Conjur is built with a DevOps optimized UX (CLI-driven), with 100% API access from a variety of languages, and can be deployed as a Linux virtual appliance, or into bare-metal systems

• **No Vendor Lock-In:** Conjur has been used in conjunction with all leading configuration management and DevOps tools (Puppet, Chef, Salt, Docker, etc.)

• **HA Design:** Conjur can natively support both high availability and fault tolerance, with a distributed configuration architecture and full fail-over/redundancy capabilities

• **Minimal Dependencies:** Conjur is self-contained, requiring no external database or server requirements

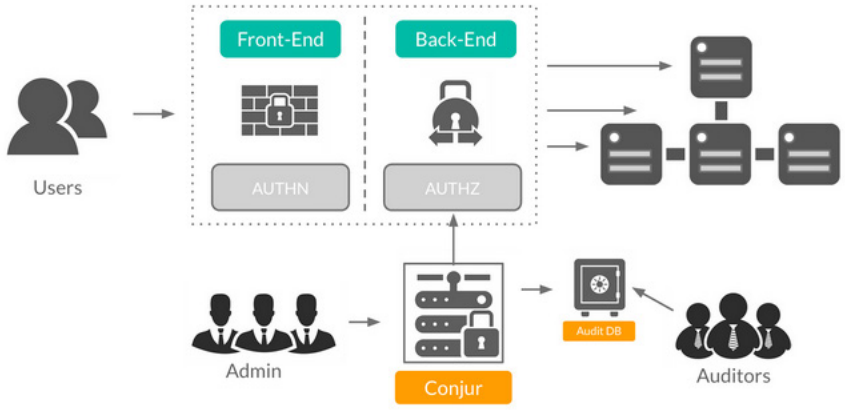
• **Secure:** All communication is fully encrypted, following industry best practices for managing data in use and at rest; additionally, Conjur has invested in a rigorous third-party security analysis (results expected to be published Q4 FY14)

IV. Architecture Overview

Conjur server

A basic Conjur installation consists of a single server, which operates on a local Postgres database.

It exposes RESTful API via HTTPS (for authentication, permissions management, permissions checks, manipulation of secrets and access to audit trail), alongside legacy LDAPS support, both of which expose directory capabilities (such as bind and search) to facilitate utilization of Conjur by third-party systems which can be easily configured to work with LDAP.



Encryption

All sensitive client data saved in Conjur is encrypted at rest, and cannot be accessed without the master key, which is stored separately from DB. All communication with clients happens over SSL.

Note: it is possible to use either custom certificate or self-signed certificate generated automatically on server installation.

Authentication

Conjur supports not only human identities ("users") but also identities for virtual machines, processes, and jobs (non-human identities are called "hosts" across the documentation).

Almost all operations (with the exception of "login" and "read access" to user public keys) require a valid, Conjur-specific authentication token, which is an expirable, cryptographically signed evidence of a Conjur identity. More details are described online at http://developer.conjur.net/reference/services/authentication.

This authentication mechanism should not be considered as a replacement for general-purpose authentication protocols. Instead, it is possible and recommended to programmatically map external authentication systems to Conjur's internal authentication mechanism, the specifics of which will vary based on existing authn infrastructure.

Audit

Operations such as permissions or secrets management and permission checks are automatically logged in the audit trail, which can be either reviewed in an easy-to-read format or exported as a JSON stream to be used for access intelligence purposes.

Administrators can also add customized records to the Conjur audit trail, for example, events generated in other parts of their infrastructure.

Authn deals primarily with user identity: who is this person? Is she who she says she is?

Authz, on the other hand, answers a different set questions: what should this user (or system - authz can manage service-to-service as well as user-to-service permissioning) be allowed to access?

Authorization

Conjur itself does not have a "super-user" concept, nor does it have predefined permission hierarchy. Each rule must be created and applied by an administrator.

Every user or host has specific permissions based on his/her assigned role, and the permissions granted to that role.

The only predefined entities in Conjur are "admin" (initial user whose sole purpose is to create identities for actual admins) and an empty "key-managers" group (allowed to manage the directory of public keys of Conjur users).

Admins set up the permissions model to fit their organization needs: create user and host identities, organize users into groups and hosts into layers, and grant permissions between them. This should be done with client tools described below.

The users directory can be created from scratch or imported from external sources such as LDAP.

While simple permissions models can be built by the explicit creation of groups, layers, and resources, then granting permissions across them, more complex models can be provisioned in a single pass by loading "policies", which are written in Conjur's Ruby-based DSL.

For more details and step-by-step tutorials, please visit: http://developer.conjur.net/tutorials/authorization

High-Availability Design

Conjur architecture can be expanded from a standalone server to a high-availability schema, which consists of:

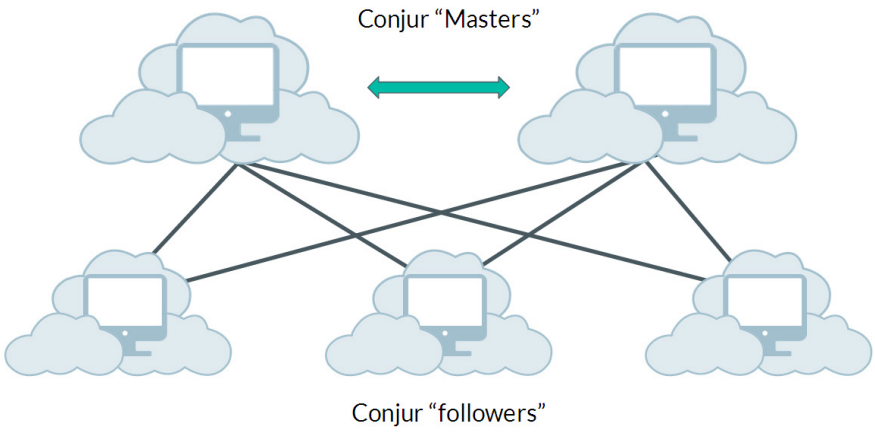
1. **Master server:** handles directory and audit records, manages all permission update operations, and hosts the root SSL certificate used to sign follower's certificates

2. **Standby master:** read-only replica of master, capable of replacing it in case of master server failure

3. **Follower server:** can perform all read-only functions, such as permission checks, distributing public keys, and providing access to secrets. Followers ship the accounting records back to the master via out-of-band communication.

The master-follower architecture provides:

- **Global distribution:** Master and followers can be spread out across availability zones, regions, and clouds
- **Low latency:** By placing followers geographically near servers, a low-latency connection to Conjur will be available whenever needed
- **High availability:** Any follower or the master can call Conjur services. Therefore, if a master or follower is lost, any other master or follower can take over its responsibilities. Failover can be performed manually, or automated with health checks and the auto-scaling capabilities of IaaS



- **Cloud-friendly network architecture:** Conjur only needs to be reachable on two ports: 443 (https) and 636 (ldaps). By distributing followers wherever they are needed, there is no need to set up complex network security and routing configurations in order to allow the servers to reach Conjur; followers should be placed where needed (e.g. in the private subnets), and connect to the master for replication on the fly

Provisioning of HA components is typically done via AWS CloudFormation templates, although it can also be performed with custom scripts on platforms different than AWS.

As a part of the follower provisioning process, a new follower connects to the master using an SSH key provided as a provisioning parameter. Once connected, the new follower downloads initialization data from the master, then finishes configuring the Conjur services. As part of this configuration, an SSL certificate for the follower is created and signed by master, and installed on the follower. It includes environment-specific hostnames used by client VMs to talk to Conjur.

The root SSL certificate is present only on the master and standby master, not on followers. For this reason, followers cannot be promoted to become masters.

More details can be found at http://developer.conjur.net/reference/architecture/ha.html

Platforms supported

Conjur server can be deployed into any platform, including but not limited to: Amazon EC2, Amazon VPC, Microsoft Azure, and bare-metal servers. Components of the Conjur HA architecture can be deployed across different clouds if needed (the only restriction being availability of audit transport and streaming replication between followers and master(s)).

Clients can be deployed in any environment, regardless of Conjur server deployment, as long as ports 443 and 636 are accessible to them.

The officially supported operating system for Conjur servers is Linux, with client tools (described below) available for Linux, Windows and MacOS.

Client tools

Conjur provides comprehensive Ruby-based command-line toolset along with client libraries for Ruby, Java, Python and Node.js. While any of them can be used for interaction with Conjur, the CLI is considered a primary tool for system management and operation.

For those who are more familiar with visual UI, CLI supports an extension, which locally runs basic web UI, providing simple capabilities for permissions management and audit review.

Custom UI solutions can be supported via the client API libraries made available by Conjur.