Workload Identity in Multi System Environments
Internet-Draft
Intended status: Informational
Expires: 12 October 2026

A. Schwenkschuster
SPIRL
Y. Rosomakho
Zscaler

10 April 2026

Workload Identity Practices
draft-ietf-wimse-workload-identity-practices-04

Abstract

This document describes industry practices for providing secure identities to workloads in container orchestration, cloud platforms, and other workload platforms. It explains how workloads obtain credentials for external authentication purposes, without managing long-lived secrets directly. It does not take into account the standards work in progress for the WIMSE architecture [WIMSE-ARCH] and other protocols, such as [WIMSE-HTTPSIG].

Table of Contents

1. Introduction

Just like people, workloads need identifiers and associated credentials to authenticate with other systems, such as databases, web servers, or other workloads. The challenge for workloads is to obtain a credential that can be used to authenticate with these resources without managing secrets directly, for instance, an OAuth 2.0 access token.

The common use of the OAuth 2.0 framework [OAUTH-FRAMEWORK] in this context poses challenges, particularly in managing credentials. To address this, the industry has shifted to a federation-based approach where credentials of the underlying workload platform are used to authenticate to identity providers, which in turn, issue credentials that grant access to resources.

Traditionally, workloads were provisioned with static client credentials (e.g., passwords, API keys) and used the corresponding flow as described in Section 1.3.4 of [OAUTH-FRAMEWORK] to retrieve an OAuth 2.0 access token. This model presents a number of security and maintenance issues. Secrets must be provisioned and rotated, which requires either automation to be built, or periodic manual effort. Secrets may be stolen and used by attackers to impersonate the workload. Flows outside of the OAuth 2.0 framework (such as direct API keys or HTTP basic authentication) suffer from the same issues.

Instead of provisioning secret material to the workload, one solution to this problem is to attest the workload by using its underlying platform. Many platforms provision workloads with a credential, such as a JWT [JWT]. Cryptographically signed by the platform’s issuer, this credential attests the workload and its attributes.

Figure 1 illustrates a generic pattern that is seen across many workload platforms, more concrete variations are found in Section 4.

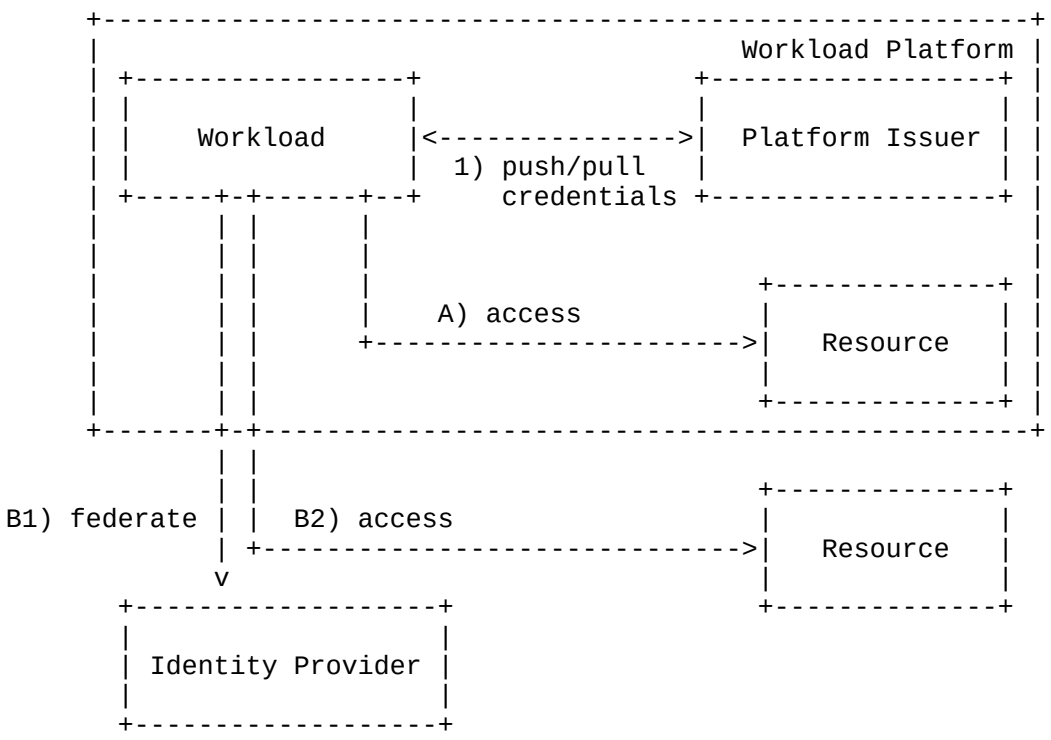

Figure 1: Generic workload identity pattern

The figure outlines the following steps which are applicable in any pattern.

* 1) The platform issues a credential to represent the workload identity after verification of workload environment and attributes. The way this is achieved varies by platform, for instance, the credential can be pushed to the workload or pulled by the workload. A workload may obtain multiple credentials from the platform, each with its own audience and lifetime, tailored to the specific resource or Identity Provider it needs to interact with. Credentials SHOULD have as small a set of audiences as possible to limit the scope of any single credential. See Section 5.7 for more details and security implications.

* A) The credential can give the workload direct access to resources within the platform or the platform itself, for example to perform infrastructure operations. The credential used for this step SHOULD be scoped specifically to the platform resource being accessed.

* B1) The workload uses a credential to federate to an Identity Provider. This step is optional and only needed when accessing outside resources. The credential used for federation SHOULD carry the Identity Provider as its sole audience and SHOULD NOT be the same credential used for platform access in step A). The Identity Provider validates the platform-issued credential, and in return, issues a new credential, such as an OAuth 2.0 access token, that the workload can use to access resources in the Identity Provider's domain.

* B2) Using the credential obtained at step B1, the workload accesses resources outside of the platform.

Accessing different outside resources may require the workload to repeat steps B1) and B2), federating to multiple Identity Providers. It is also possible that step 1) needs to be repeated, for instance in situations where the platform-issued credential is scoped to accessing a certain resource or federating to a specific Identity Provider.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Delivery Patterns

Credentials can be provisioned to the workload by different mechanisms, each of which has its own advantages, challenges, and security risks. The following section highlights the pros and cons of common solutions. Security recommendations for these methods are covered in Section 5.1.

3.1. Environment Variables

Injecting the credentials into the environment variables allows for simple and fast deployments. Applications can directly access them through system-level mechanisms, e.g., through the env command in Linux. Note that environment variables are static in nature in that they cannot be changed after application initialization.

3.2. Filesystem

Filesystem delivery allows both container secret injection and access control. Many solutions find the main benefit in the asynchronous provisioning of the credentials to the workload. This allows the workload to run independently of the credentials update, and to access them by reading the file.

Credential rotation requires a solution to detect soon-to-expire secrets as a rotation trigger. One practice is that the new secret is renewed _before_ the old secret is invalidated. For example, the solution can choose to update the secret an hour before it is invalidated. This gives applications time to update without downtime.

Because credentials are written to a shared filesystem, the solution is responsible for ensuring atomicity when updating them. Writes SHOULD be performed in a way that prevents workloads from observing a partially written file (for example by writing to a temporary file and renaming it atomically). Solutions SHOULD also perform a flush operation immediately after the update to minimize the chance of race conditions and ensure durability.

3.3. Local APIs

In this pattern, the workload obtains credentials by communicating with a local API exposed by the credential issuer. Implementations commonly use UNIX domain sockets (e.g., SPIFFE), loopback interfaces, or link-local "magic addresses" 169.254.169.254 commonly used for cloud provider Instance Metadata Services as the transport mechanism.

Local APIs support re-provisioning of updated credentials, either on demand or through persistent connections that enable the issuer to push new credentials. This enables the use of short-lived, narrowly scoped credentials, improving security posture compared to long-lived secrets.

The security of this approach relies heavily on network isolation to prevent unauthorised access to the local API. In addition, the pattern requires client-side code that is specific to the exposed API, which may introduce portability challenges across platforms and providers. Further security considerations for local APIs are discussed in Section 5.1.3.

4. Practices

The following practices outline more concrete examples of platforms, including their delivery patterns.

4.1. Kubernetes

In Kubernetes, machine identity is implemented through "service accounts" [KubernetesServiceAccount]. Service accounts can be explicitly created, or a default one is automatically assigned. Service accounts use JSON Web Tokens ([JWT]) as their credential format, with the Kubernetes Control Plane acting as the signer.

Service accounts serve multiple authentication purposes within the Kubernetes ecosystem. They are used to authenticate to Kubernetes APIs, between different workloads and to access external resources. This latter use case is particularly relevant for the purposes of this document.

To programmatically use service accounts, workloads can:

* Have the token "projected" into the file system of the workload. This is similar to volume mounting in non-Kubernetes environments, and is commonly referred to as "projected service account token".

* Use the Token Request API [TokenRequestV1] of the control plane. This option, however, requires an initial projected service account token as a means of authentication.

Both options allow workloads to:

* Specify a custom audience. Possible audiences can be restricted based on policy.

* Specify a custom lifetime. Maximum lifetime can be restricted by policy.

* Bind the token lifetime to an object lifecycle. This allows the token to be invalidated when the object is deleted. For example, this may happen when a Kubernetes Deployment is removed from the server. Note that invalidation is only detected when the Token Review API [TokenReviewV1] of Kubernetes is used to validate the token.

* Obtain multiple tokens, each with its own customized audience and lifetime. For example, a workload may obtain one token audienced for the Kubernetes API server, another for an internal service, and yet another for federation with an external Identity Provider. Tokens SHOULD have a minimal set of audiences; see Section 5.7 for more details and security implications.

To validate service account tokens, Kubernetes allows workloads to:

* Make use of the Token Review API [TokenReviewV1]. This API introspects the token, makes sure it hasn't been invalidated and returns the claims.

* Mount the public keys used to sign the tokens into the file system of the workload. This allows workloads to validate a token's signature without calling the Token Review API.

* Optionally, a JSON Web Key Set [JWK] is exposed via a web server. This allows the Service Account Token to be validated outside of the cluster and access to the actual Kubernetes Control Plane API.

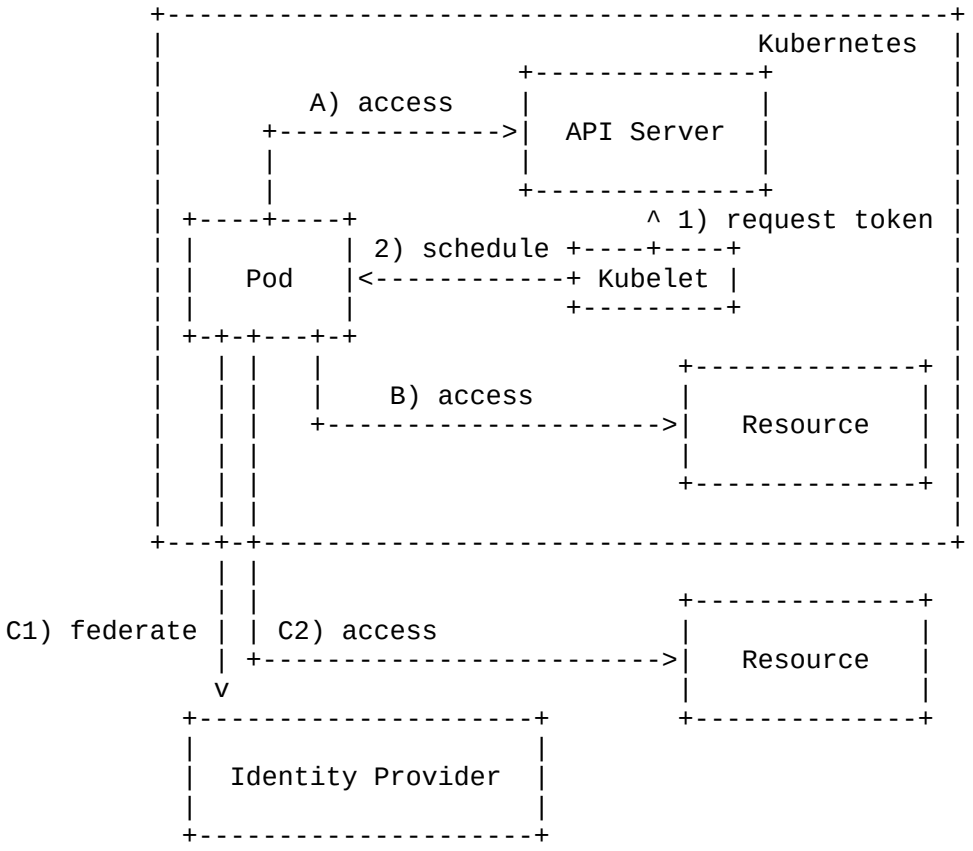

Figure 2: Kubernetes workload identity in practice

The steps shown in Figure 2 are:

* 1) The kubelet is tasked to schedule a Pod. Based on configuration, it requests one or more Service Account Tokens from the Kubernetes API server, each scoped to its intended use, for example with a distinct audience.

* 2) The kubelet starts the Pod and, based on the configuration of the Pod, delivers the token(s) to the containers within the Pod.

Now, the Pod can use the tokens to:

* A) Access the Kubernetes Control Plane, using a token audienced for the API server, considering it has access to it.

* B) Access other resources within the cluster, for instance, other Pods, using a token audienced for the target resource.

* C) Access resources outside of the cluster:

* C1) The application within the Pod uses a Service Account Token audienced for the external Identity Provider to federate to that Identity Provider outside of the Kubernetes Cluster. This token SHOULD NOT be the same token used for steps A or B. The Identity Provider validates the token and issues a new credential to the workload, such as an OAuth 2.0 access token.

* C2) Using the credential issued in step C1, the application within the Pod accesses resources outside of the cluster.

As an example, the following JSON illustrates the claims contained in a Kubernetes Service Account token.

```
   {
     "aud": [  # matches the requested audiences, or the API server's default audiences w
hen none are explicitly requested
       "https://kubernetes.default.svc"
     ],
     "exp": 1731613413,
     "iat": 1700077413,
     "iss": "https://kubernetes.default.svc",  # matches the first value passed to the --
service-account-issuer flag
     "jti": "ea28ed49-2e11-4280-9ec5-bc3d1d84661a",  # ServiceAccountTokenJTI feature mus
t be enabled for the claim to be present
     "kubernetes.io": {
       "namespace": "my-namespace",
       "node": {  # ServiceAccountTokenPodNodeInfo feature must be enabled for the API se
rver to add this node reference claim
         "name": "127.0.0.1",
         "uid": "58456cb0-dd00-45ed-b797-5578fdceaced"
       },
       "pod": {
         "name": "my-workload-69cbfb9798-jv9gn",
         "uid": "778a530c-b3f4-47c0-9cd5-ab018fb64f33"
       },
       "serviceaccount": {
         "name": "my-workload",
         "uid": "a087d5a0-e1dd-43ec-93ac-f13d89cd13af"
       },
       "warnafter": 1700081020
     },
     "nbf": 1700077413,
     "sub": "system:serviceaccount:my-namespace:my-workload"
   }
```

Figure 3: Example Kubernetes Service Account Token claims

4.2. Secure Production Identity Framework For Everyone (SPIFFE)

The Secure Production Identity Framework For Everyone, also known as SPIFFE [SPIFFE], is a Cloud Native Computing Foundation (CNCF) project that defines a "Workload API" to deliver machine identity to workloads. Workloads can retrieve identity credentials in one of two forms:

* X509-SVID, a X.509 certificate containing the workload's SPIFFE ID in the Subject Alternative Name (SAN) URI field, along with the corresponding key pair.

* JWT-SVID, a signed JWT containing the workload's SPIFFE ID in the sub claim. The Workload API does not require clients to authenticate themselves.

Instead, the API implementation identifies workloads by collecting contextual information from the environment, such as process attributes, kernel metadata, or orchestrator-provided labels. This out-of-band identification allows workloads to obtain their identity credentials without needing a pre-existing secret, avoiding the bootstrapping problem of requiring a credential to obtain a credential.

Workloads may request multiple JWT-SVIDs, each with a distinct audience, to interact with different resources or Identity Providers. As with all patterns in this document, it is best practice to use a separate credential for each target; see Section 5.7 for details.

For validation, SPIFFE defines a "trust bundle" per trust domain. A trust bundle is a set of public keys encoded in JWK format [JWK] that can be used to validate credentials. For JWT-SVIDs, the bundle contains signing keys identified by a use value of jwt-svid. For X509-SVIDs, the bundle contains CA certificates identified by a use value of x509-svid. Trust bundle contents can be retrieved from the Workload API or from a dedicated SPIFFE Bundle Endpoint (see [SPIFFE]).

The following figure illustrates how a workload can use its SPIFFE identity to access a protected resource outside of the trust domain. The example uses a JWT-SVID, but using an X509-SVID is also possible.

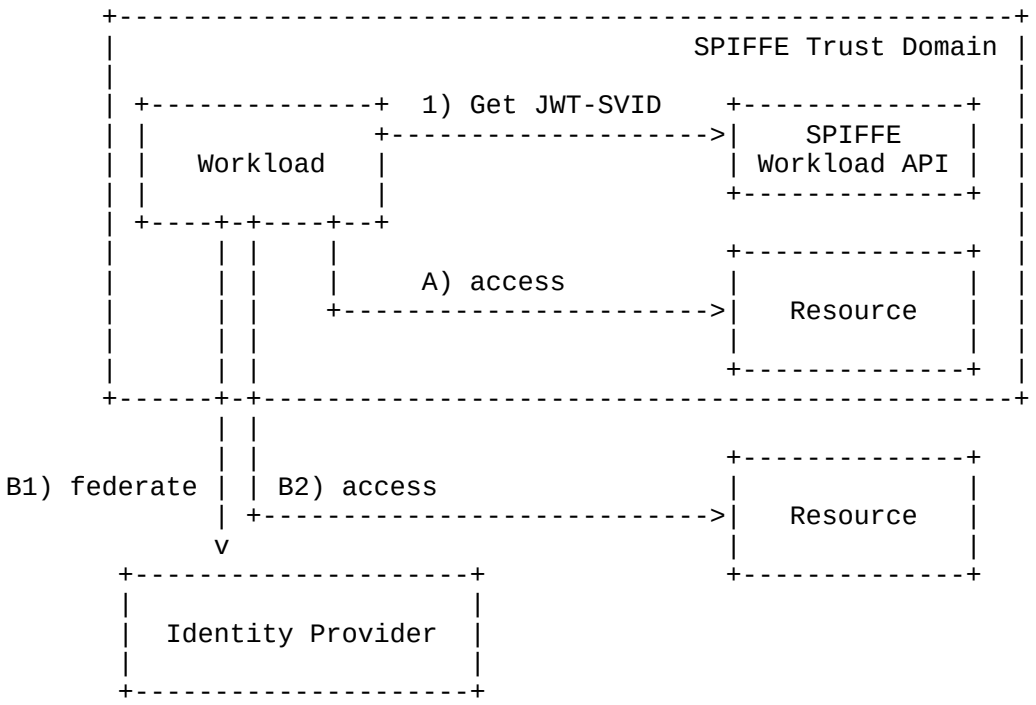

Figure 4: Workload identity in SPIFFE

The steps shown in Figure 4 are:

* 1) The workload requests one or more JWT-SVIDs from the SPIFFE Workload API, each with a distinct audience matching its intended use.

* A) A JWT-SVID audienced for the target resource can be used to directly access resources or other workloads within the same SPIFFE Trust Domain.

* B1) To access resources protected by other Identity Providers, the workload uses a JWT-SVID audienced for the Identity Provider to federate. This SHOULD be a separate JWT-SVID from the one used in step A). The Identity Provider validates the JWT-SVID and issues a new credential such as an OAuth 2.0 access token, to the workload.

* B2) Using the credential issued in step B1, the workload can access resources outside of its trust domain.

Here are example claims for a JWT-SVID:

```
{
  "aud": [
    "external-authorization-server"
  ],
  "exp": 1729087175,
  "iat": 1729086875,
  "sub": "spiffe://example.org/myservice"
}
```

4.3. Cloud Providers

Workloads in cloud platforms can have any shape or form. Historically, virtual machines were the most common. The introduction of containerization brought hosted container environments or Kubernetes clusters. Containers have evolved into serverless offerings. Regardless of the actual workload packaging, distribution, or runtime platform, all these workloads need identities.

The biggest cloud providers have established the pattern of an "Instance Metadata Endpoint". Aside from allowing workloads to retrieve metadata about themselves, it also allows them to receive identity. The credential types offered can vary. JWT, however, is the one that is common across all of them. The issued credential provides proof to anyone it is being presented to that the workload platform has attested the workload and it can be considered authenticated.

Within a cloud provider, the issued credential can often directly be used to access resources of any kind across the platform, making integration between the services straightforward. From the workload perspective, no credential needs to be issued, provisioned, rotated or revoked, as everything is handled internally by the platform.

This is not true for resources outside of the platform, such as on-premise resources, generic web servers or other cloud provider resources. Here, the workload first needs to federate to the Secure Token Service (STS) of the respective cloud, which is effectively an Identity Provider. The STS issues a new credential with which the workload can then access resources.

This pattern also applies when accessing resources in the same cloud but across different security boundaries (e.g., different account or tenant). The actual flows and implementations may vary in these situations though.

When a workload needs to access both internal platform resources and external resources, it SHOULD obtain separate credentials for each purpose. The credential used for internal platform access (step A) SHOULD NOT be reused for federation to an external STS (step B1), as these represent different trust and audience boundaries. The workload may need to contact the Instance Metadata Service multiple times to obtain appropriately scoped credentials.

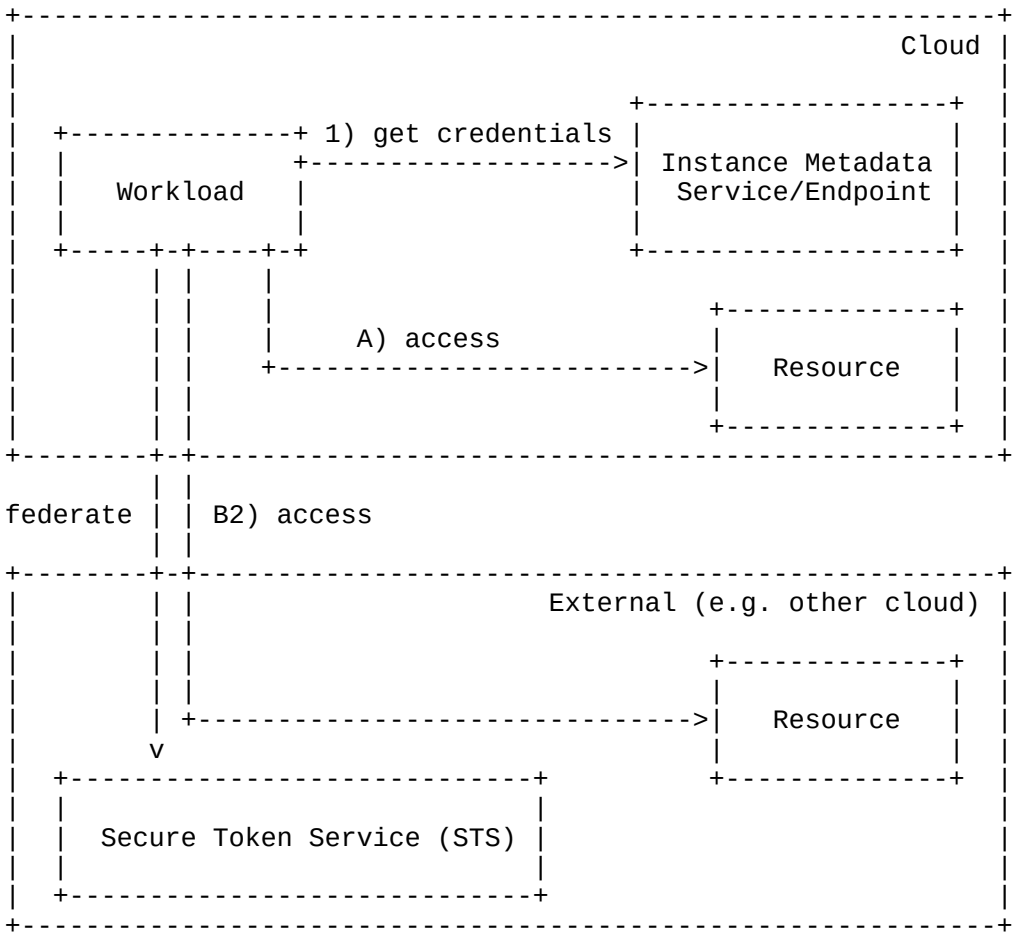

Figure 5: Workload identity in a cloud provider

The steps shown in Figure 5 are:

* 1) The workload retrieves one or more identity credentials from the Instance Metadata Service or Endpoint. This endpoint exposes an API and is available at a well-known, but local-only location such as 169.254.169.254. Each credential SHOULD be scoped to its intended use with a distinct audience.

When the workload needs to access a resource within the cloud (e.g., located in the same security boundary; protected by the same issuer as the workload identity):

* A) The workload directly accesses the protected resource with a credential scoped for that resource, as issued in Step 1.

When the workload needs to access a resource outside of the cloud (e.g., different cloud; same cloud, but different security boundary):

* B1) The workload uses a separate cloud-issued credential, audienced for the external STS, to federate to the Secure Token Service of the other cloud/account. This credential SHOULD NOT be the same as the one used in step A). The STS validates the credential and issues a new credential, such as an access token to the workload.

* B2) Using the credential issued in step B1, the workload can access the resource outside, assuming the credential has the necessary permissions.

4.4. Continuous Integration and Deployment Systems

Continuous integration and deployment (CI-CD) systems allow their pipelines (or workflows) to receive an identity at runtime. It is a common task to upload build outputs and other artifacts to external resources. For this, federation to external Identity Providers is often necessary.

As with other platforms, CI-CD workloads may obtain multiple tokens from the platform, each with a distinct audience for the specific resource or Identity Provider it needs to interact with.

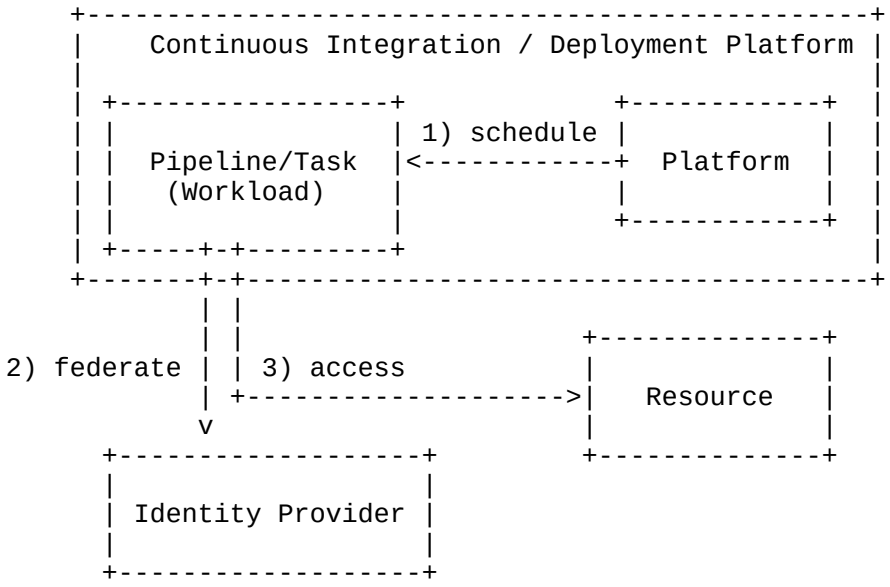

Figure 6: OAuth2 Assertion Flow in a continuous integration/ deployment environment

The steps shown in Figure 6 are:

* 1) The CI-CD platform schedules a workload (pipeline or task). Based on configuration, a Workload Identity is made available by the platform.

* 2) The workload uses the platform-issued credential to federate to an Identity Provider, which validates the credential and issues a new credential, such as an access token, for the workload.

* 3) The workload uses the issued credential to access resources. For instance, an artifact store to upload compiled binaries, or to download libraries needed to resolve dependencies. It is also common to access actual infrastructure as resources to make deployments or changes to it.

While token structure is vendor-specific, all tokens contain claims carrying the basic context of the executed tasks, such as source code management data such as git branch, initiation context and more.

4.5. Service Meshes

Service meshes provide infrastructure-level workload identity and secure communication for applications through sidecar proxies deployed alongside each workload. In a service mesh, workload identity is typically implemented using X.509 certificates issued by the service mesh. Service meshes handle identity credential provisioning to sidecar proxies rather than directly to application workloads. The sidecar intercepts network traffic and handles authentication transparently to the application code.

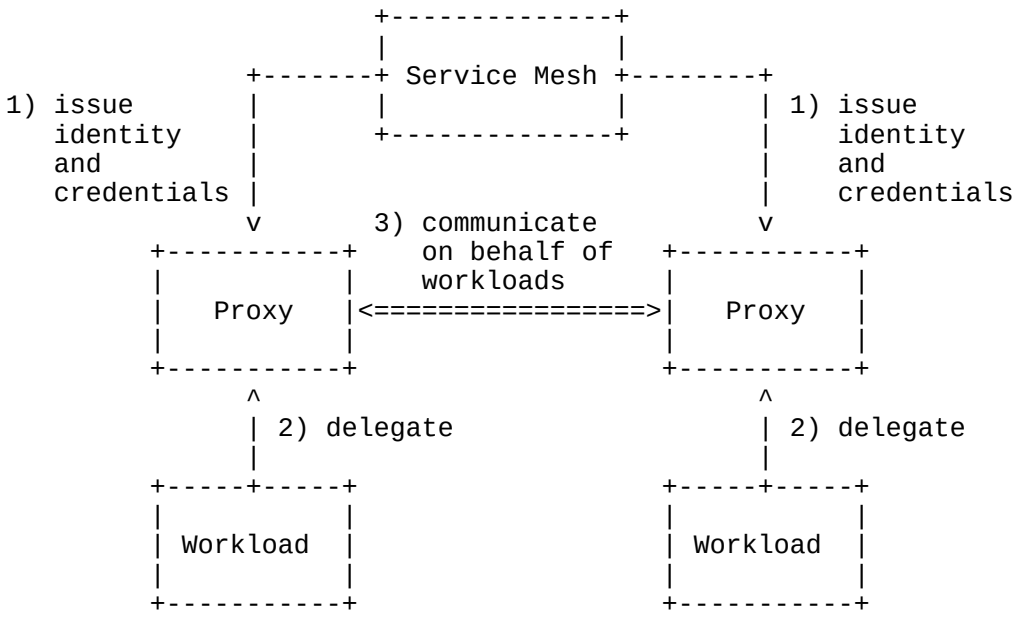

Figure 7: Simple service mesh communication between 2 workloads

The steps shown in Figure 7 are:

* 1) The Service Mesh issues identity credentials to proxies. For X.509-based meshes, this consists of an X.509 certificate containing the workload’s identity along with the associated key pair.

* 2) The proxies act on behalf of workloads that delegate their communication to them. In above figure each workload has its own proxy that solely represents it and no other workload.

* 3) The proxies communicate with each other on behalf of the workloads they represent. This communication includes authentication aspects, for instance mutual TLS using X.509 certificates.

In above pattern each workload has a specific sidecar. An alternative deployment is to share proxies between workloads. This often results in a single proxy on each node acting on behalf of all workloads on the node.

5. Security Considerations

All security considerations in section 8 of [OAUTH-ASSERTION] apply.

5.1. Credential Delivery

5.1.1. Environment Variables

Leveraging environment variables to provide credentials presents many security limitations. Environment variables have a wide set of use cases and are observed by many components. They are often captured for monitoring, observability, debugging and logging purposes and sent to components outside of the workload. Access control is not trivial and does not achieve the same security results as other methods. Additionally, environment variables may be spoofed or altered by other processes running on the same host, making them an unreliable transport for credentials in environments where process isolation is not strictly enforced.

This approach should be limited to non-production cases where convenience outweighs security considerations, and the provided secrets are limited in validity or utility. For example, an initial secret might be used during the setup of the application.

5.1.2. Filesystem

* 1) Access control to the mounted file should be configured to limit reads to authorized applications. Linux supports solutions such as DAC (uid and gid) or MAC (e.g., SELinux, AppArmor).

* 2) Mounted shared memory should be isolated from other host OS paths and processes. For example, on Linux this can be achieved by using namespaces.

5.1.3. Local APIs

Local APIs often operate in clear-text such as unencrypted HTTP without any confidentiality or integrity protection. Privileged component on the machine or in the infrastructure can be able to eavesdrop on the connection and the credential within it.

Mitigating measures are required to mitigate a particular variant of Server-Side Request Forgery attacks against local APIs. For example, requiring a specific header that cannot be controlled externally or preventing the use of link-local IPs, including through redirects. See Section 5.1.4 for details.

Adequate assurance that the identity represents the workload is required to make sure unauthorized access is denied and credentials are not issued to other parties when the Local API is unauthenticated. What constitutes adequate assurance depends on the security requirements of the deployment. Introspection of the platform, like in SPIFFE or cloud providers, can be used to identify workloads and grant access. The more fine-grained and strict this verification, the smaller the attack surface. For instance, allowing access by IP or other machine-global identifiers permits any process to receive the identity, while including user ID or other process-scoped identifiers prevents this broader access.

The potential for denial-of-service attacks against Local APIs need to be taken into account and protective measures should be implemented. Depending on the platform these attacks can affect other workloads and their ability to receive a platform credential.

5.1.4. Application Interaction with Credential Sources

Implementations MUST assume that application vulnerabilities can expose workload credentials even when platform isolation is correctly configured. Attackers commonly exploit the workload itself to retrieve credentials rather than accessing the credential service directly.

For example, untrusted input may be used to manipulate file paths when credentials are mounted on a filesystem, or to trigger requests to local credential endpoints such as metadata or workload APIs (for example via server-side request forgery). Similarly, command execution or unintended outbound requests may result in bearer tokens or proof-of-possession key material being disclosed.

Workloads therefore MUST treat credential locations as sensitive security boundaries. Untrusted input MUST NOT influence how credential files are accessed or how local credential APIs are

contacted. Implementations SHOULD minimise which components can access credentials and prefer proof-of-possession credentials over bearer tokens where supported. Failure to minimise credential access increases the attack surface by allowing more code paths to interact with sensitive material. Failing to use proof-of-possession credentials where available means that stolen bearer tokens can be replayed by an attacker from any location.

These risks exist even when credential services are reachable only locally, since compromise often occurs through application behaviour rather than network access to the credential provider.

5.2. Token typing

Issuers SHOULD strongly type the issued tokens to workloads via the JOSE typ header and Identity Providers accepting these tokens SHOULD validate the value of it according to policy. See Section 3.1 of [JWT-BCP] for details on explicit typing. Without explicit typing, a token intended for one purpose (e.g., a refresh token or an identity assertion) may be accepted in a context where a different token type is expected, enabling cross-protocol or cross-context token confusion attacks.

Issuers SHOULD use authorization-grant+jwt as a typ value according to [OAUTH-JWT]. For broad support, JWT or JOSE MAY be used by issuers and accepted by authorization servers but it is important to highlight that a wide range of tokens, meant for all sorts of purposes, use these values and would be accepted. Using generic type values such as JWT or JOSE is acceptable only when the deployment cannot support more specific types, for instance due to limitations in existing infrastructure or token libraries. Even in such cases, additional validation of token claims and context is essential to mitigate confusion.

5.3. Custom claims are important for context

Some platform-issued credentials have custom claims that are vital for context and are required to be validated. For example, in a continuous integration and deployment platform where a workload is scheduled for a Git repository, the branch is crucial. A "main" branch may be protected and considered trusted to federate to external authorization servers. But other branches may not be allowed to access protected resources.

Authorization servers that validate assertions SHOULD make use of these claims. Ignoring custom claims may result in overly permissive authorization decisions, such as granting a credential issued for an untrusted branch the same access as one issued for a protected

branch. Platform issuers SHOULD allow differentiation based on the subject claim alone, so that authorization policies can be expressed without requiring deep knowledge of vendor-specific claim structures.

5.4. Token lifetime

Tokens SHOULD NOT exceed the lifetime of the workloads they represent. For example, a workload that has an expected lifetime of one hour should not receive a token valid for two hours or more. A token that outlives its workload may continue to be accepted by relying parties even after the workload (and its associated authorization context) has ceased to exist, enabling unauthorized access if the token is compromised.

Within the scope of this document, where a platform-issued credential is used to authenticate to retrieve an access token for an external authorization domain, short-lived credentials are recommended. Short-lived credentials reduce the window during which a stolen credential can be exploited and limit the need for explicit revocation infrastructure.

5.5. Workload lifecycle and invalidation

Platform issuers SHOULD invalidate tokens when the workload stops, pauses, or ceases to exist and SHOULD offer validators a mechanism to query this status. Without invalidation, tokens for terminated workloads remain usable until their natural expiry, creating a window for unauthorized use. Without a status query mechanism, relying parties have no way to detect that a workload has been removed and must accept the token as is. How these credentials are invalidated and the status is queried varies and is not in scope of this document.

5.6. Proof of possession

Identity credentials SHOULD be bound to workloads, and proof of possession SHOULD be performed when these credentials are used. This mitigates token theft. Without proof of possession, a bearer token intercepted in transit (e.g., via a compromised log, a man-in-the-middle, or SSRF) can be replayed by any party, from any location, for the remaining lifetime of the token. For X.509-based credentials, proof of possession is inherent through the private key associated with the certificate. For JWT-based credentials, the JWT SHOULD be key-bound with an adequate proof-of-key-possession mechanism. Where proof of possession is not supported by the platform or the relying party, deployments SHOULD compensate with shorter token lifetimes, stricter audience scoping, and additional network-level controls such as IP allowlisting or mutual TLS. This proof of possession applies

to both the platform credential and the access token of the external authorization domains.

5.7. Audience

For issued credentials in the form of JWTs, they MUST be audienced using the aud claim. Each JWT SHOULD only carry a single audience. Using multiple audiences in a single token means that any relying party listed in the aud claim can present that token to any other party listed in the same claim, potentially gaining unintended access. A single-audience token limits the blast radius if the token is compromised or misused. We RECOMMEND using URIs to specify audiences. See Section 3 of [OAUTH-RESOURCEINDICATORS] for more details and security implications.

Some workload platforms provide credentials for interacting with their own APIs (e.g., Kubernetes). These credentials MUST NOT be used beyond the platform API. In the example of Kubernetes, a token used for anything other than the Kubernetes API itself MUST NOT carry the Kubernetes server in the aud claim. Reusing a platform API token for federation or resource access outside the platform conflates trust boundaries: the token’s audience includes the platform, so any relying party that accepts it could impersonate the workload back to the platform.

5.8. Multi-Tenancy Considerations

In multi-tenant platforms, relying parties MUST carefully evaluate which attributes are considered trustworthy when making authorization decisions. Access or federation MUST NOT be granted based solely on untrusted or easily forgeable attributes. In particular, the issuer claim in such environments may not uniquely identify a trusted authority, since each tenant could be configured with the same issuer identifier.

Relying parties SHOULD ensure that attributes used for authorization are bound to a trust domain under their control or validated by an entity with a clearly defined trust boundary. Failing to do so may allow a malicious tenant to obtain credentials that are indistinguishable from those of a legitimate tenant, leading to cross-tenant privilege escalation or unauthorized access to shared resources.

6. IANA Considerations

This document does not require actions by IANA.

7. Acknowledgements

The authors and contributors would like to thank the following people for their feedback and contributions to this document (in no particular order): Dag Sneeggen, Ned Smith, Dean H. Saxe, Yaron Sheffer, Andrii Deinega, Marcel Levy, Justin Richer, Pieter Kasselmann, Simon Canning, Evan Gilman, Joseph Salowey, Kathleen Moriarty and Flemming Andreasen.

8. References

8.1. Normative References

[JWK] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/rfc/rfc7517>.

[JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/rfc/rfc7519>.

[JWT-BCP] Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", BCP 225, RFC 8725, DOI 10.17487/RFC8725, February 2020, <https://www.rfc-editor.org/rfc/rfc8725>.

[OAUTH-ASSERTION] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <https://www.rfc-editor.org/rfc/rfc7521>.

[OAUTH-FRAMEWORK] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/rfc/rfc6749>.

[OAUTH-JWT] Jones, M. B., Campbell, B., Mortimore, C., and F. Skokan, "Updates to OAuth 2.0 JSON Web Token (JWT) Client Authentication and Assertion-Based Authorization Grants", Work in Progress, Internet-Draft, draft-ietf-oauth-rfc7523bis-07, 26 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-rfc7523bis-07>.

[OAUTH-RESOURCEINDICATORS]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, <https://www.rfc-editor.org/rfc/rfc8707>.

[OAUTH-TOKENEXCHANGE]
Jones, M., Nadalin, A., Campbell, B., Ed., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", RFC 8693, DOI 10.17487/RFC8693, January 2020, <https://www.rfc-editor.org/rfc/rfc8693>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

8.2. Informative References

[KubernetesServiceAccount]
"Kubernetes Service Account", May 2024, <https://kubernetes.io/docs/concepts/security/service-accounts/>.

[OIDC] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.

[OIDCDiscovery]
Sakimura, N., Bradley, J., Jones, M. and Jay, E., "OpenID Connect Discovery 1.0 incorporating errata set 2", December 2023, <https://openid.net/specs/openid-connect-discovery-1_0.html>.

[SPIFFE] "Secure Production Identity Framework for Everyone (SPIFFE)", May 2023, <https://github.com/spiffe/spiffe/blob/main/standards/SPIFFE.md>.

[TokenRequestV1]
"Kubernetes Token Request API V1", August 2024, <https://kubernetes.io/docs/reference/kubernetes-api/authentication-resources/token-request-v1/>.

[TokenReviewV1]
"Kubernetes Token Review API V1", August 2024, <https://kubernetes.io/docs/reference/kubernetes-api/authentication-resources/token-review-v1/>.

[WIMSE-ARCH]
Salowey, J. A., Rosomakho, Y., and H. Tschofenig, "Workload Identity in a Multi System Environment (WIMSE) Architecture", Work in Progress, Internet-Draft, draft-ietf-wimse-arch-07, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-wimse-arch-07>.

[WIMSE-HTTPSIG]
Salowey, J. A. and Y. Sheffer, "WIMSE Workload-to-Workload Authentication with HTTP Signatures", Work in Progress, Internet-Draft, draft-ietf-wimse-http-signature-03, 7 April 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-wimse-http-signature-03>.

Appendix A. Variations

A.1. Direct access to protected resources

Resource servers that protect resources may choose to trust multiple authorization servers, including the one that issues the platform identities. Instead of using the platform-issued identity to receive an access token of a different authorization domain, workloads can directly use the platform-issued identity to access a protected resource.

In this case, technically, the protected resource and workload are part of the same authorization domain.

A.2. Custom assertion flows

While [OAUTH-ASSERTION] and [OAUTH-JWT] are the proposed standards for this pattern, some authorization servers use [OAUTH-TOKENEXCHANGE] or a custom API for the issuance of an access token based on existing platform identity credentials. These patterns are not recommended and prevent interoperability.

Appendix B. Document History

[[To be removed from the final specification]]

-04

* Address review feedback from Kathleen Moriarty and Joe Salowey

* Expand introduction: explain the workload identity bootstrapping problem and the limitations of static credentials

* Expand SPIFFE section: trust bundles, JWT-SVID vs X509-SVID types, and Workload API identification

* Explicitly discuss obtaining multiple tokens with distinct audiences across all platform patterns

* Add "Application Interaction with Credential Sources" section covering SSRF and path traversal risks

* Update reference formatting

* Editorial improvements and updated acknowledgements

-03

* Add service-mesh section

* Add multi-tenancy considerations

* Add atomicity and flushing requirements to filesystem section

* Make it clear that invalidation is a matter of querying the status

* Rework local api section & security considerations

* Refer to RFC7517 in SPIFFE and add clarity on key distribution

* Editorial changes

-02

* Updated structure, bringing concrete examples back into the main text.

* Use more generic "federation" term instead of RFC 7523 specifics.

* Overall editorial improvements.

* Fix reference of Kubernetes Token Request API

* Prefer the term "document" over "specification".

* Update contributor and acknowledgements sections.

* Remove section about OIDC as it is too specific to a certain implementation.

* Rewrite abstract to better reflect the current content of the document.

-01

* Add credential delivery mechanisms

* Highlight relationship to other WIMSE work

* Add details about token typing and relation to OpenID Connect

* Add security considerations for audience

-00

* Rename draft with no content changes.

* Set Arndt to Editor role.

[as draft-wimse-workload-identity-bcp]

-02

* Move scope from Kubernetes to generic workload identity platform

* Add various patterns to appendix

  - Kubernetes

  - Cloud providers

  - SPIFFE

  - CI/CD

* Add some security considerations

* Update title

-01

* Editorial updates

-00

* Adopted by the WIMSE WG

Contributors

Benedikt Hofmann
Siemens
Email: hofmann.benedikt@siemens.com

Hannes Tschofenig
Siemens
Email: hannes.tschofenig@gmx.net

Edoardo Giordano
Nokia
Email: edoardo.giordano@nokia.com

Authors' Addresses

Arndt Schwenkschuster
SPIRL
Email: arndts.ietf@gmail.com

Yaroslav Rosomakho
Zscaler
Email: yrosomakho@zscaler.com