Unlocking Zero Trust Delegation through Permissions and Policies

nicola-gallo
Published in ztauth
11 min read · Nov 29, 2024

Chapter 2

In the previous chapter, I discussed resources, actions, and accounts. In this chapter, I will introduce the concept of **Zero Trust Delegation**, which will be referred to as either **Trusted Delegation** or **ZTDelegation** throughout the rest of the chapters.

Trusted delegation involves both authentication (**AuthN**) and authorization (**AuthZ**), addressing each in distinct ways. As discussed in the previous chapter, this requires adhering to **Zero Trust** principles, tailored to the specific context of software systems.

Here they are, in case you don't have them on hand.

- **Never trust, always verify:** Never trust implicitly; always verify the identity and context of users, devices, and applications before granting access.

- **Least privilege access:** Grant the minimum level of access necessary for a task, ensuring users or systems only interact with the resources they truly need.
- **Assume breach:** Operate under the assumption that a breach could occur at any time, designing systems to contain potential damage and prevent lateral movement.

To make the discussion easier to follow, let's introduce an example to guide us through the explanation of the concepts.

Let's use an **accounting system** as an example, where we define two business roles:

- **John:** An ***accountant*** who manages the full lifecycle of invoices (***view***, ***create***, ***update***, ***delete***, ***approve***, ***reject***).
- **Bob**: An **apprentice** who can **view** invoices but cannot perform any other actions..

John has started training Bob, and Bob has gained some skills. When John is busy or on holiday, he wants to allow Bob to create new invoices. However, these invoices would not be approved unless someone else reviews and approves them. Essentially, John wants to **delegate** a **subset** of his **responsibilities** to Bob, limited to specific operations, for a defined period, and with proper tracking.

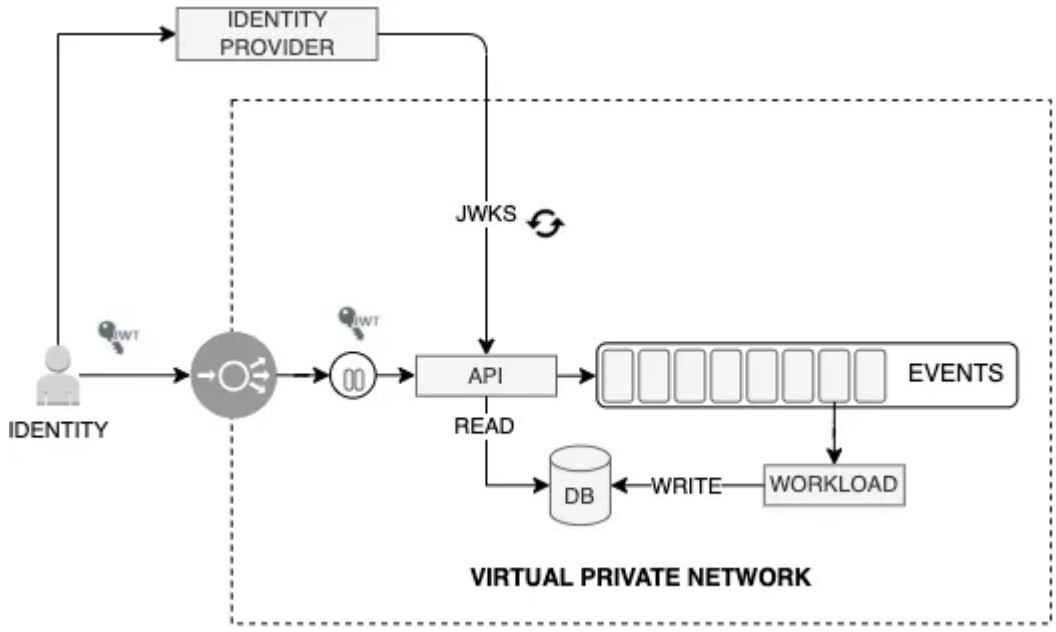
> *Certainly, you might think this problem is simple to solve — John could create a dedicated role, assign it to Bob, and add or remove it as needed. However, this approach focuses on role assignment, which temporarily grants Bob a specific role. While similar, this is fundamentally different from delegation. Delegation involves Bob acting on behalf of John, meaning his actions are performed as if executed by John, with clear attribution and accountability.*
>
> *The challenge is to implement delegation in a way that maintains the identity separation and avoids altering the structure of identities, roles, permissions, policies, or schemas.*

> *Instead, the focus is on enabling a mechanism where Bob can act under John's authority without changing their predefined roles or permissions.*
>
> *It is important to keep in mind that this solution must address Autonomous and Disconnected Challenges, where the Proximity PDP synchronizes immutable Auth* models. Immutable means they cannot be changed, and assigning or removing roles would change them. The challenge lies in preserving the integrity of these models while ensuring reliable functionality, even in disconnected environments.*

This example will help illustrate how different roles in the accounting system correspond to specific permissions and how those permissions are implemented.

Let's imagine a scenario where the system consists of APIs that receive a **JSON Web Token (JWT)** issued by an **Identity Provider (IDP)**. The APIs process the token and perform operations that may vary depending on the type of request. For example:

- **Non-mutating operations:** The API queries the database and retrieves the necessary data without making any modifications.
- **Mutating operations (e.g., creating or approving invoices):** The API publishes an asynchronous message, which is later processed by a workload to perform the required changes.



Accounting System Architecture

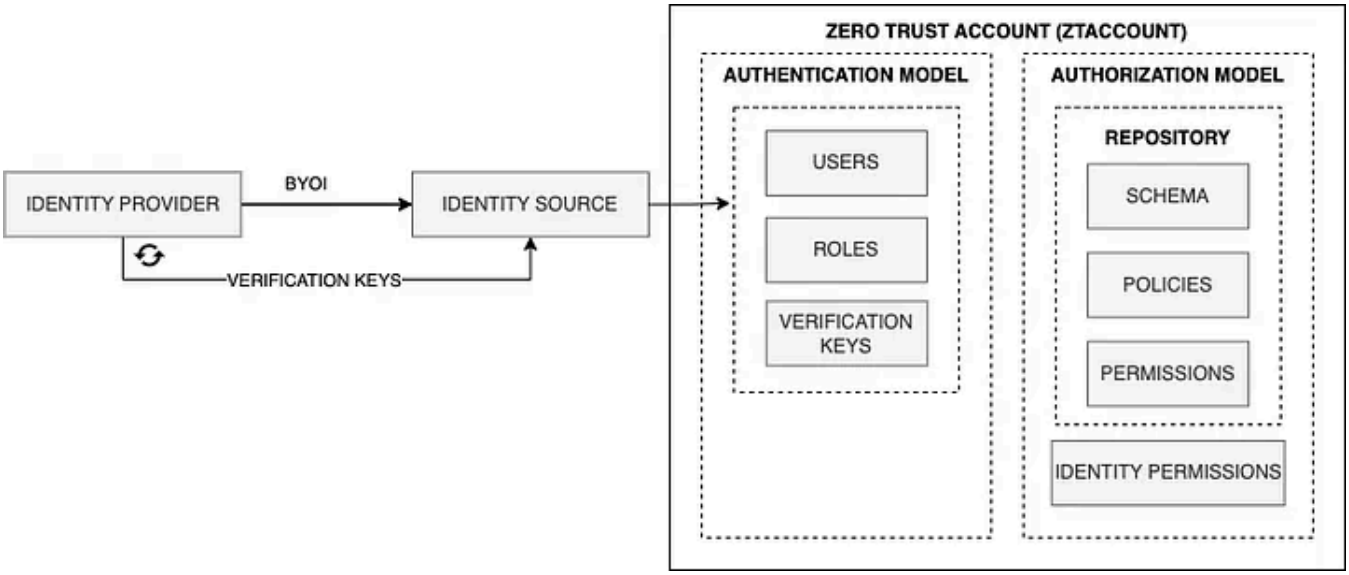
To build this system, beyond adopting an IDP and creating APIs, workloads, etc., an essential step is setting up the necessary components, which include the following:

- **Identity Source Connector:** The accounting system must synchronize identity metadata from the IDP, ensuring that only the essential identity information is imported into the system. Identity management remains the responsibility of the IDP, while the accounting system simply fetches metadata to map it to its internal Auth* models.
- **Public Keys for Token Validation:** The system must synchronize the public keys from the *IDP* using the **JSON Web Key Set** (**JWKS**). These keys are used to validate the *JWT* locally, ensuring its authenticity.
- **Infrastructure Access to Apache Kafka:** Access to the Kafka broker for asynchronous processing is facilitated using infrastructure-level credentials, typically associated with service accounts, rather than individual user identities.

With this setup, the API can successfully verify external JWTs and handle incoming requests as expected.

Naturally, these aspects must be handled in some way by the accounting system, which therefore must be capable of synchronizing JWKS or, more generally, verification metadata.

> Fetched verification keys should be an integral part of the Authentication Model.



ZERO TRUST ACCOUNT

However, as you may already be thinking, at this point in the process, we are still violating several critical **Zero Trust** principles:

- **Never Trust, Always Verify:** While the JWT is validated for the API, the workload processing the Kafka message implicitly trusts the incoming message.
- **Least Privilege Access:** The Kafka broker allows infrastructure-level credentials that may have broader permissions than necessary.
- **Assume Breach:** If the workload is compromised, the trust placed in the Kafka connection and the infrastructure credentials becomes a significant risk.

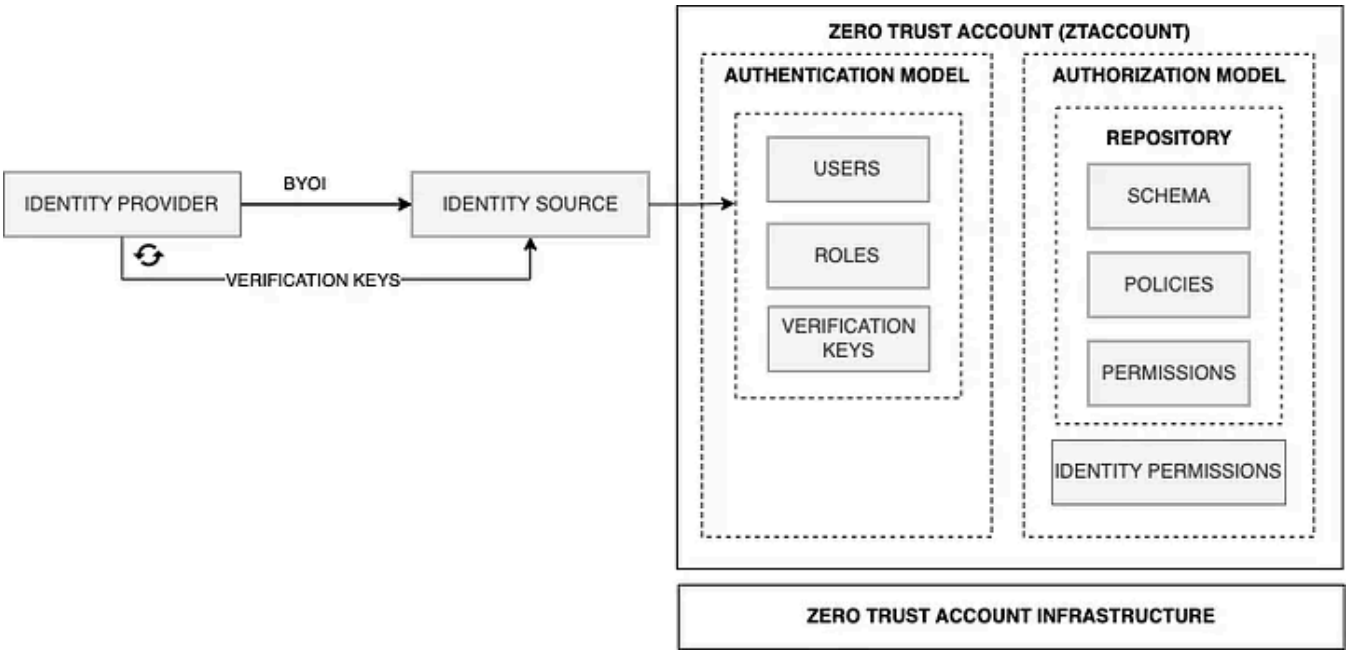
This means that, despite verifying the JWT at the API level, the workload processing the message violates Zero Trust principles, which is unacceptable in a secure environment.

To address this, the system must ***extend Zero Trust principles to every layer***, including the ***workloads*** processing ***asynchronous messages***. Each component must validate the identity and permissions of the identity at every interaction, avoiding implicit trust at any stage.

Wait a second — we just mentioned the **magic word**: every layer. The same logic applies here. The message should be signed securely, just like the **JWT token**, and it must be pushed to **Apache Kafka** as a secure message.

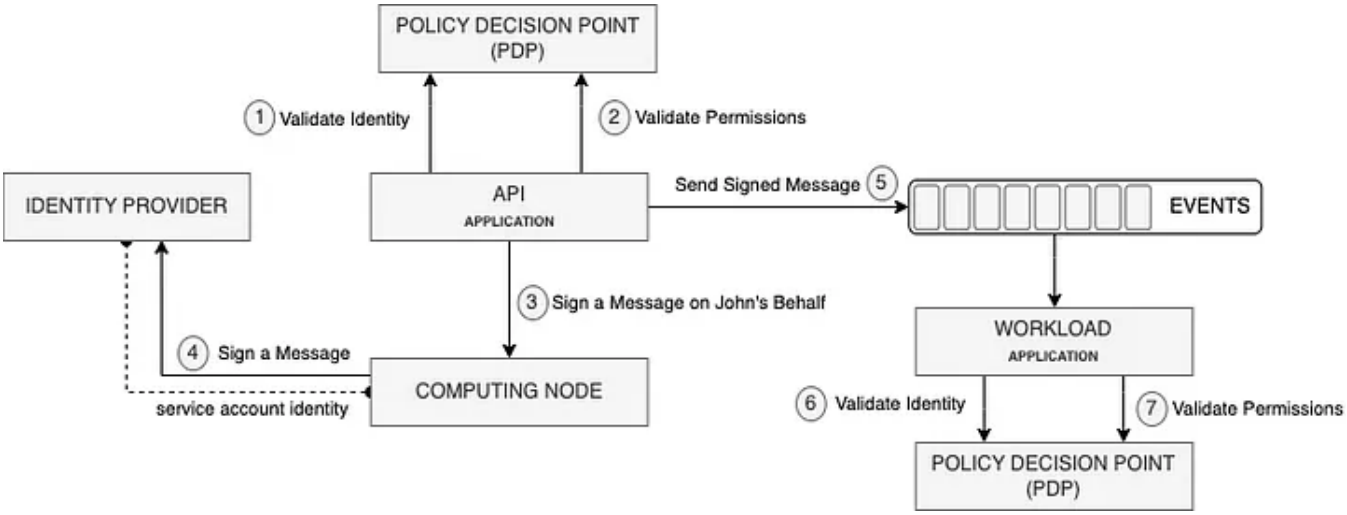
This introduces a challenge at the **infrastructure level**: the service account running the system must have an **identity** linked to an **IDP** to **sign** the **message** when it is written. Infrastructure that ensures this federation can occur and has the capability to sign the message is referred to as **Zero Trust Account Infrastructure**. At this stage, we are not focused on specifying how this would be architected.

> *It is important to note that IDPs do not have such capabilities in their standard implementation. Therefore, this is an area that requires further exploration and understanding.*



ZERO TRUST ACCOUNT

When the **message** is **read**, the **workload** must validate it using the **validation keys**, likewise it does for the JWT use case. This separates the process of signing from validation.



Api, Kafka flow

Let's look in more detail at how this process happens:

1. **Identity Validation:** The API validates the identity with the PDP (Policy Decision Point), which internally uses the validation keys defined in the Authentication Model.

2. **Permission Validation:** The API validates whether the identity has the required permissions to execute the requested action. This is done using the Authorization Model.

3. **Message Signing Request:** The API requests a computing node, part of the Zero Trust Infrastructure, to sign a message on John's behalf.
4. **IDP Signature:** The computing node interacts with the IDP (Identity Provider) to securely sign the message.
5. **Message Delivery:** The API sends the signed message to Apache Kafka. The signed message includes a signature certifying that the message originates from the Service Account acting on John's behalf.
6. **Workload Identity Validation**: The workload validates the identity with the PDP, which again uses the validation keys from the Authentication Model.
7. **Workload Permission Validation:** The workload verifies that the identity has permission to execute the requested action. This is done using the Authorization Model.

Why is the **ZTAccount** limited to holding validation keys and does not implement operations for creating signed messages?

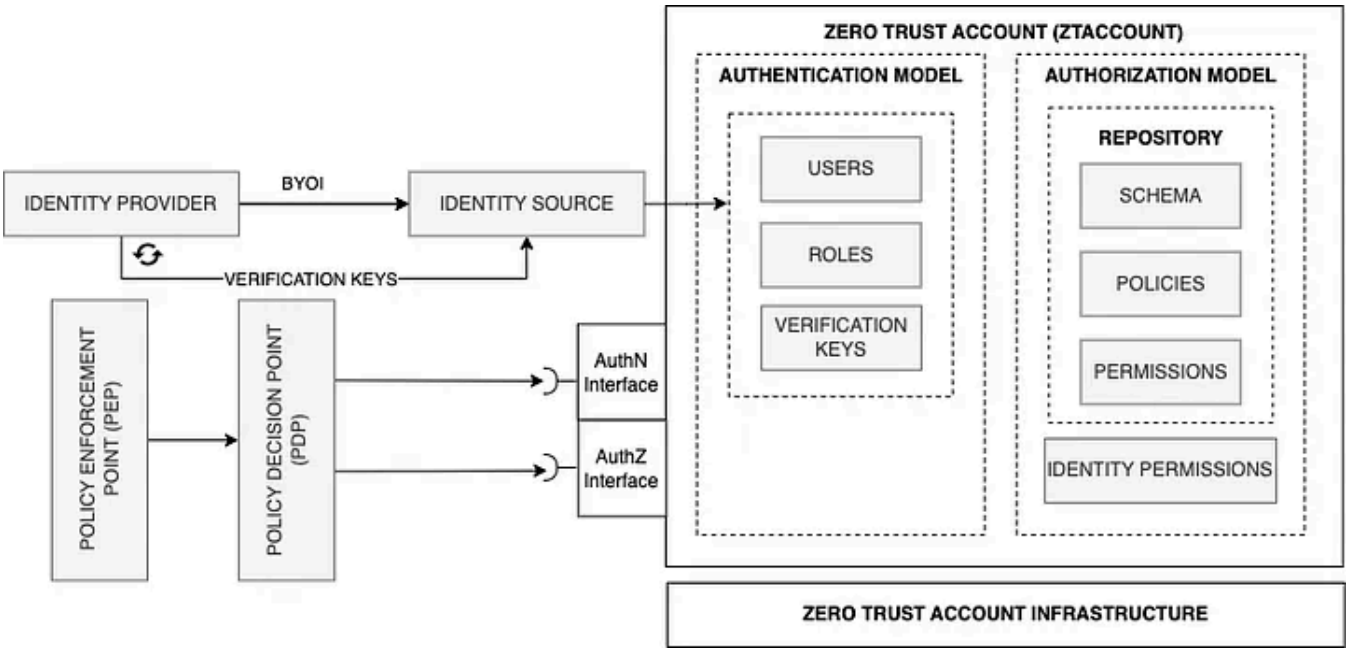
The answer lies in a fundamental principle of good design: **separation of concerns**. Let me ask you this: Is the **ZTAccount responsible** for creating **JWT tokens**? If not, why should it also be responsible for both creating and validating secure, certified messages?

To maintain a robust, scalable, and secure system, **roles** and **responsibilities** must remain **distinct**. Signing a message serves two purposes: certifying the message and certifying/verifying the identity of its creator. This is the responsibility of the **IDP**, not the **ZTAccount**.

Before completing this section, we need to clarify how the **PDP** should communicate with the **ZTAccount**. The **ZTAccount** will certainly need to include:

- An **AuthN interface** to interact with the Authentication Model
- An **AuthZ interface** to interact with the Authorization Model

By **interface**, I do not refer to a specific implementation; it can involve any type of implementation, such as reading from files, APIs, or other methods.



ZERO TRUST ACCOUNT

Now it's time to go back to Authorization. Let's design the ***Authorization Model Schema*** first.

```yaml
domains:
  - name: accounting
    description: Manage accounting operations
    resources:
      - name: invoice
        description: Represents an invoice in the accounting system
        actions:
          - name: view
            description: View details of an existing invoice
          - name: create
            description: Create a new invoice for a invoice
          - name: update
            description: Update the details of an existing invoice
          - name: delete
            description: Cancel or remove an existing invoice
          - name: approve
            description: Approve an order for further invoice
          - name: reject
            description: Reject an invoice and prevent further processing
```

After that, we'll define the permissions and policies for the **Accountant** role.

```
---
name: accountant-permission-for-creator
policies:
  - accountant-invoice-creator-policy
---
name: accountant-permission-for-approver
policies:
  - accountant-invoice-approver-policy
---
name: accountant-invoice-creator-policy
effect: permit
actions:
  - invoice:view
  - invoice:create
  - invoice:update
  - invoice:delete
resources:
  - accounting/invoice/*
---
name: accountant-invoice-approver-policy
effect: permit
actions:
  - invoice:approve
  - invoice:reject
resources:
  - accounting/invoice/*
```

Finally, the Apprentice role is defined.

```
---
name: apprentice-permission
policies:
  - apprentice-invoice-viewer-policy
---
name: apprentice-invoice-viewer-policy
effect: permit
actions:
  - invoice:view
resources:
  - accounting/invoice/*
```

This sample **domain-specific language** means that the resource is represented by the domain, the action is defined within the resource, and a policy can either permit

or forbid the action. Permissions are used to group these policies, which are then attached to identities.

The final step is to associate the roles with the appropriate ***permissions***:

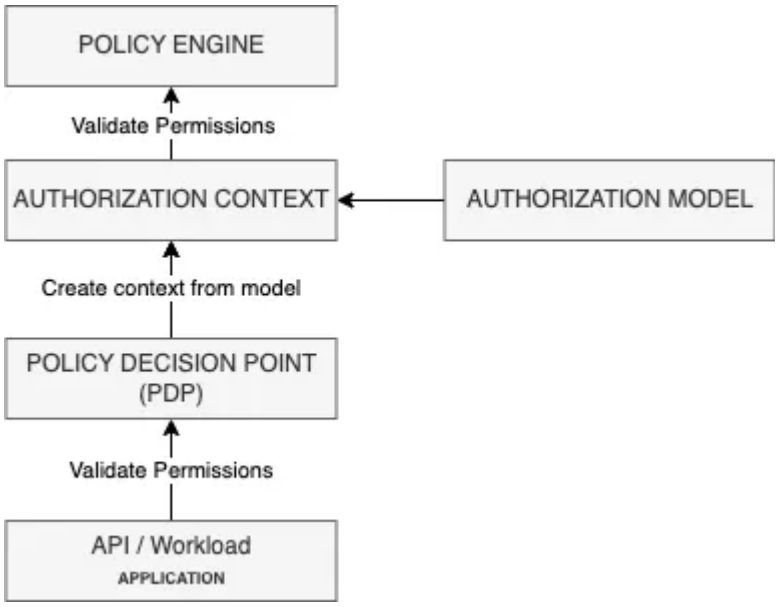
- **Accountant**: Associated with **accountant-permission-for-creator**, **accountant-permission-for-viewer**
- **Apprentice**: Associated with **apprentice-permission**

> *Please note that these are just representative examples. In the next chapters, we'll dive deeper into each element to fully understand their meanings, how they work together, and, of course, begin creating a more suitable domain-specific language to define permissions and policies.*

Suppose John calls an API to approve an invoice. Here's what happens:

- **API Authorization Check (Before Sending to Apache Kafka):** The API validates John's permissions using the Authorization Model to ensure he is authorized to approve the invoice. This check happens before the message is sent to Apache Kafka.
- **Workload Authorization Check (After Reading from Apache Kafka):** The workload validates permissions again after reading the message from Apache Kafka. This ensures the action is authorized before processing the message.

The **PDP** will read the **Authorization Model**, extract the **permissions** associated with the **identity** — in this case, John — and build an **Authorization Context**. Essentially, this context includes all the active and associated permissions for the identity, providing a comprehensive view of what actions the identity is authorized to perform.



Authorization Model and Authorization Context

This Authorization Context will be passed to the **Policy Engine.** Naturally, other contextual information will also be included, which we will explore in more detail in future chapters.

Now let's think about delegation. Suppose we want Bob to perform an operation, such as creating an invoice, on behalf of John. This requires John to create a delegation that explicitly grants Bob the authority to carry out this action on his behalf.

Step 1: John creates a delegation that explicitly grants Bob the authority to create invoices on his behalf.

```
name: sa-accountant-approve-delegation-bob
delegation:
  delegated-identity: serviceaccount1
  delegator-identity: bob
  validity-period: 1h from 29/11/2024 05:00:00
  target-permission: accountant-permission-for-approver
```

Step 2: Bob is authorized to perform the operation as a proxy for John.

Step 3: The delegation ensures that the system recognizes Bob's actions as being performed on behalf of John.

Step 3: This setup allows Bob to create the invoice while maintaining transparency and accountability.

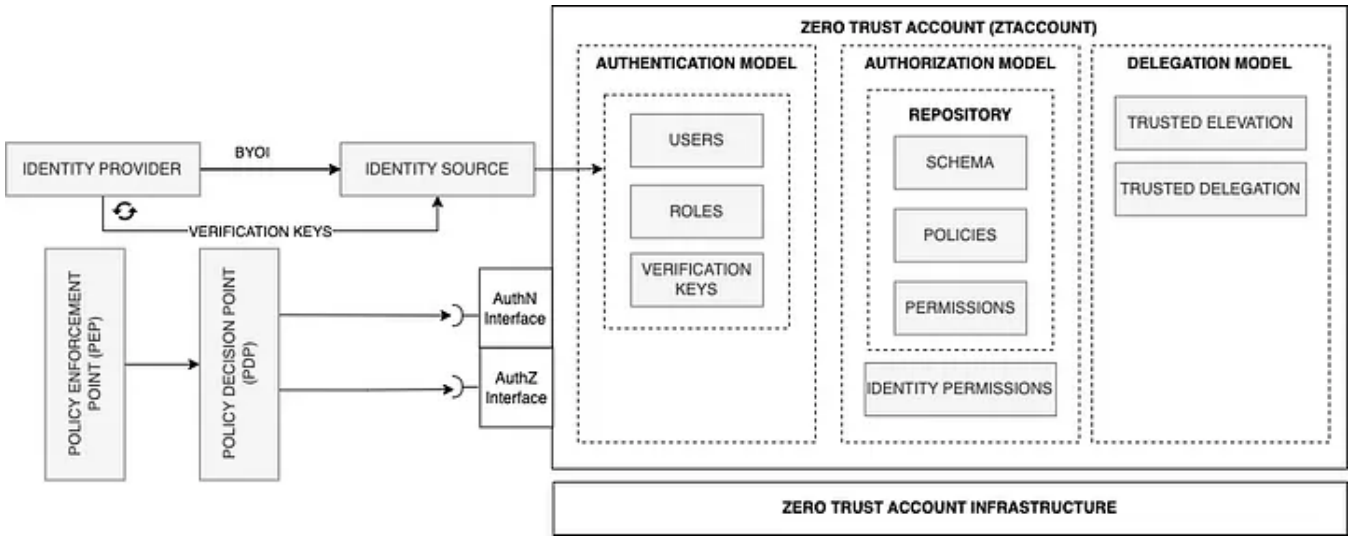
To better understand this, let's use the API scenario. The system validates that Bob has been granted delegation by John to create invoices, ensures the delegation is valid, processes the request, and sends a message to Apache Kafka containing the delegation details.

1. The **API** receives a request from **Bob** to create an invoice on **behalf** of **John.**
2. The **API** verifies that **Bob** has a valid **delegation** from **John** and elevates his **Authorization Context** to simulate John's, but restricted to the conditions specified in the delegation. It then validates the necessary permissions.
3. The **API**, through a **computing node**, creates a **message signed** by **Bob** acting on **behalf** of **John**, explicitly specifying that the operation originated from a **Service Account** acting on **Bob's behalf.**

> *It is important to note that if Bob had not specified he was acting on behalf of John, the request would have failed. This is because the Authorization Context would not have been elevated to include the delegation from John. Consequently, the request would fail as Bob's Authorization Context does not have the necessary rights to create an invoice.*

By now, we have completed the ***Trust Delegation*** flow. However, thinking from a ***Zero Trust*** perspective, what could we add to be more compliant with the principle of Assume Breach?

We mentioned that the ***service accoun***t loads an ***Authorization Context*** and, in some cases, can ***elevate*** this context to another one through ***delegation.*** But what happens if, for any reason, we want to **restrict** certain **computing nodes**, and their associated service accounts, from working with specific types of Authorization Contexts? The same consideration could also apply to a specific **identity**, where we might need to **restrict** its ability to **elevate** or interact with certain ***Authorization Contexts.***



ZERO TRUST ACCOUNT

One solution could be to define the concept of **Zero Trust Elevation**, referred to as either **Trusted Elevation** or **ZTElevation** throughout the rest of the chapters. This concept would specify which identities are permitted to load or elevate specific **Authorization Contexts**. For example:

- A **Trusted Elevation** for the service account could allow it to elevate to any identity's Authorization Context.
- A specific **Trusted Elevation** for Bob could restrict elevation to John's Authorization Context only.

Here's an example of how it could look like:

```
name: sa-accountant-elevation
elevation:
  current-identity: serviceaccount1
  target-identity: *
  validity-period: 1h from 29/11/2024 05:00:00
```

This approach would enable granular control over service accounts and individual identities, preventing certain elevations when risks are detected.

> *Risk Mitigation Example: Imagine a scenario where a risk analysis system detects a potential threat or high-risk situation. To mitigate the risk, it could revoke the Trusted Elevation for the service account. This would effectively prevent context elevation without altering the existing permission policies. The permissions themselves remain intact, but*

> *the elevation of contexts is blocked, adding an additional layer of protection. This model ensures flexibility and compliance with Zero Trust principles, enhancing security while maintaining operational continuity.*

Next Chapter: Introducing the ZTAuth* Architecture

Zero Trust | Policy As Code | Security | Authorization

ZTAuth*

Following

Published in ztauth

4 Followers · Last published 6 days ago

ZTAuth*: Redefining AuthN, AuthZ, and Trusted Delegation with Transferable, Immutable, and Resilient Models for a Zero Trust World

Edit profile

Written by nicola-gallo

9 Followers · 4 Following

A tech entrepreneur specializing in cloud and enterprise strategic technology, with a focus on distributed systems.

No responses yet

What are your thoughts?

Respond

More from nicola-gallo and ztauth