CAEP Best Practices

Atul Tulshibagwale, CTO, SGNL

01

Overview

CAEP and the underlying SSF standards are critical to implementing the continuous security paradigm and, ultimately, Continuous Identity at scale. This white paper describes the standards, what they do, and why they are important; the concepts embodied in the standards; the continuous security paradigm; and tips to implementing CAEP.

02

Basics

Before we get into the details, let's take a look at what these standards are, and why they are important.

What are CAEP, SSF, and related standards?

The Continuous Access Evaluation Protocol or Profile (CAEP) is a way for independent services that share the same logged-in users to inform each other of changes to the properties of their logged-in sessions. Individual services can define policies about how such information should be used. For example, if a user changes their password at an identity provider (IdP), then it can send the CAEP event "credential changed" to other services, which can then determine whether the user needs to be re-authenticated or can operate partially without re-authentication.

CAEP is built on the Shared Signals Framework (SSF), which is a generic mechanism to asynchronously exchange information about common subjects. CAEP and SSF are open standards developed by the OpenID Foundation's Shared Signals Working Group (SSWG). The CAEP specification defines a set of session management-related events on top of SSF. Similarly, other standards such as RISC (Risk Incident Sharing and Coordination), which defines a set of events for account security, and SCIM Events, which defines a set of events for account management, can leverage SSF. RISC is developed in the OpenID SSWG, and SCIM Events is being developed in the IETF OAuth working group.

Why is this important?

Users are most commonly logged in via single sign-on, which leverages federated identity standards like OpenID Connect and SAML. Once the user logs in to a third-party application (such as a SaaS service), the ability to inform of any changes to that logged-in user's properties is severely limited. Organizations have policies around what device posture is required for a user to access certain services via single sign-on, what to do when a user changes their password (or other credentials), and other such conditions. However, in federated identity protocols, the only time available to enforce such policy is when the user logs in.

So, to achieve such policy compliance, sometimes companies use short-lived tokens, so that the user is bounced back to the identity provider every hour or so. Even if the user doesn't have to log in again, the identity provider can check the policy and deny access if the policy check fails. This type of "polling behavior" used to be thought of as the only way to achieve "continuous authentication" before CAEP. It was unnecessarily chatty and caused a poor user experience because the browser would have to reload the application every time the user was bounced back and forth. CAEP is an open standard that enables such dynamic updates to be communicated where they are required, so that short-lived tokens are no longer required. This is described in the SGNL blog: **CAEP Use Case: Increase Token Lifetime**.

03

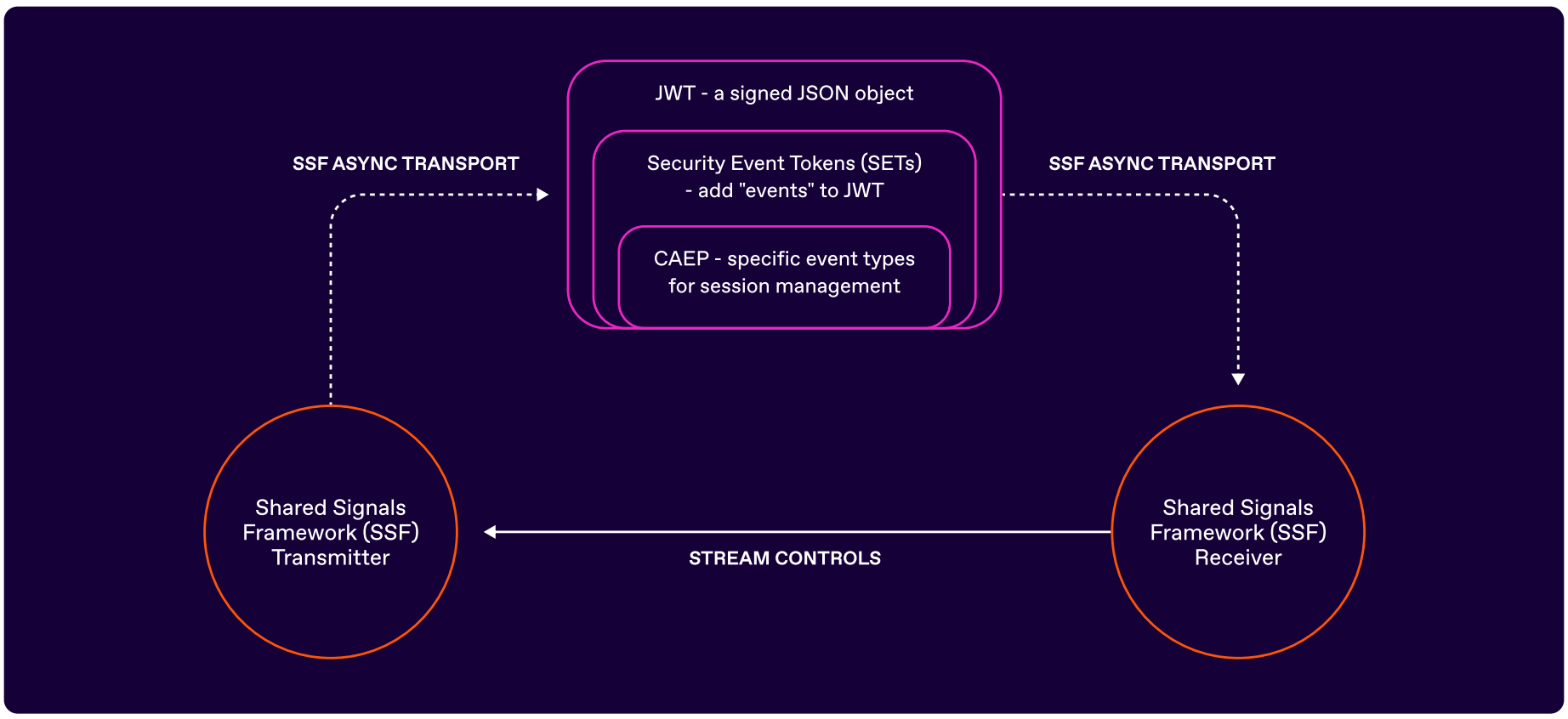
SSF Concepts

- **Transmitters and Receivers:** The shared signals framework considers individual systems to be either Transmitters or Receivers. A system may be both, but these aspects are independent of each other.
- **Events:** Events are the unit of communication. Transmitters send events to Receivers. Each Event is a Security Event Token (SET), which is itself a form of a signed JSON object, called JWT (JSON Web Token).
- **Event Types:** Each event has a type, which is one of a defined set of types in one of the specifications, such as CAEP, RISC, or SCIM Events. It is possible for other specifications to define new event types, or for parties to use custom event types between themselves. Custom event types should be avoided because they can lead to incompatibility and ambiguity in what they mean.
- **Subjects:** A subject is what an event is about. The subject can be expressed as a simple subject (e.g.,email) or a complex subject (e.g.,a specific session identifier, on a specific device for a specific user).
- **Streams:** A stream contains events of specific event types, about specific subjects, and delivered in a specific way (push or poll) between a Transmitter and Receiver. Streams have specific mechanisms for how they are created, updated, controlled (paused, disabled, enabled), and verified.
- **Transmitter Configuration Metadata:** The Transmitter defines an API by which Receivers can call various functionality about it. The Transmitter Configuration Metadata informs the Receiver about all properties of the Transmitter in order to be able to communicate with it.

04

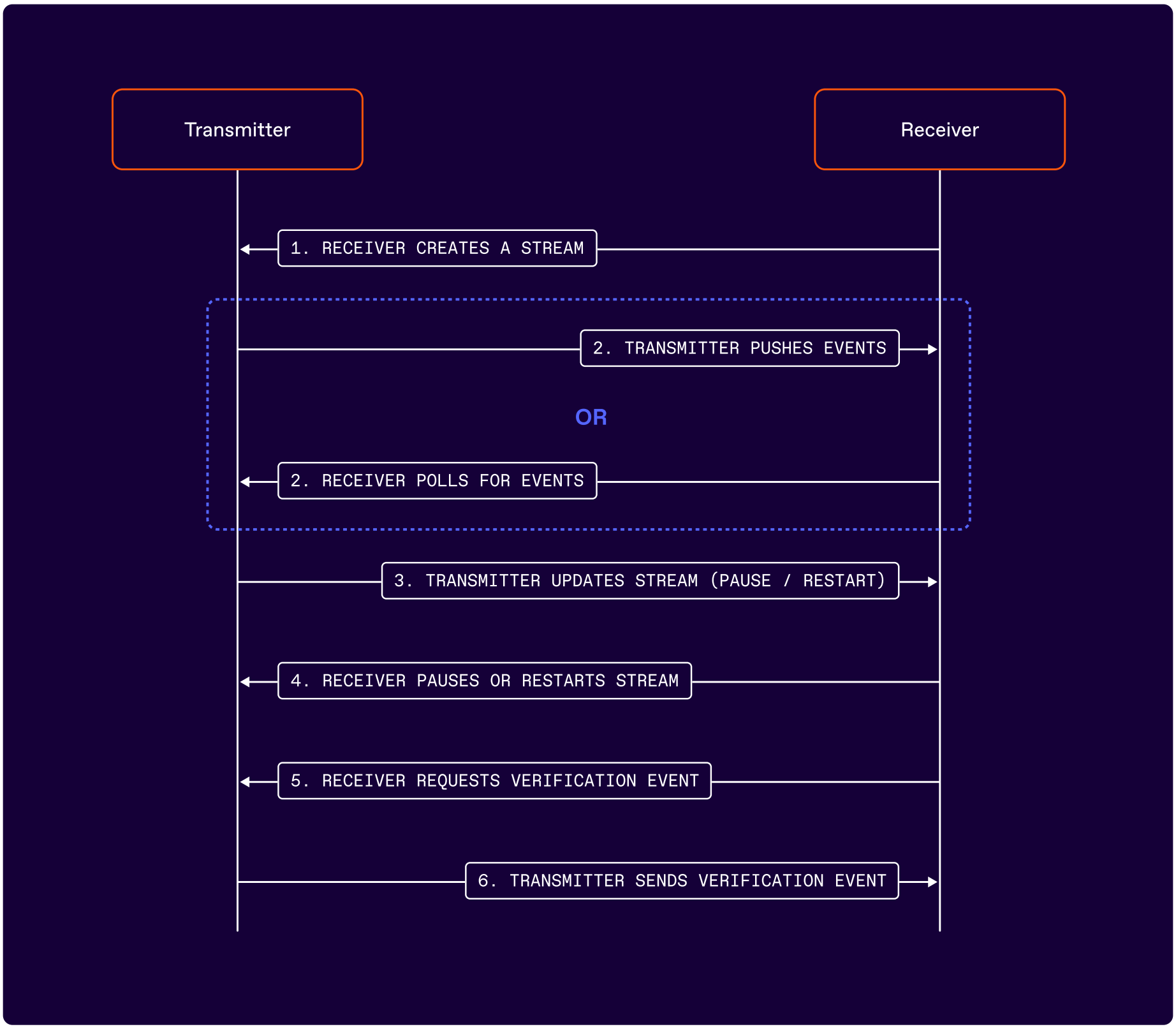
How does it work

The general architecture is as follows:



As seen above, the thing that is sent by a Transmitter to a Receiver is a specific form of JWT called a SET, which is further profiled in the form of a CAEP event. The stream controls enable the Receiver and Transmitter to negotiate the event types that go into the stream, and the subjects about which the events are. It also enables the Transmitter and Receiver to control the status of the stream and request verification.

The sequence of events is described in the diagram below:



In addition to the above, a Receiver may add or remove subjects at any time by calling the relevant endpoint of the Transmitter's API. Streams may also be set up such that all subjects are included by default.

The Continuous Security Paradigm (CSP)

SSF is a foundational framework that enables the higher level security paradigm called the "Continuous Security Paradigm" or CSP. CSP proposes an architecture wherein independent services asynchronously exchange data that is relevant to making access decisions. Such asynchronous exchange of data provides the up-to-date context that is essential to good access decisions.

CSP is an important shift in security thinking; It recognizes the architectural change in how enterprises are organized, and defines how security can work in that context. CSP is also a foundational way of thinking to deliver on **Continuous Identity** in enterprise environments.

Network of nodes

Modern organizations typically use a number of cloud services, including SaaS, PaaS, and IaaS services. Your IaaS platforms host multiple systems and applications. Each one of these services (including your own systems and applications) can be thought of as a node in a network. Each node has relative autonomy in its operation and has management concerns of its own, but as an enterprise, you want it to work and be managed in coordination with other systems that you run.

Zero trust

Your users are located anywhere on the globe, and work from their individual homes, on the road, or in offices, and need to access cloud-based services. These services are themselves globally distributed; it is natural to have users connect directly to such cloud services without going through any specific network hops. A zero-trust architecture—one in which every access is verified independently and directly by the service being accessed—is natural to enforce access security in such environments. This is because any network security architecture places unnecessary hops and, as a consequence, points of failure. Network components also cannot incorporate any context in an access decision, so their decisions are necessarily based on static role memberships and network properties.

Access tokens

In a zero-trust architecture, access is enforced at every access request, based on information that is verified at the time of access. In real life, this means verifying the access token that gives the user the ability to access a specific service. Access tokens are issued by a combination of your organization's identity provider and the specific system a user is trying to access. The IdP issues tokens using federated identity protocols such as OpenID Connect (OIDC) or SAML, whereas the individual systems may use proprietary or JWT format OAuth access tokens for their individual use. The tokens are either stored as cookies in browsers or as access tokens in mobile apps.

The access tokens have a validity lifetime of their own, determined by your organization by configuring the individual system that issues the tokens. Security practitioners have to decide whether the access token should be long-lived (e.g. 24 hours), or short-lived (e.g. minutes to an hour). This is typically determined by how often a user's access posture is likely to change. Keeping a short token lifetime forces the user to have to go back to the IdP to re-issue the federated identity token, which is disruptive to the user's experience and can place an unacceptably high burden on the IdP. Therein lies the dilemma of the security practitioner: If you make token lifetimes too long, security suffers, and if you reduce token lifetimes, user experience and system reliability suffer.

Access decisions

In a zero-trust architecture, access decisions should ideally be made independently for every access. Instead, systems often end up using token validity as a quick means of verifying access, because that seems like a good way to do a low-latency check that doesn't require a lot of complex computation that might go into an access decision.

But let's take a look at how it should really work:

- **Context is distributed:** Each node has your organization's data, and data from any node could be required to make access decisions at any other node. When a user (or an API call) makes a request within any node, it needs a near-instantaneous access decision. However, the data required to deliver that access decision may be drawn from other nodes. Expecting all such nodes to work synchronously in real-time to provide an access decision with ultra-low latency is virtually impossible. Consider the following examples:

  - **Consumer use case:** A customer of a bank has logged in and is requesting to transfer money from one of their accounts. The application responsible for executing the transfer needs data from the fraud alerting system to know whether the user's login session is showing any signs of compromise, such as session hijacking or credentials compromise. An independent fraud detection system can compute this information, but it needs access to other systems such as the identity provider, geolocation, etc., to do such computation. So, when the transfer execution system is invoked in order to execute the transfer, it is unrealistic to expect it to reach out to the fraud detection system, and then have that reach out to the IdP and geolocation services to compute all this in real-time. It also doesn't work because each system may have differing availability and latency characteristics.

  - **Enterprise use case:** An employee wishes to make a change in the configuration of your organization's IaaS platform. This is a highly sensitive change, so your organization has decided that this should be allowed only if:

    1. There is a support case that is approved for configuration changes in production.
    2. The user is assigned as the engineer on that case.
    3. The user is currently on duty as an on-call engineer.
    4. The user is in the "site reliability engineering" team that is permitted to make such changes.
    5. The user is not on leave at this time.
    6. The user is coming from a PC that does not have any incidents reported, and is in good management posture.

    All of these pieces of data are in different systems (e.g.,ticketing system, on-call scheduling system, enterprise directory, HR, XDR, etc.). All of this data influences the access decision.

Achieving continuous security

To achieve continuous security in a zero-trust environment, you need to make these access decisions for every access request at every node, and you need to do that in a reliable, low-latency way! Keep in mind that in large organizations, individual systems may have hundreds of thousands of access requests every second.

So, while it is impossible to rely on all this distributed data to make such highly reliable, low-latency access decisions, that doesn't mean continuous security is impossible to achieve. The continuous security paradigm makes this possible. The idea is actually quite simple:

- Always make instantaneous access decisions based on data available at the individual node. In its simplest form, this can be validating an access token.

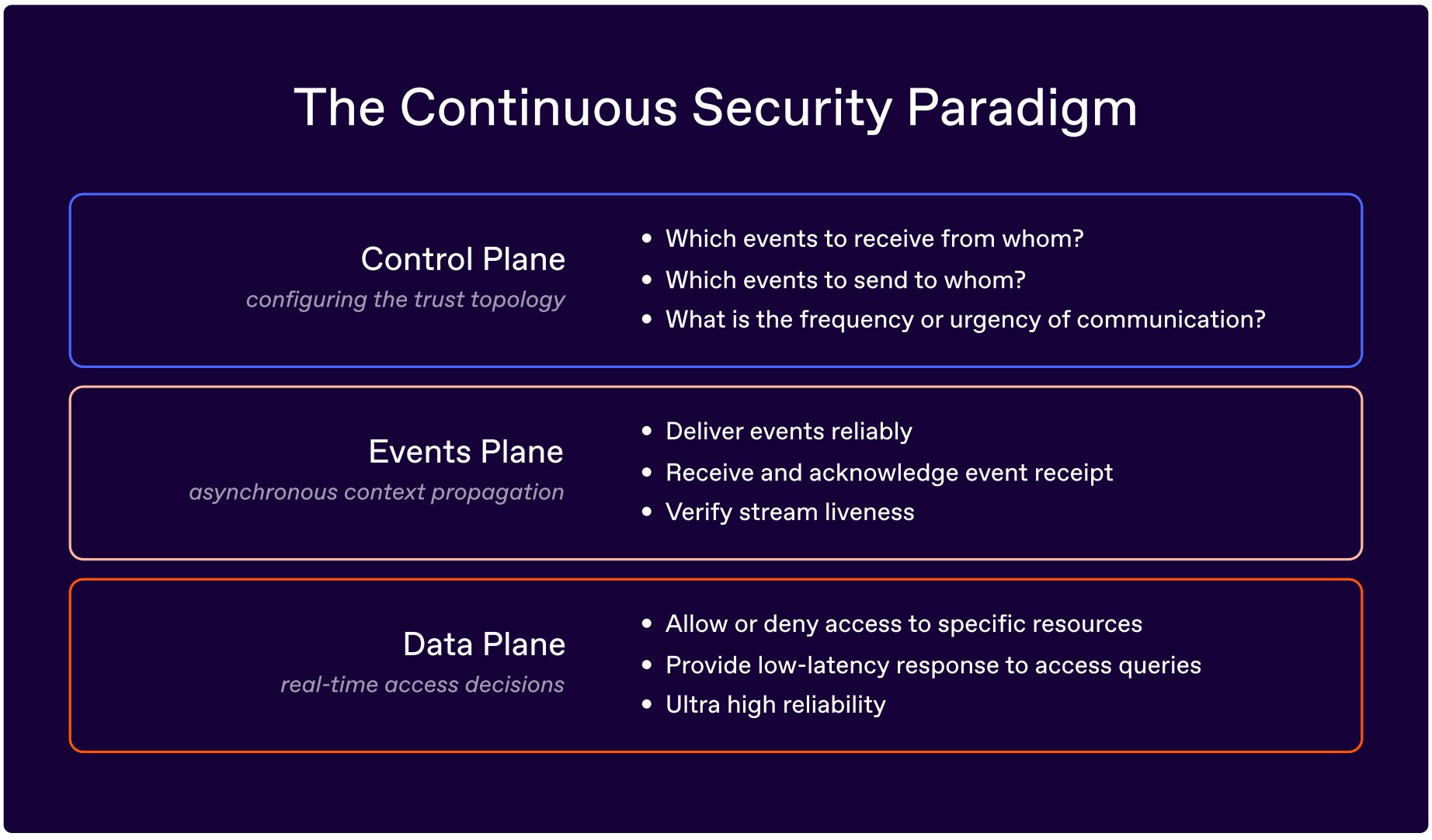
- Ensure that data required for making access decisions at any node is available as soon as it changes, regardless of which node that data originates from. This can be achieved using asynchronous communication of such changes.

- If the data you have changes, you need to ensure it is communicated to the nodes that need it in order to make decisions.

If you are simply relying on access tokens, you need to make sure that nodes that rely on the validity of the access token know in advance if the access token should be considered invalid due to changes in other systems (e.g.,a user being terminated). That way, when a user shows up with that token, it is immediately discarded and access is denied.

Achieving continuous security

CSP envisions three planes:



Of this, the Control Plane and Events Plane can be implemented using SSF.

Bringing CSP to reality: Implementing CAEP

Establishing the trust topology

To implement CSP, one needs to first establish the trust topology between the nodes in your network:

01

Which events need to be sent by a particular node to another particular node?

02

Which events should a particular node expect from another particular node?

03

How frequently should the events be sent?

- **Determining events to send and receive:** SSF can help with the first two things:
  - As an SSF Transmitter, you can specify which events you support through the Transmitter Configuration Metadata
  - As an SSF Receiver, you can request the creation of a stream from the Transmitter, which specifies the types of events that you want (i.e., the `events_requested` parameter in the stream creation request. The Transmitter will commit to sending a subset of these events in the `events_delivered` parameter of the response. As an SSF Receiver, you can decide whether or not the event set is sufficient for you, or if you cannot continue. If the event set is insufficient and you cannot continue, then you should delete the stream and take appropriate error handling steps.
- **Establishing trust:** A good way to protect the SSF Transmitter API is by using OAuth. You can configure your SSF Transmitter to trust an OAuth Server, and specify the OAuth Server and token scopes that you require in an “OAuth Protected Resource Metadata” (OPRM) document. See RFC 9728 for more details. If you are developing an SSF Receiver, then in order to set up the trust, you need to obtain a token from the OAuth server that you can present to the Transmitter when making SSF API calls. This is typically done in an admin console, where the user who can obtain tokens with the appropriate scopes from the OAuth server is already logged into your SSF Receiver.
- **Determining policy:** CAEP is a non-prescriptive standard. CAEP events signify changes at the Transmitter, but do not command or instruct the Receiver to take a specific action. This means that the Receiver has to implement its own policy to determine how to handle the events. Part of establishing the trust topology is to define what each node must do in response to receiving a specific type of event from another node. The policy may depend upon a number of factors:
  - The event type
  - The sender of the event
  - The subject of the event
  - Other data related to the subject, event, and the resources it might impact.

Sending events

As an SSF Transmitter, once you have committed to sending events to a Receiver, you need to ensure the timely delivery of events. You need to determine the following:

- How will you determine that an event needs to be sent to a Receiver? Typically, you will have internal event triggers (e.g.,an API method being called, or an internal queue receives an item that results in the need to send the event). However, in some cases, you might have to monitor a data source in order to determine whether a value has changed and, as a result, it needs to be communicated. In the latter case, you need to establish a polling cadence in order to send those events.
- You will need to determine a service level objective (SLO) for when the events will be sent. This SLO may or may not need to be communicated to the Receiver, but it can be something that you can use to monitor the health of your Transmitter.
- Depending upon the delivery methods your Transmitter supports, you will need to have the appropriate queueing infrastructure for poll delivery of events, or the appropriate retry logic in case a push delivery fails.

Propagating events

An SSF Receiver may also generate events as a result of receiving one. For example, if a policy engine receives an event from an Extended Detection and Response (XDR) system about an increase in a user's risk, it may generate session revocation events to all applications that depend on it for such signaling. Some of those applications may generate events of their own in response, so one has to make sure that any node in the network doesn't take multiple actions in response to the same originating event. This is achieved through correlating the `txn` claim in SSF events. If your node generates events in response to receiving an event, then you must copy the `txn` value from the incoming event to the outgoing events you generate. As a result, if you receive an event with a `txn` value that you have already processed, you do not need to process that event.

Receiving events

As an SSF Receiver, you need to ensure that you can reliably receive events and take the appropriate action upon receiving them. The receipt of the event serves as the internal trigger for other actions that you need to take, e.g.,updating a database or cache. You need to respond appropriately to the Transmitter to indicate that you have received the event. You need to ensure an audit mechanism to verify that you have processed every event that you have received, in order to make sure that incoming events aren't being dropped. This can be a good measure to monitor your service's health.

You should also request verification events periodically to monitor the stream liveness. You can provide appropriate alerting if the stream is determined to be no longer active.

Common use cases

There are a number of use cases being implemented in the industry today. They include:

Session revocation

An application or an identity service (such as a single sign-on identity provider, a policy engine, or an Identity Governance and Administration (IGA) service) may communicate that a particular user or a particular session of a user is terminated at their end, as indicated by the subject of the event. The Receiver of the event may:

- Terminate their sessions for the same user or if they can, terminate the specific session.
- Evaluate whether the user's session needs to be terminated based on other data that they have received about the user. This can happen when an application indicates abnormal activity to a policy engine, and the policy engine evaluates whether similar activity, device risks, or user risks have been flagged by other services for the same user. Based on the result of the evaluation, the Receiver may terminate their own session, and transmit session revocation events to other Receivers.

Device compliance change

Services that manage devices for users on behalf of their employers watch those devices for compliance with the enterprise policy. Devices managed using such services typically periodically check in with the service, whereby the service can check the device "posture". A posture that violates policy is understood by the device management service at the time of check-in, or if the device fails to check in. Using the CAEP `device-compliance-change` event, the device management service can signal both the change of a device from being compliant to non-compliant or vice-versa. The subject of this event could just be the user (in which case it is assumed to apply to all devices the user is using), or it might include a specific device (either in addition to or instead of the user identity). The Receivers of this event may:

- Terminate user access to their service, or
- Permit only limited access until the device becomes compliant again.

A Receiver may be informed of the user's device being used in the session at the time of session establishment. In this case, the Receiver can correlate the device identifier in the `device-compliance-change` event subject to the identifier it has. If not, the Receiver can take a defensive position of assuming the user's device (which is unknown to the Receiver) has become non-compliant. To ascertain compliance again, the Receiver in that case will have to re-authenticate the user, with the assumption that the identity provider will verify the device compliance at the time of re-authentication.

Credential change

A user may change their password or change the strong authentication mechanism, device, or phone number. Any of these events can result in the service managing the user's credential (typically the identity provider, or credential provider) can signal the CAEP `credential-change event`. Receivers of this event may:

- Force the user to re-authenticate
- Permit limited access until the user re-authenticates

Risk level change

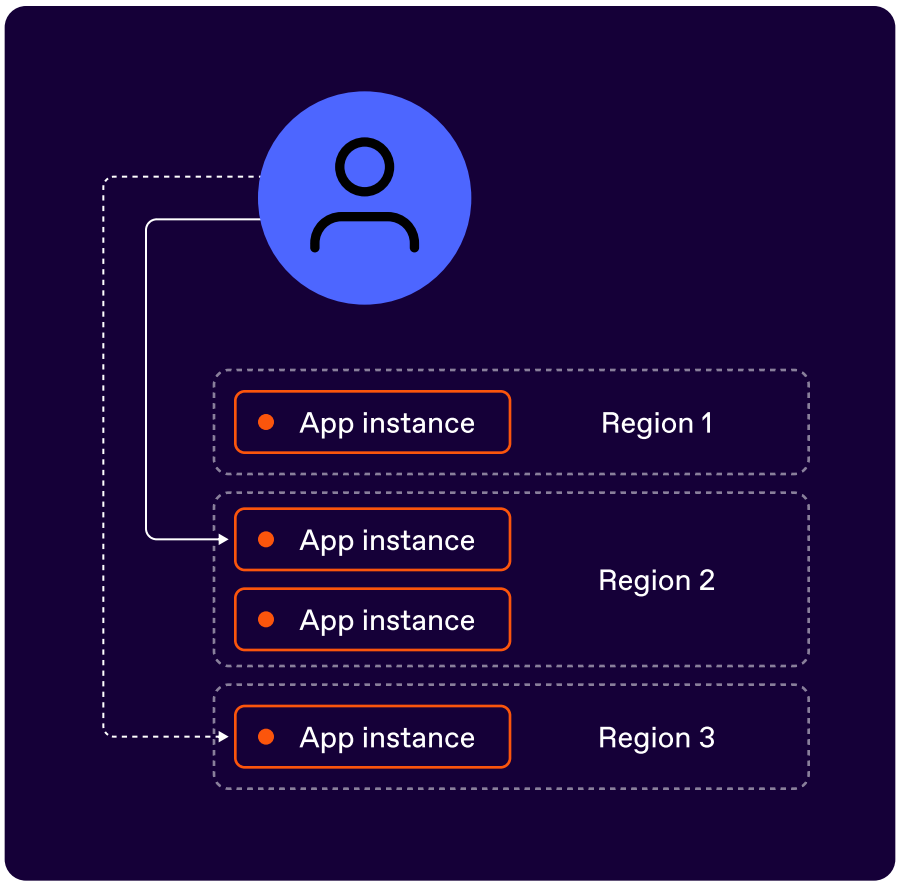
An application being used by the user, a monitoring service, or an XDR service can detect that the user's risk has changed. This could be based on a variety of factors, such as user behavior, device risk, environmental risk (i.e., the user is in an untrusted location), etc. The service that detects such change can send the `risk-level-change` event. A Receiver may take appropriate action, such as:

- Terminate the user session
- Permit limited access
- Add the event to its own risk assessment, and determine the appropriate action on cumulative risk

Example: Implementing session revocation in an application

Application description

An application is composed of a globally distributed set of nodes. It uses access tokens as a means of authenticating users. The access tokens identify the user and their permissions within the application. The application uses an external IdP to issue ID Tokens, which it then converts to its own access tokens for session identification. The application does not persist a list of logged-in users in its database, but assumes that anyone presenting a valid access token is logged in (as long as the token has not expired). The following picture illustrates the application architecture:



In the above diagram, the same user may at one point in time access an app instance in one region, whereas at some other point in time might access an app instance in another region.

Session revocation event

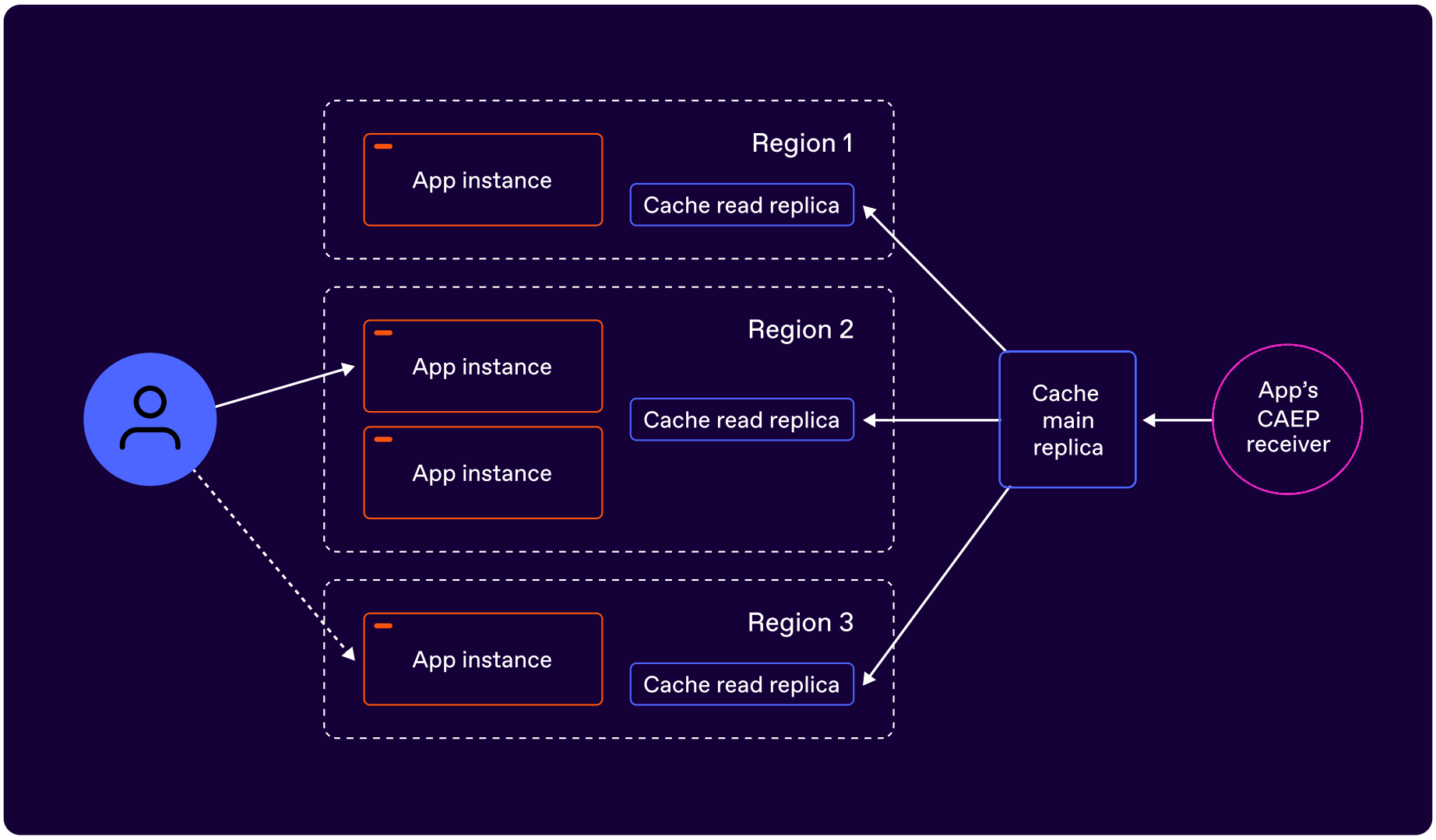
The IdP supports CAEP and sends a `session_revoked` event to the application when a user's session is revoked at the IdP. The application policy is to revoke its local session when that happens. The content of the session revoked event from the IdP is:

```JSON
{
  "iss": "https://idp.example.com/12345/",
  "jti": "24c63fb56e5a2d77a6b512616ca9fa24",
  "iat": 1615305159,
  "aud": "https://myorg.example/caep",
  "txn": "8675309",
  "sub_id": {
    "format": "email",
    "email": "user@idp.example"
  },
  "events": {
    "https://schemas.openid.net/secevent/
caep/event-type/session-revoked": {
      "reason_admin": {
        "en": "Policy Violation: C076E822"
      },
      "event_timestamp": 1615304991
    }
  }
}
```

Architecture for handling session revocation events

Since the application has no persistent record of a user logging in, when the application receives the `session_revoked` event, it has no way to correlate that with existing login sessions. To ensure it can revoke live sessions, it creates a distributed cache, which has a read replica at every node where the application is running. It has one main replica where it receives CAEP events. This is propagated to all the read replicas to achieve eventual consistency.

This architecture is described below.



As seen in the diagram above, the CAEP Receiver (an SSF Receiver used to receive CAEP events) of the application inserts the received event into the main replica of the cache. This gets propagated to the read replicas as soon as possible to reach eventual consistency. Each entry in the cache can be the entire `session_revoked` event, or it could be just the relevant portion of the event, which is the "issued at time" and the subject identifier. For example, the cache item can be:

```JSON
{
  "iat": 1615305159,
  "id": "user@idp.example"
}
```

Application logic

When any node within the application receives an access token, it verifies that the local copy of the cache does not contain an event for that user. If it does, then it compares the “issued at time” of the access token with the `iat` of the cache entry. If the access token is older than the entry in the cache, then it rejects the request as unauthorized. The normal application logic for handling unauthorized users then kicks in to guide the user through the login process again.

Extending to other event types

The above example illustrates how session revocation may be implemented. To extend to other event types, say the application’s permissions need to be changed based on a `token_claims_change` event received from an IdP, then the architecture and logic remain the same, but the cache entries are extended to contain an “event type”, and the relevant details (e.g.,updated permissions) that are specific to that event type. So an entry created as a consequence of receiving a `token_claims_change` event can be:

```
JSON

{
  "iat": 1615305159,
  "id": "user@idp.example",
  "https://schemas.openid.net/secevent/caep/
event-type/token-claims-change" : {
    "permissions": [ "admin", "user"]
  }
}
```

After this is implemented, the application will override any permissions within the access token with the permissions defined in the cache, if a cache entry is found. That way, if the user’s permissions have changed within the duration of a session, the updated permissions will be effective instantaneously across the application.

ABOUT SGNL

SGNL is the only enterprise platform purpose-built for Continuous Identity—trusted by Fortune 500 enterprises worldwide to eliminate standing access to cloud environments, code, and critical applications.

SGNL delivers continuous, context-aware protection for sensitive data, infrastructure, cloud workloads, and applications, and now extends that protection to AI through the SGNL MCP Security Gateway—enforcing real-time policies on AI agent actions. By centralizing context, real-time enforcement, and enterprise-wide orchestration through open standards like CAEP and the Shared Signals Framework, SGNL enables Zero Standing Privilege across the entire identity footprint and makes Zero Trust an operational reality.

SGNL named a Cool Vendor in the 2025 Gartner® Cool Vendors™ in Identity-First Security

REQUEST A DEMO

Gartner, Cool Vendors in Identity-First Security, 24 October 2025

GARTNER is a registered trademark and service mark of Gartner, Inc. and/or its affiliates in the U.S. and internationally and is used herein with permission. All rights reserved.