Access and Security trade-offs for DevSecOps teams

Teleport

Table of Contents

Introduction

Engineering teams building cloud software are always under pressure to deliver new features, fix bugs and improve performance. To move quickly, engineers need access to computing resources: Kubernetes clusters, individual servers, databases, monitoring dashboards, and so on.

Meanwhile, to a security professional, all of these resources represent an ever growing attack surface area. Just think how many attack vectors exist in a production database: an attacker can get SSH access to a database machine via a compromised key, or via a Kubernetes API, or via a compromised web UI, or even via the database's own remote protocol.

This leads to a conflict of interest between engineering and DevSecOps teams. In this article we'll look into the available technologies that can if not alleviate, but at least reduce the tension.

What is Access?

Granting access to modern computing environments is a multi-step process:

- *Connectivity.* First, network connectivity must be established. A client must be capable of establishing a network connection to a resource it is trying to access.
- *Authentication.* The second step is to implement an authentication scheme, i.e. only authorized clients must be able to connect to a protected resource.
- *Authorization.* Finally, a client with access clearly shouldn't be allowed to perform any operation on a resource. For example, some tables in a database must not be accessible to interns, or `'sudo'` command should be reserved only to senior engineers.
- *Audit.* It is important to always keep the audit log as well as a live view of what's happening and who is responsible.

Security Overhead

Let's enumerate what types of computing resources that typically need to be accessed. Regardless of a cloud provider used, the usual suspects may include, but not limited to:

- A cloud API like AWS API for provisioning virtualized infrastructure
- SSH access to virtual machines
- Access to databases used by applications
- Internal web-based tooling such as monitoring dashboards, ticketing systems, various internal web apps that are a part of the stack.
- Kubernetes API (growing in popularity)

Each resource type is a layer in a so-called "tech stack". Each of these layers listens on a socket, speaks its own protocol, has its own implementation of concepts like role-based access control (RBAC), has its own list of users, and often comes with its own configuration file.

Moreover, environments are not singletons. Usually engineering teams need at least two environments: one for staging and another for production, but the practice of creating many disposable environments for experimentation and rapid prototyping is growing more popular.

Needless to say, configuring every single socket of every single instance of every single environment for the best possible security is a laborious task. Moreover, it requires significant expertise because every resource type comes with its own unique security considerations. PostgreSQL security best practices are not the same as SSH best practices, for example.

In addition to the large and constantly changing amount of listening sockets needing protection, there is also a growing number of engineers who need access.

Enabling connectivity, authentication, authorization and audit for every socket in every resource type in every environment for all engineers is hard. That is why very few organizations are capable of implementing access to every resource type properly. A compromise is often made.

Common trade-offs

There are several popular strategies to tame the complexity of remote access. All of them come with their own trade-offs, and organizations often employ a combination of the following strategies:

- *Shared secrets*. In this scenario, a security team carefully configures every resource type for remote access. But to cut on overhead, they configure access only for a handful of predefined users, like "admin" and "app". The credentials for these predefined users are often stored in some form of an encrypted vault.
- *Access restrictions*. It is possible to reduce attack surface area by making certain resource types not accessible by engineers. For example, SSH access can be completely disabled for everyone and only Kubernetes access is allowed.
- *Relying on the perimeter*. Some teams choose to disable security within a private network entirely. They choose to take advantage of the fact that the network itself can authenticate clients via solutions like VPN.

These approaches present numerous problems. Let's highlight just a few:

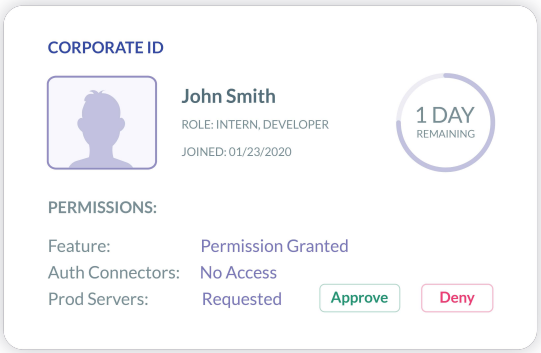
- *Shared secrets* can be stolen because laptops can be stolen. Also, shared secrets can remain on an engineer's laptop even after she changes employers. Moreover, shared secrets do not create a proper audit log, because Bob the Intern and Alice the CTO will both be blended behind the same shared "admin" account.
- *Relying on perimeter security* creates a single point of failure: when attackers get access to a private network, nothing stops them from getting access to everything. Moreover, perimeter security does not allow for a proper audit trail for each resource (server, database, etc).
- *Access restrictions* severely limit engineering productivity and creativity. They also create incentives for engineering teams to be building backdoors for themselves to avoid dealing with corporate bureaucracy.

Emerging Solutions

The cloud computing industry is responding to the access crisis. The buzzwords to pay attention to are: *identity, zero trust, and access plane.*

Identity

Identity-based access means moving away from shared accounts. Each user must login as herself. However, it would be impractical to configure every server, every database and every Kubernetes cluster with all employees of an organization. Instead, identity-based access works by having an employee authenticate against a single user database, receive some sort of a token, and then use this token to access all resources.



For this to work, all resources must understand how to use the token to authenticate users. This problem has been solved for web applications with open protocols such as SAML, OpenID Connect, OAuth2, etc.

However, these standards are not compatible with resources that do not speak HTTP, such as SSH servers or databases.

Zero Trust

Zero trust based access means moving away from perimeter security. A computing environment built on zero trust principles means that every resource (server, database, kubernetes API, web dashboards, etc) acts as if it was running on a publicly accessible IP address. It means that every resource listens on a socket that:

- Uses encryption
- Performs authentication and authorization of users
- Maintains its own audit log

It's as if instead of having a single door lock for a house, we'd agree to have an open door, but the fridge, the microwave, the TV, or everything else inside the house had its own lock.

The obvious problem with Zero Trust is that users have to authenticate over and over again, to touch anything, but if we combine Zero Trust approach with identity-based access, it means that users can merely use the same token over and over with all resources, which can be automated.

Access Plane

An acces plane builds on top of Identity and Zero Trust and it allows organizations to let go of the access restrictions. An access plane consolidates access and:

- Creates a single access point for all engineers, all resources types, across all environments.
- Enforces identity-based access for all resources and people. It uses certificate-based authentication and authorization, therefore acts as a certificate authority of an organization.
- Automatically creates certificate-based connections to all resource types, even if they do not natively support it. This removes the overhead of having to configure every resource type separately.
- Maintains the centralized audit log, creating real time and historical view of all events.
- Implements authorization for every supported protocol.

An access plane is a combination of an identity-aware proxy, a certificate authority, and a protocol-aware “sidecar” for every resource type, tied to a centralized audit log. It is the glue that connects together identities of people and computing environments, along with all resources running in them.

The Example

Let's consider an engineering team which uses an access plane configured to integrate with Github as their identity source.
A new intern, Bob, joins the team. IT adds Bob to Github and makes him a member of the "interns" group ("team" in Github). Bob's mentor is Alice, a software engineer. She is also in Github, as a member of the "dev" team.

Bob fires up the terminal and tries accessing SSH servers on the playground environment, he is asked to login into Github via a browser. He authenticates into Github using his email address, password and the 2FA app on his phone. Upon successful authentication, Bob's terminal is configured with certificates for all remote access protocols.

From now on, when Bob types "`ssh`", "`kubectl`", "`psql`" or "`mysql`", the access is automatically established across all servers, Kubernetes clusters, and databases that interns have access to. Meanwhile, Alice has access to a completely different set of servers and databases, because she's a member of a more privileged "devs" team.

The DevSecOps team can see all actions performed by Bob or Alice in real time. When Bob or Alice access servers or databases, all of their actions are stored in a centralized audit log in ElasticSearch.

When Bob leaves the office, his access is automatically revoked. If Bob's laptop gets stolen or broken into, an attacker will not be able to find any useful credentials on its SSD.

You can check out **this demo video** to see how the open sourced **Teleport** access plane works in practice.

Advanced Techniques

There are additional improvements to security that can be made. One recommendation is to implement the principle of least privilege.

In practice it means eliminating "root" type accounts. *Nobody should have absolute access to everything.*

Engineers are already used to the idea that nobody should be able to launch new code into production by himself, a peer code review is required. A similar idea lies behind an advanced access technique called "access requests". Here's how it works:

What if an engineer could create a "git pull request" requesting temporary access to critical production infrastructure? Then her peers would review and approve such requests, granting her temporary access,

with solid security and compliance guarantees.

In practice, this works by an engineer (let's use Alice from above example) requesting to be placed into a privileged group for a specific amount of time, also providing a reason for access. The access plane will forward the request to her peers (or to a dedicated security team) via Slack (or other workflow tool), where her request can be approved or denied.

Some security-minded organizations implement a more advanced version of access requests called a "four eye policy", when access is granted only when the live session is streamed and viewed by another person, making sure that not less than four human eyes are watching what Alice is doing.

Conclusion

Implementing connectivity, authentication, authorization and audit logging for every socket in every cloud environment used to be an insurmountable task. The best tech companies in Silicon Valley employ the best talent to build and maintain in-house solutions dedicated to this task.

But thanks to recent advancements in access technologies, everyone can now apply identity-based authentication and authorization and zero-trust principles for their computing resources using the concept of an access plane.

More advanced organizations can implement concepts such as principle of least privilege and temporary privilege elevations with techniques such as access requests or "four eye" policies.

The end result is a simple remote access which makes engineers more productive, doesn't compromise security, enforces compliance and allows visibility into everyone's behavior. A win-win.