

Workgroup: Network Working Group
Internet-Draft: draft-private-access-tokens-01
Published: 25 October 2021
Intended Status: Standards Track
Expires: 28 April 2022
Authors: S. Hendrickson J. Iyengar T. Pauly
 Google LLC Fastly Apple Inc.
 S. Valdez C.A. Wood
 Google LLC Cloudflare

Private Access Tokens

Abstract

This document defines a protocol for issuing and redeeming privacy-preserving access tokens. These tokens can adhere to an issuance policy, allowing a service to limit access according to the policy without tracking client identity.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at <https://github.com/tfpaully/privacy-proxy>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 28 April 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

(<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. [Introduction](#)
 - 1.1. [Motivation](#)
 - 1.1.1. [Rate-limited Access](#)
 - 1.1.2. [Client Geo-Location](#)
 - 1.1.3. [Private Client Authentication](#)
 - 1.2. [Architecture](#)
 - 1.3. [Properties and Requirements](#)
 - 1.4. [Client Identity](#)
 - 1.5. [User Interaction](#)
2. [Notation and Terminology](#)
3. [Configuration](#)
4. [Token Challenge and Redemption Protocol](#)
 - 4.1. [Token Challenge](#)
 - 4.2. [Token Redemption](#)
5. [Issuance Protocol](#)
 - 5.1. [State Requirements](#)
 - 5.1.1. [Client State](#)
 - 5.1.2. [Mediator State](#)
 - 5.1.3. [Issuer State](#)
 - 5.2. [Issuance HTTP Headers](#)
 - 5.3. [Client-to-Mediator Request](#)
 - 5.4. [Mediator-to-Issuer Request](#)
 - 5.5. [Issuer-to-Mediator Response](#)
 - 5.6. [Mediator-to-Client Response](#)
 - 5.7. [Encrypting Origin Names](#)
 - 5.8. [Non-Interactive Schnorr Proof of Knowledge](#)
6. [Instantiating Uses Cases](#)
 - 6.1. [Rate-limited Access](#)
 - 6.2. [Client Geo-Location](#)
 - 6.3. [Private Client Authentication](#)
7. [Security Considerations](#)
 - 7.1. [Client Identity](#)
 - 7.2. [Denial of Service](#)
 - 7.3. [Channel Security](#)
8. [Privacy Considerations](#)
 - 8.1. [Client Token State and Origin Tracking](#)
 - 8.2. [Origin Verification](#)
 - 8.3. [Client Identification with Unique Keys](#)
 - 8.4. [Collusion Among Different Entities](#)

- [9. Deployment Considerations](#)
 - [9.1. Origin Key Rollout](#)
- [10. IANA Considerations](#)
 - [10.1. Authentication Scheme](#)
 - [10.2. HTTP Headers](#)
 - [10.3. Media Types](#)
 - [10.3.1. "message/access-token-request" media type](#)
 - [10.3.2. "message/access-token-response" media type](#)
- [11. References](#)
 - [11.1. Normative References](#)
 - [11.2. Informative References](#)
- [Appendix A. Related Work: Privacy Pass](#)
- [Authors' Addresses](#)

1. Introduction

Servers commonly use passive and persistent identifiers associated with clients, such as IP addresses or device identifiers, for enforcing access and usage policies. For example, a server might limit the amount of content an IP address can access over a given time period (referred to as a "metered paywall"), or a server might rate-limit access from an IP address to prevent fraud and abuse. Servers also commonly use the client's IP address as a strong indicator of the client's geographic location to limit access to services or content to a specific geographic area (referred to as "geofencing").

However, passive and persistent client identifiers can be used by any entity that has access to it without the client's express consent. A server can use a client's IP address or its device identifier to track client activity. A client's IP address, and therefore its location, is visible to all entities on the path between the client and the server. These entities can trivially track a client, its location, and servers that the client visits.

A client that wishes to keep its IP address private can hide its IP address using a proxy service or a VPN. However, doing so severely limits the client's ability to access services and content, since servers might not be able to enforce their policies without a stable and unique client identifier.

This document describes an architecture for Private Access Tokens (PATs), using RSA Blind Signatures as defined in [BLINDSIG], as an explicit replacement for these passive client identifiers. These tokens are privately issued to clients upon request and then redeemed by servers in such a way that the issuance and redemption events for a given token are unlinkable.

At first glance, using PATs in lieu of passive identifiers for policy enforcement suggests that some entity needs to know both the client's identity and the server's policy, and such an entity would be trivially able to track a client and its activities. However, with appropriate mediation and separation between the parties involved in the issuance and the redemption protocols, it is possible to eliminate this information concentration without any functional regressions. This document describes such a protocol.

The relationship of this work to Privacy Pass ([\[I-D.ietf-privacypass-protocol\]](#)) is discussed in [Appendix A](#).

1.1. Motivation

This section describes classes of use cases where an origin would traditionally use a stable and unique client identifier for enforcing attribute-based policy. Hiding these identifiers from origins would therefore require an alternative for origins to continue enforcing their policies. Using the Privacy Address Token architecture for addressing these use cases is described in [Section 6](#).

1.1.1. Rate-limited Access

An origin provides rate-limited access to content to a client over a fixed period of time. The origin does not need to know the client's identity, but needs to know that a requesting client has not exceeded the maximum rate set by the origin.

One example of this use case is a metered paywall, where an origin limits the number of page requests to each unique user over a period of time before the user is required to pay for access. The origin typically resets this state periodically, say, once per month. For example, an origin may serve ten (major content) requests in a month before a paywall is enacted. Origins may want to differentiate quick refreshes from distinct accesses.

Another example of this use case is rate-limiting page accesses to a client to help prevent fraud. Operations that are sensitive to fraud, such as account creation on a website, often employ rate-limiting as a defense in depth strategy. Captchas or additional verification can be required by these pages when a client exceeds a set rate-limit.

Origins routinely use client IP addresses for this purpose.

1.1.2. Client Geo-Location

An origin provides access to or customizes content based on the geo-location of the client. The origin does not need to know the

client's identity, but needs to know the geo-location, with some level of accuracy, for providing service.

A specific example of this use case is "geo-fencing", where an origin restricts the available content it can serve based on the client's geographical region.

Origins almost exclusively use client IP addresses for this purpose.

1.1.3. Private Client Authentication

An origin provides access to content for clients that have been authorized by a delegated or known mediator. The origin does not need to know the client's identity.

A specific example of this use case is a federated service that authorizes users for access to specific sites, such as a federated news service or a federated video streaming service. The origin trusts the federator to authorize users and needs proof that the federator authorized a particular user, but it does not need the user's identity to provide access to content.

Origins could currently redirect clients to a federator for authentication, but origins could then track the client's federator user ID or the client's IP address across accesses.

1.2. Architecture

At a high level, the PAT architecture seeks to solve the following problem: in the absence of a stable Client identifier, an Origin needs to verify the identity of a connecting Client and enforce access policies for the incoming Client. To accomplish this, the PAT architecture employs four functional components:

1. Client: requests a PAT from an Issuer and presents it to a Origin for access to the Origin's service.
2. Mediator: authenticates a Client, using information such as its IP address, an account name, or a device identifier. Anonymizes a Client to an Issuer and relays information between an anonymized Client and an Issuer.
3. Issuer: issues PATs to an anonymized Client on behalf of an Origin. Anonymizes an Origin to a Mediator and enforces the Origin's policy.
4. Origin: directs a Client to an Issuer with a challenge and enables access to content or services to the Client upon verification of any PAT sent in response by the Client.

In the PAT architecture, these four components interact as follows.

An Origin designates a trusted Issuer to issue tokens for it. The Origin then redirects any incoming Clients to the Issuer for policy enforcement, expecting the Client to return with a proof from the Issuer that the Origin's policy has been enforced for this Client.

The Client employs a trusted Mediator through which it communicates with the Issuer for this proof. The Mediator performs three important functions:

- *authenticate and associate the Client with a stable identifier;
- *maintain issuance state for the Client and relay it to the Issuer; and
- *anonymize the Client and mediate communication between the Client and the Issuer.

When a Mediator-anonymized Client requests a token from an Issuer, the Issuer enforces the Origin's policies based on the received Client issuance state and Origin policy. Issuers know the Origin's policies and enforce them on behalf of the Origin. An example policy is: "Limit 10 accesses per Client". More examples and their use cases are discussed in [Section 6](#). The Issuer does not learn the Client's true identity.

Finally, the Origin provides access to content or services to a Client upon verifying a PAT presented by the Client. Verification of this token serves as proof that the Client meets the Origin's policies as enforced by the delegated Issuer with the help of a Mediator. The Origin can then provide any services or content gated behind these policies to the Client.

[Figure 1](#) shows the components of the PAT architecture described in this document. Protocol details follow in [Section 4](#) and [Section 5](#).

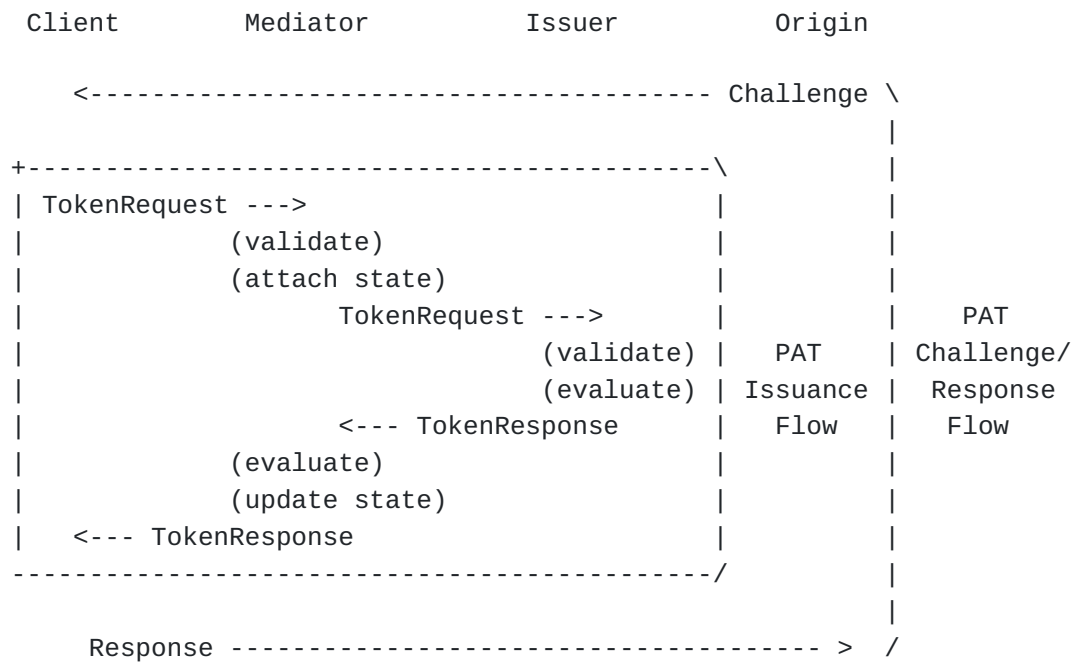


Figure 1: PAT Architectural Components

1.3. Properties and Requirements

In this architecture, the Mediator, Issuer, and Origin each have partial knowledge of the Client's identity and actions, and each entity only knows enough to serve its function (see [Section 2](#) for more about the pieces of information):

*The Mediator knows the Client's identity and learns the Client's public key (CLIENT_KEY), the Issuer being targeted (ISSUER_NAME), the period of time for which the Issuer's policy is valid (ISSUER_POLICY_WINDOW), and the number of tokens issued to a given Client for the claimed Origin in the given policy window. The Mediator does not know the identity of the Origin the Client is trying to access (ORIGIN_ID), but knows a Client-anonymized identifier for it (ANON_ORIGIN_ID).

*The Issuer knows the Origin's secret (ORIGIN_SECRET) and policy about client access, and learns the Origin's identity (ORIGIN_NAME) and the number of previous tokens issued to the Client (as communicated by the Mediator) during issuance. The Issuer does not learn the Client's identity.

*The Origin knows the Issuer to which it will delegate an incoming Client (ISSUER_NAME), and can verify that any tokens presented by the Client were signed by the Issuer. The Origin does not learn which Mediator was used by a Client for issuance.

Since an Issuer enforces policies on behalf of Origins, a Client is required to reveal the Origin's identity to the delegated Issuer. It

is a requirement of this architecture that the Mediator not learn the Origin's identity so that, despite knowing the Client's identity, a Mediator cannot track and concentrate information about Client activity.

An Issuer expects a Mediator to verify its Clients' identities correctly, but an Issuer cannot confirm a Mediator's efficacy or the Mediator-Client relationship directly without learning the Client's identity. Similarly, an Origin does not know the Mediator's identity, but ultimately relies on the Mediator to correctly verify or authenticate a Client for the Origin's policies to be correctly enforced. An Issuer therefore chooses to issue tokens to only known and reputable Mediators; the Issuer can employ its own methods to determine the reputation of a Mediator.

A Mediator is expected to employ a stable Client identifier, such as an IP address, a device identifier, or an account at the Mediator, that can serve as a reasonable proxy for a user with some creation and maintenance cost on the user.

For the Issuance protocol, a Client is expected to create and maintain stable and explicit secrets for time periods that are on the scale of Issuer policy windows. Changing these secrets arbitrarily during a policy window can result in token issuance failure for the rest of the policy window; see [Section 5.1.1](#) for more details. A Client can use a service offered by its Mediator or a third-party to store these secrets, but it is a requirement of the PAT architecture that the Mediator not be able to learn these secrets.

The privacy guarantees of the PAT architecture, specifically those around separating the identity of the Client from the names of the Origins that it accesses, are based on the expectation that there is not collusion between the entities that know about Client identity and those that know about Origin identity. Clients choose and share information with Mediators, and Origins choose and share policy with Issuers; however, the Mediator is generally expected to not be colluding with Issuers or Origins. If this occurs, it can become possible for a Mediator to learn or infer which Origins a Client is accessing, or for an Origin to learn or infer the Client identity. For further discussion, see [Section 8.4](#).

1.4. Client Identity

The PAT architecture does not enforce strong constraints around the definition of a Client identity and allows it to be defined entirely by a Mediator. If a user can create an arbitrary number of Client identities that are accepted by one or more Mediators, a malicious

user can easily abuse the system to defeat the Issuer's ability to enforce per-Client policies.

These multiple identities could be fake or true identities.

A Mediator alone is responsible for detecting and weeding out fake Client identities in the PAT architecture. An Issuer relies on a Mediator's reputation; as explained in [Section 1.3](#), the correctness of the architecture hinges on Issuers issuing tokens to only known and reputable Mediators.

Users have multiple true identities on the Internet however, and as a result, it seems possible for a user to abuse the system without having to create fake identities. For instance, a user could use multiple Mediators, authenticating with each one using a different true identity.

The PAT architecture offers no panacea against this potential abuse. We note however that the usages of PATs will cause the ecosystem to evolve and offer practical mitigations, such as:

- *An Issuer can learn the properties of a Mediator - specifically, which stable Client identifier is authenticated by the Mediator - to determine whether the Mediator is acceptable for an Origin.
- *An Origin can choose an Issuer based on the types of Mediators accepted by the Issuer, or the Origin can communicate its constraints to the designated Issuer.
- *An Origin can direct a user to a specific Issuer based on client properties that are visible. For instance, properties that are observable in the HTTP User Agent string.
- *The number of true Mediator-authenticated identities for a user is expected to be small, and therefore likely to be small enough to not matter for certain use cases. For instance, when PATs are used to prevent fraud by rate-limiting Clients (as described in [Section 1.1.1](#)), an Origin might be tolerant of the potential amplification caused by an attacking user's access to multiple true identities with Issuer-trusted Mediators.

1.5. User Interaction

When used in contexts like websites, origin servers that challenge clients for Private Access Tokens need to consider how to optimize their interaction model to ensure a good user experience.

Private Access Tokens are designed to be used without explicit user involvement. Since tokens are only valid for a single origin and in response to a specific challenge, there is no need for a user to

manage a limited pool of tokens across origins. The information that is available to an origin upon token redemption is limited to the fact that this is a client that passed a Mediator's checks and has not exceeded the per-origin limit defined by an Issuer. Generally, if a user is willing to use Private Access Tokens with a particular origin (or all origins), there is no need for per-challenge user interaction. Note that the Issuance flow may separately involve user interaction if the Mediator needs to authenticate the Client.

Since tokens are issued using a separate connection through a Mediator to an Issuer, the process of issuance can add user-perceivable latency. Origins SHOULD NOT block useful work on token authentication. Instead, token authentication can be used in similar ways to CAPTCHA validation today, but without the need for user interaction. If issuance is taking a long time, a website could show an indicator that it is waiting, or fall back to another method of user validation.

If an origin is requesting an unexpected number of tokens, such as requesting token authentication more than once for a single website load, it can indicate that the server is not functioning correctly, or is trying to attack or overload the client or issuance servers. In such cases, the client SHOULD ignore redundant token challengers, or else alert the user.

2. Notation and Terminology

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.

Unless said otherwise, this document encodes protocol messages in TLS notation from [[TLS13](#)], Section 3.

This draft includes pseudocode that uses the functions and conventions defined in [[HPKE](#)].

Encoding an integer to a sequence of bytes in network byte order is described using the function "encode(n, v)", where "n" is the number of bytes and "v" is the integer value. The function "len()" returns the length of a sequence of bytes.

The following terms are defined to refer to the different pieces of information passed through the system:

ISSUER_NAME: The Issuer Name identifies which Issuer is able to provide tokens for a Client. The Client sends the Issuer Name to

the Mediator so the Mediator know where to forward requests. Each Issuer is associated with a specific `ISSUER_POLICY_WINDOW`.

ISSUER_POLICY_WINDOW: The period over which an Issuer will track access policy, defined in terms of seconds and represented as a `uint64`. The state that the Mediator keeps for a Client is specific to a policy window. The effective policy window for a specific Client starts when the Client first sends a request associated with an Issuer.

ORIGIN_TOKEN_KEY: The public key used when generating and verifying Private Access Tokens. Each Origin Token Key is unique to a single Origin. The corresponding private key is held by the Issuer.

ISSUER_KEY: The public key used to encrypt values such as `ORIGIN_NAME` in requests from Clients to the Issuer, so that Mediators cannot learn the `ORIGIN_NAME` value. Each `ISSUER_KEY` is used across all requests on the Issuer, for different Origins.

ORIGIN_NAME: The name of the Origin that requests and verifies Private Access Tokens.

ANON_ORIGIN_ID: An identifier that is generated by the Client and marked on requests to the Mediator, which represents a specific Origin anonymously. The Client generates a stable `ANON_ORIGIN_ID` for each `ORIGIN_NAME`, to allow the Mediator to count token access without learning the `ORIGIN_NAME`.

CLIENT_KEY: A public key chosen by the Client and shared only with the Mediator.

CLIENT_SECRET: The secret key used by the Client during token issuance, whose public key (`CLIENT_KEY`) is shared with the Mediator.

ORIGIN_SECRET: The secret key used by the Issuer during token issuance, whose public key is not shared with anyone.

ANON_ISSUER_ORIGIN_ID: An identifier that is generated by Issuer based on an `ORIGIN_SECRET` that is per-Client and per-Origin. See [Section 5.6](#) for details of derivation.

3. Configuration

Issuers MUST provide three parameters for configuration:

1. `ISSUER_KEY`: a `KeyConfig` as defined in [\[OHTTP\]](#) to use when encrypting the `ORIGIN_NAME` in issuance requests. This parameter uses resource media type "application/ohttp-keys".

2. ISSUER_POLICY_WINDOW: a uint64 of seconds as defined in [Section 2](#).
3. ISSUER_REQUEST_URI: a Private Access Token request URL for generating access tokens. For example, an Issuer URL might be `https://issuer.example.net/access-token-request`. This parameter uses resource media type "text/plain".

These parameters can be obtained from an Issuer via a directory object, which is a JSON object whose field names and values are raw values and URLs for the parameters.

Field Name	Value
issuer-key	ISSUER_KEY resource URL as a JSON string
issuer-policy-window	ISSUER_POLICY_WINDOW as a JSON number
issuer-request-uri	ISSUER_REQUEST_URI resource URL as a JSON string

Table 1

As an example, the Issuer's JSON directory could look like:

```
{
  "issuer-key": "https://issuer.example.net/key",
  "issuer-token-window": 86400,
  "issuer-request-uri": "https://issuer.example.net/access-token-reque
}
```

Mediators MUST provide a single parameter for configuration, `MEDIATOR_REQUEST_URI`, which is Private Access Token request URL for proxying protocol messages to Issuers. For example, a Mediator URL might be `https://mediator.example.net/relay-access-token-request`. Similar to Issuers, Mediators make this parameter available by a directory object with the following contents:

Field Name	Value
mediator-request-uri	MEDIATOR_REQUEST_URI resource URL

Table 2

As an example, the Mediator's JSON dictionary could look like:

```
{
  "mediator-request-uri": "https://mediator.example.net/relay-access-t
}
```

Issuer and Mediator directory resources have the media type "application/json" and are located at the well-known location `/.well-known/private-access-tokens-directory`.

4. Token Challenge and Redemption Protocol

This section describes the interactive protocol for the token challenge and redemption flow between a Client and an Origin.

Token redemption is performed using HTTP Authentication ([\[RFC7235\]](#)), with the scheme "PrivateAccessToken". Origins challenge Clients to present a unique, single-use token from a specific Issuer. Once a Client has received a token from that Issuer, it presents the token to the Origin.

Token redemption only requires Origins to verify token signatures computed using the Blind Signature protocol from [\[BLINDSIG\]](#). Origins are not required to implement the complete Blind Signature protocol. (In contrast, token issuance requires Clients and Issuers to implement the Blind Signature protocol, as described in [Section 5](#).)

4.1. Token Challenge

Origins send a token challenge to Clients in an "WWW-Authenticate" header with the "PrivateAccessToken" scheme. This challenge includes a TokenChallenge message, along with information about what keys to use when requesting a token from the Issuer.

The TokenChallenge message has the following structure:

```
struct {  
    uint8_t version;  
    opaque origin_name<1..2^16-1>;  
    opaque issuer_name<1..2^16-1>;  
    opaque redemption_nonce[32];  
} TokenChallenge;
```

The structure fields are defined as follows:

*"version" is a 1-octet integer. This document defines version 1.

*"origin_name" is a string containing the name of the Origin (ORIGIN_NAME).

*"issuer_name" is a string containing the name of the Issuer (ISSUER_NAME).

*"redemption_nonce" is a fresh 32-byte nonce generated for each redemption request.

When used in an authentication challenge, the "PrivateAccessToken" scheme uses the following attributes:

- *"challenge", which contains a base64url-encoded [[RFC4648](#)] TokenChallenge value. This MUST be unique for every 401 HTTP response to prevent replay attacks.
- *"token-key", which contains a base64url encoding of the SubjectPublicKeyInfo object for use with the RSA Blind Signature protocol (ORIGIN_TOKEN_KEY).
- *"issuer-key", which contains a base64url encoding of a KeyConfig as defined in [[OHTTP](#)] to use when encrypting the ORIGIN_NAME in issuance requests (ISSUER_KEY).
- *"max-age", an optional attribute that consists of the number of seconds for which the challenge will be accepted by the Origin.

Origins MAY also include the standard "realm" attribute, if desired.

As an example, the WWW-Authenticate header could look like this:

```
WWW-Authenticate: PrivateAccessToken challenge=abc..., token-key=123...,  
issuer-key=456...
```

Upon receipt of this challenge, a Client uses the message and keys in the Issuance protocol (see [Section 5](#)). If the TokenChallenge has a version field the Client does not recognize or support, it MUST NOT parse or respond to the challenge. This document defines version 1, which indicates use of private tokens based on RSA Blind Signatures [[BLINDSIG](#)], and determines the rest of the structure contents.

Note that it is possible for the WWW-Authenticate header to include multiple challenges, in order to allow the Client to fetch a batch of multiple tokens for future use.

For example, the WWW-Authenticate header could look like this:

```
WWW-Authenticate: PrivateAccessToken challenge=abc..., token-key=123...,  
issuer-key=456..., PrivateAccessToken challenge=def..., token-key=234...,  
issuer-key=567...
```

4.2. Token Redemption

The output of the issuance protocol is a token that corresponds to the Origin's challenge (see [Section 4.1](#)). A token is a structure that begins with a single byte that indicates a version, which MUST match the version in the TokenChallenge structure.

```

struct {
    uint8_t version;
    uint8_t token_key_id[32];
    uint8_t message[32];
    uint8_t signature[Nk];
} Token;

```

The structure fields are defined as follows:

*"version" is a 1-octet integer. This document defines version 1.

*"token_key_id" is a collision-resistant hash that identifies the ORIGIN_TOKEN_KEY used to produce the signature. This is generated as SHA256(public_key), where public_key is a DER-encoded SubjectPublicKeyInfo object carrying the public key.

*"message" is a 32-octet message containing the hash of the original TokenChallenge, SHA256(TokenChallenge). This message is signed by the signature,

*"signature" is a Nk-octet RSA Blind Signature that covers the message. For version 1, Nk is indicated by size of the Token structure and may be 256, 384, or 512. These correspond to RSA 2048, 3072, and 4096 bit keys. Clients implementing version 1 MUST support signature sizes with Nk of 512 and 256.

When used for client authorization, the "PrivateAccessToken" authentication scheme defines one parameter, "token", which contains the base64url-encoded Token struct. All unknown or unsupported parameters to "PrivateAccessToken" authentication credentials MUST be ignored.

Clients present this Token structure to Origins in a new HTTP request using the Authorization header as follows:

Authorization: PrivateAccessToken token=abc...

Origins verify the token signature using the corresponding policy verification key from the Issuer, and validate that the message matches the hash of a TokenChallenge it previously issued and is still valid, SHA256(TokenChallenge), and that the version of the Token matches the version in the TokenChallenge. The TokenChallenge MAY be bound to a specific HTTP session with Client, but Origins can also accept tokens for valid challenges in new sessions.

If a Client's issuance request fails with a 401 error, as described in [Section 5.4](#), the Client MUST react to the challenge as if it could not produce a valid Authorization response.

5. Issuance Protocol

This section describes the Issuance protocol for a Client to request and receive a token from an Issuer. Token issuance involves a Client, Mediator, and Issuer, with the following steps:

1. The Client sends a token request to the Mediator, encrypted using an Issuer-specific key
2. The Mediator validates the request and proxies the request to the Issuer
3. The Issuer decrypts the request and sends a response back to the Mediator
4. The Mediator verifies the response and proxies the response to the Client

The Issuance protocol has a number of underlying cryptographic dependencies for operation:

*[\[HPKE\]](#), for encrypting information in transit between Client and Issuer across the Mediator.

*RSA Blind Signatures [\[BLINDSIG\]](#), for issuing and constructing Tokens as described in [Section 4.2](#).

*Prime Order Groups (POGs), for computing stable mappings between (Client, Origin) pairs. This document uses notation described in [\[VOPRF\]](#), [Section 2.1](#), and, in particular, the functions `RandomScalar()`, `Generator()`, `SerializeScalar()`, `SerializeElement()`, and `HashToScalar()`.

*Non-Interactive proof-of-knowledge (POK), as described in [Section 5.8](#), for verifying correctness of Client requests.

Clients and Issuers are required to implement all of these dependencies, whereas Mediators are required to implement POG and POK support.

5.1. State Requirements

The Issuance protocol requires each participating endpoint to maintain some necessary state, as described in this section.

5.1.1. Client State

A Client is required to have the following information, derived from a given TokenChallenge:

- *Origin name (ORIGIN_NAME), a URI referring to the Origin [[RFC6454](#)]. This is the value of TokenChallenge.origin_name.
- *Origin token public key (ORIGIN_TOKEN_KEY), a blind signature public key corresponding to the Origin identified by TokenChallenge.origin_name.
- *Issuer public key (ISSUER_KEY), a public key used to encrypt requests corresponding to the Issuer identified by TokenChallenge.issuer_name.

Clients maintain a stable CLIENT_ID that they use for all communication with a specific Mediator. CLIENT_ID is a public key, where the corresponding private key CLIENT_SECRET is known only to the client.

If the client loses this (CLIENT_ID, CLIENT_SECRET), they may generate a new tuple. The mediator will enforce if a client is allowed to use this new CLIENT_ID. See #mediator-state for details on this enforcement.

Clients also need to be able to generate an ANON_ORIGIN_ID value that corresponds to the ORIGIN_NAME, to send in requests to the Mediator.

ANON_ORIGIN_ID MUST be a stable and unpredictable 32-byte value computed by the Client. Clients MUST NOT change this value across token requests for the same ORIGIN_NAME. Doing so will result in token issuance failure (specifically, when a Mediator rejects a request upon detecting two ANON_ORIGIN_ID values that map to the same Origin).

One possible mechanism for implementing this identifier is for the Client to store a mapping between the ORIGIN_NAME and a randomly generated ANON_ORIGIN_ID for future requests. Alternatively, the Client can compute a PRF keyed by a per-client secret (CLIENT_SECRET) over the ORIGIN_NAME, e.g., $ANON_ORIGIN_ID = HKDF(secret=CLIENT_SECRET, salt="", info=ORIGIN_NAME)$.

5.1.2. Mediator State

A Mediator is required to maintain state for every authenticated Client. The mechanism of identifying a Client is specific to each Mediator, and is not defined in this document. As examples, the

Mediator could use device-specific certificates or account authentication to identify a Client.

Mediators must enforce that Clients don't change their CLIENT_ID frequently, to ensure Clients can't regularly evade the per-client policy as seen by the issuer. Mediators MUST NOT allow Clients to change their CLIENT_ID more than once within a policy window, or in the subsequent policy window after a previous CLIENT_ID change. Alternative schemes where the mediator stores the encrypted (CLIENT_ID, CLIENT_SECRET) tuple on behalf of the client are possible but not described here.

Mediators are expected to know the ISSUER_POLICY_WINDOW for any ISSUER_NAME to which they allow access. This information can be retrieved using the URIs defined in [Section 3](#).

For each Client-Issuer pair, a Mediator maintains a policy window start and end time for each Issuer from which a Client requests a token.

For each tuple of (CLIENT_ID, ANON_ORIGIN_ID, policy window), the Mediator maintains the following state:

- *A counter of successful tokens issued
- *Whether or not a previous request was rejected by the Issuer
- *The last received ANON_ISSUER_ORIGIN_ID value for this ANON_ORIGIN_ID, if any

5.1.3. Issuer State

Issuers maintain a stable ORIGIN_SECRET that they use in calculating values returned to the Mediator for each origin. If this value changes, it will open up a possibility for Clients to request extra tokens for an Origin without being limited, within a policy window.

Issuers are expected to have the private key that corresponds to ISSUER_KEY, which allows them to decrypt the ORIGIN_NAME values in requests.

Issuers also need to know the set of valid ORIGIN_TOKEN_KEY public keys and corresponding private key, for each ORIGIN_NAME that is served by the Issuer. Origins SHOULD update their view of the ORIGIN_TOKEN_KEY regularly to ensure that Client requests do not fail after ORIGIN_TOKEN_KEY rotation.

5.2. Issuance HTTP Headers

The Issuance protocol defines four new HTTP headers that are used in requests and responses between Clients, Mediators, and Issuers (see [Section 10.2](#)).

The "Sec-Token-Origin" is an Item Structured Header [[RFC8941](#)]. Its value MUST be a Byte Sequence. This header is sent both on Client-to-Mediator requests ([Section 5.3](#)) and on Issuer-to-Mediator responses ([Section 5.5](#)). Its ABNF is:

```
Sec-Token-Origin = sf-binary
```

The "Sec-Token-Client" is an Item Structured Header [[RFC8941](#)]. Its value MUST be a Byte Sequence. This header is sent on Client-to-Mediator requests ([Section 5.3](#)), and contains the bytes of CLIENT_KEY. Its ABNF is:

```
Sec-Token-Client = sf-binary
```

The "Sec-Token-Nonce" is an Item Structured Header [[RFC8941](#)]. Its value MUST be a Byte Sequence. This header is sent on Client-to-Mediator requests ([Section 5.3](#)), and contains a per-request nonce value. Its ABNF is:

```
Sec-Token-Nonce = sf-binary
```

The "Sec-Token-Count" is an Item Structured Header [[RFC8941](#)]. Its value MUST be an Integer. This header is sent on Mediator-to-Issuer requests ([Section 5.3](#)), and contains the number of times a Client has previously received a token for an Origin. Its ABNF is:

```
Sec-Token-Count = sf-integer
```

5.3. Client-to-Mediator Request

The Client and Mediator MUST use a secure and Mediator-authenticated HTTPS connection. They MAY use mutual authentication or mechanisms such as TLS certificate pinning, to mitigate the risk of channel compromise; see [Section 7](#) for additional about this channel.

Issuance begins by Clients hashing the TokenChallenge to produce a token input as `message = SHA256(challenge)`, and then blinding message as follows:

```
blinded_req, blind_inv = rsabssa_blind(ORIGIN_TOKEN_KEY, message)
```

The Client MUST use a randomized variant of RSABSSA in producing this signature with a salt length of at least 32 bytes.

The Client uses CLIENT_SECRET to generate proof of its request.

```
blind = RandomScalar()
blind_key = blind * CLIENT_SECRET
blind_generator = blind * Generator()
key_proof = SchnorrProof(CLIENT_SECRET, blind_key, blind_generator)
```

The Client then transforms this proof into "mapping_nonce", "mapping_key", "mapping_generator", and "mapping_proof".

```
mapping_nonce = SerializeScalar(blind)
mapping_key = SerializeElement(blind_key)
mapping_generator = SerializeElement(blind_generator)
mapping_proof = SerializeProof(key_proof)
```

The Client then constructs a Private Access Token request using mapping_key, mapping_generator, mapping_proof, blinded_req, and origin information.

```
struct {
    uint8_t version;
    uint8_t mapping_generator[Ne];
    uint8_t mapping_key[Ne];
    uint8_t mapping_proof[Np];
    uint8_t token_key_id;
    uint8_t blinded_req[Nk];
    uint8_t name_key_id[32];
    uint8_t encrypted_origin_name<1..2^16-1>;
} AccessTokenRequest;
```

The structure fields are defined as follows:

*"version" is a 1-octet integer, which matches the version in the TokenChallenge. This document defines version 1.

*"mapping_generator", "mapping_key", and "mapping_proof" are computed as described above.

*"token_key_id" is the least significant byte of the ORIGIN_TOKEN_KEY key ID, which is generated as SHA256(public_key), where public_key is a DER-encoded SubjectPublicKeyInfo object carrying ORIGIN_TOKEN_KEY.

*"blinded_req" is the Nk-octet request defined above.

*"name_key_id" is a collision-resistant hash that identifies the ISSUER_KEY public key, generated as SHA256(KeyConfig).

*"encrypted_origin_name" is an encrypted structure that contains ORIGIN_NAME, calculated as described in [Section 5.7](#).

The Client then generates an HTTP POST request to send through the Mediator to the Issuer, with the AccessTokenRequest as the body. The media type for this request is "message/access-token-request". The Client includes the "Sec-Token-Origin" header, whose value is ANON_ORIGIN_ID; the "Sec-Token-Client" header, whose value is CLIENT_KEY; and the "Sec-Token-Nonce" header, whose value is mapping_nonce. The Client sends this request to the Mediator's proxy URI. An example request is shown below, where Nk = 512.

```
:method = POST
:scheme = https
:authority = issuer.net
:path = /access-token-request
accept = message/access-token-response
cache-control = no-cache, no-store
content-type = message/access-token-request
content-length = 512
sec-token-origin = ANON_ORIGIN_ID
sec-token-client = CLIENT_KEY
sec-token-nonce = mapping_nonce
```

<Bytes containing the AccessTokenRequest>

If the Mediator detects a version in the AccessTokenRequest that it does not recognize or support, it MUST reject the request with an HTTP 400 error.

The Mediator also checks to validate that the name_key_id in the client's AccessTokenRequest matches a known ISSUER_KEY public key for the Issuer. For example, the Mediator can fetch this key using the API defined in [Section 3](#). This check is done to help ensure that the Client has not been given a unique key that could allow the Issuer to fingerprint or target the Client. If the key does not match, the Mediator rejects the request with an HTTP 400 error. Note that Mediators need to be careful in cases of key rotation; see [Section 8](#).

The Mediator finally checks to ensure that the AccessTokenRequest.mapping_proof is valid for the given CLIENT_KEY; see [Section 5.8](#) for verification details. If the index is invalid, the Mediator rejects the request with an HTTP 400 error.

If the Mediator accepts the request, it will look up the state stored for this Client. It will look up the count of previously generate tokens for this Client using the same ANON_ORIGIN_ID. See [Section 5.1.2](#) for more details.

If the Mediator has stored state that a previous request for this ANON_ORIGIN_ID was rejected by the Issuer in the current policy

window, it SHOULD reject the request without forwarding it to the Issuer.

If the Mediator detects this Client has changed their CLIENT_ID more frequently than allowed as described in #mediator-state, it SHOULD reject the request without forwarding it to the Issuer.

5.4. Mediator-to-Issuer Request

The Mediator and the Issuer MUST use a secure and Issuer-authenticated HTTPS connection. Also, Issuers MUST authenticate Mediators, either via mutual TLS or another form of application-layer authentication. They MAY additionally use mechanisms such as TLS certificate pinning, to mitigate the risk of channel compromise; see [Section 7](#) for additional about this channel.

Before copying and forwarding the Client's AccessTokenRequest request to the Issuer, the Mediator adds a header that includes the count of previous tokens as "Sec-Token-Count". The Mediator MAY also add additional context information, but MUST NOT add information that will uniquely identify a Client.

```
:method = POST
:scheme = https
:authority = issuer.net
:path = /access-token-request
accept = message/access-token-response
cache-control = no-cache, no-store
content-type = message/access-token-request
content-length = 512
sec-token-count = 3
```

<Bytes containing the AccessTokenRequest>

Upon receipt of the forwarded request, the Issuer validates the following conditions:

- *The "Sec-Token-Count" header is present
- *The AccessTokenRequest contains a supported version
- *For version 1, the AccessTokenRequest.name_key_id corresponds to the ID of the ISSUER_KEY held by the Issuer
- *For version 1, the AccessTokenRequest.encrypted_origin_name can be decrypted using the Issuer's private key (the private key associated with ISSUER_KEY), and matches an ORIGIN_NAME that is served by the Issuer

*For version 1, the `AccessTokenRequest.blinded_req` is of the correct size

*For version 1, the `AccessTokenRequest.token_key_id` corresponds to an ID of an `ORIGIN_TOKEN_KEY` for the corresponding `ORIGIN_NAME`

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the Mediator, which will forward the error to the client.

If the request is valid, the Issuer then can use the value from "Sec-Token-Count" to determine if the Client is allowed to receive a token for this Origin during the current policy window. If the Issuer refuses to issue more tokens, it responds with an HTTP 429 (Too Many Requests) error to the Mediator, which will forward the error to the client.

The Issuer determines the correct `ORIGIN_TOKEN_KEY` by using the decrypted `ORIGIN_NAME` value and `AccessTokenRequest.token_key_id`. If there is no `ORIGIN_TOKEN_KEY` whose truncated key ID matches `AccessTokenRequest.token_key_id`, the Issuer MUST return an HTTP 401 error to Mediator, which will forward the error to the client. The Mediator learns that the client's view of the Origin key was invalid in the process.

5.5. Issuer-to-Mediator Response

If the Issuer is willing to give a token to the Client, the Issuer verifies the token request using "mapping_generator", "mapping_key", and "mapping_proof":

```
valid = SchnorrVerify(mapping_generator, mapping_key, mapping_proof)
```

If this fails, the Issuer rejects the request with a 400 error. Otherwise, the Issuer decrypts `AccessTokenRequest.encrypted_origin_name` to discover "origin". If this fails, the Issuer rejects the request with a 400 error. The Issuer then evaluates the mapping over the `ORIGIN_SECRET` pertaining to the origin for this issuer:

```
mapping_input = DeserializeElement(AccessTokenRequest.mapping_key)
index = ORIGIN_SECRET * mapping_input
mapping_index = SerializeElement(index)
```

If `DeserializeElement` fails, or if `AccessTokenRequest.mapping_key` is the identity element, the Issuer rejects the request with a 400 error.

The Issuer completes the issuance flow by computing a blinded response as follows:

```
blind_sig = rsabssa_blind_sign(skP, AccessTokenRequest.blinded_req)
```

skP is the private key corresponding to ORIGIN_TOKEN_KEY, known only to the Issuer.

The Issuer generates an HTTP response with status code 200 whose body consists of blind_sig, with the content type set as "message/access-token-response" and the mapping_tag set in the "Sec-Token-Origin" header.

```
:status = 200
content-type = message/access-token-response
content-length = 512
sec-token-origin = mapping_index
```

<Bytes containing the blind_sig>

5.6. Mediator-to-Client Response

Upon receipt of a successful response from the Issuer, the Mediator extracts the "Sec-Token-Origin" header, and uses the value to determine ANON_ISSUER_ORIGIN_ID.

```
index = DeserializeElement(mapping_index)
nonce = DeserializeScalar(mapping_nonce)
ANON_ISSUER_ORIGIN_ID = (nonce^(-1)) * index
```

If the "Sec-Token-Origin" is missing, or if the same ANON_ISSUER_ORIGIN_ID was previously received in a response for a different ANON_ORIGIN_ID within the same policy window, the Mediator MUST drop the token and respond to the client with an HTTP 400 status. If there is not an error, the ANON_ISSUER_ORIGIN_ID is stored alongside the state for the ANON_ORIGIN_ID.

For all other cases, the Mediator forwards all HTTP responses unmodified to the Client as the response to the original request for this issuance.

When the Mediator detects successful token issuance, it MUST increment the counter in its state for the number of tokens issued to the Client for the ANON_ORIGIN_ID.

Upon receipt, the Client handles the response and, if successful, processes the body as follows:

```
sig = rsabssa_finalize(ORIGIN_TOKEN_KEY, nonce, blind_sig, blind_inv)
```

If this succeeds, the Client then constructs a Private Access Token as described in [Section 4.1](#) using the token input message and output sig.

5.7. Encrypting Origin Names

Given a KeyConfig (ISSUER_KEY), Clients produce encrypted_origin_name and authenticate all other contents of the AccessTokenRequest using the following values:

- *the key identifier from the configuration, keyID, with the corresponding KEM identified by kemID, the public key from the configuration, pkI, and;
- *a selected combination of KDF, identified by kdfID, and AEAD, identified by aeadID.

Beyond the key configuration inputs, Clients also require the AccessTokenRequest inputs. Together, these are used to encapsulate ORIGIN_NAME (origin_name) and produce ENCRYPTED_ORIGIN_NAME (encrypted_origin) as follows:

1. Compute an [HPKE](#) context using pkI, yielding context and encapsulation key enc.
2. Construct associated data, aad, by concatenating the values of keyID, kemID, kdfID, aeadID, and all other values of the AccessTokenRequest structure.
3. Encrypt (seal) request with aad as associated data using context, yielding ciphertext ct.
4. Concatenate the values of aad, enc, and ct, yielding an Encapsulated Request enc_request.

Note that enc is of fixed-length, so there is no ambiguity in parsing this structure.

In pseudocode, this procedure is as follows:

```
enc, context = SetupBaseS(pkI, "AccessTokenRequest")
aad = concat(encode(1, keyID),
             encode(2, kemID),
             encode(2, kdfID),
             encode(2, aeadID),
             encode(1, version),
             encode(Ne, mapping_generator),
             encode(Ne, mapping_key),
             encode(Np, mapping_proof),
             encode(1, token_key_id),
             encode(Nk, blinded_req),
             encode(32, name_key_id))
ct = context.Seal(aad, origin_name)
encrypted_origin_name = concat(enc, ct)
```

Issuers reverse this procedure to recover `ORIGIN_NAME` by computing the AAD as described above and decrypting `encrypted_origin_name` with their private key `skI`, the private key corresponding to `pkI`. In pseudocode, this procedure is as follows:

```
enc, ct = parse(encrypted_origin_name)
aad = concat(encode(1, keyID),
             encode(2, kemID),
             encode(2, kdfID),
             encode(2, aeadID),
             encode(1, version),
             encode(Ne, mapping_generator),
             encode(Ne, mapping_key),
             encode(Np, mapping_proof),
             encode(1, token_key_id),
             encode(Nk, blinded_req),
             encode(32, name_key_id))
enc, context = SetupBaseR(enc, skI, "AccessTokenRequest")
origin_name, error = context.Open(aad, ct)
```

5.8. Non-Interactive Schnorr Proof of Knowledge

Each Issuance request requires evaluation and verification of a Schnorr proof-of-knowledge. Given input secret "secret" and two elements, "base" and "target", generation of this proof (`u`, `c`, `z`), denoted `SchnorrProof(secret, base, target)`, works as follows:

```
r = RandomScalar()
u = r * base
c = HashToScalar(SerializeElement(base) ||
                 SerializeElement(target) ||
                 SerializeElement(mask),
                 dst = "PrivateAccessTokensProof")
z = r + (c * secret)
```

The proof is encoded by serializing (`u`, `c`, `z`) as follows:

```
struct {
    uint8_t u[Ne];
    uint8_t c[Ns];
    uint8_t z[Ns];
} Proof;
```

The size of this structure is $N_p = N_e + 2 \cdot N_s$ bytes.

Verification of a proof (`u`, `c`, `z`), denoted `SchnorrVerify(base, target, proof)`, works as follows:

```
c = HashToScalar(SerializeElement(base) ||
                  SerializeElement(target) ||
                  SerializeElement(mask),
                  dst = "PrivateAccessTokensProof")
expected_left = base * z
expected_right = u + (target * c)
```

The proof is considered valid if `expected_left` is the same as `expected_right`.

6. Instantiating Uses Cases

This section describes various instantiations of this protocol to address use cases described in [Section 1.1](#).

6.1. Rate-limited Access

To instantiate this case, the site acts as an Origin and registers a "bounded token" policy with the Issuer. In this policy, the Issuer enforces a fixed number of tokens that it will allow a Client to request for a single `ORIGIN_NAME`.

Origins request tokens from Clients and, upon successful redemption, the Origin knows the Client was able to request a token for the given `ORIGIN_NAME` within its budget. Failure to present a token can be interpreted as a signal that the client's token budget was exceeded.

Clients can redeem a token from a specific challenge up to the max-age in the challenge. Servers can choose to issue many challenges in a single HTTP 401 response, providing the client with many challenge nonces which can be used to redeem tokens over a longer period of time.

6.2. Client Geo-Location

To instantiate this use case, the Issuer has an issuing key pair per geographic region, i.e., each region has a unique policy key. When verifying the key for the Client request, the Mediator obtains the per-region key from the Issuer based on the Client's perceived location. During issuance, the Mediator checks that this key matches that of the Client's request. If it matches, the Mediator forwards the request to complete issuance. The number of key pairs is then the cross product of the number of Origins that require per-region keys and the number of regions.

During redemption, Clients present their geographic location to Origins in a "Sec-CH-Geohash" header. Origins use this to obtain the appropriate policy verification key. Origins request tokens from

Clients and, upon successful redemption, the Origin knows the Client obtained a token for the given ORIGIN_NAME in the specified region.

6.3. Private Client Authentication

To instantiate this case, the site acts as an Origin and registers an "unlimited token" policy with the Issuer. In this policy, the Issuer does not enforce any limit on the number of tokens a given user will obtain.

Origins request tokens from Clients and, upon successful redemption, the Origin knows the Client was able to request a token for the given ORIGIN_NAME tuple. As a result, the Origin knows this is an authentic client.

7. Security Considerations

This section discusses security considerations for the protocol.

[OPEN ISSUE: discuss trust model]

7.1. Client Identity

The HTTPS connection between Client and Mediator is minimally Mediator-authenticated. Mediators can also require Client authentication if they wish to restrict Private Access Token proxying to trusted or otherwise authenticated Clients. Absent some form of Client authentication, Mediators can use other per-Client information for the client identifier mapping, such as IP addresses.

7.2. Denial of Service

Requesting and verifying a Private Access Token is more expensive than checking an implicit signal, such as an IP address, especially since malicious clients can generate garbage Private Access Tokens and for Origins to work. However, similar DoS vectors already exist for Origins, e.g., at the underlying TLS layer.

7.3. Channel Security

An attacker that can act as an intermediate between Mediator and Issuer communication can influence or disrupt the ability for the Issuer to correctly rate-limit token issuance. All communication channels use server-authenticated HTTPS. Some connections, e.g., between a Mediator and an Issuer, require mutual authentication between both endpoints. Where appropriate, endpoints MAY use further enhancements such as TLS certificate pinning to mitigate the risk of channel compromise.

An attacker that can intermediate the channel between Client and Origin can observe a TokenChallenge, and can view a Token being presented for authentication to an Origin. Scoping the TokenChallenge nonce to the Client HTTP session prevents Tokens being collected in one session and then presented to the Origin in another. Note that an Origin cannot distinguish between a connection to a single Client and a connection to an attacker intermediating multiple Clients. Thus, it is possible for an attacker to collect and later present Tokens from multiple clients over the same Origin session.

8. Privacy Considerations

8.1. Client Token State and Origin Tracking

Origins SHOULD only generate token challenges based on client action, such as when a user loads a website. Clients SHOULD ignore token challenges if an Origin tries to force the client to present tokens multiple times without any new client-initiated action. Failure to do so can allow malicious origins to track clients across contexts. Specifically, an origin can abuse per-user token limits for tracking by assigning each new client a random token count and observing whether or not the client can successfully redeem that many tokens in a given context. If any token redemption fails, then the origin learns information about how many tokens that client had previously been issued.

By rejecting repeated or duplicative challenges within a single context, the origin only learns a single bit of information: whether or not the client had any token quota left in the given policy window.

8.2. Origin Verification

Private Access Tokens are defined in terms of a Client authenticating to an Origin, where the "origin" is used as defined in [\[RFC6454\]](#). In order to limit cross-origin correlation, Clients MUST verify that the origin_name presented in the TokenChallenge structure ([Section 4.1](#)) matches the origin that is providing the HTTP authentication challenge, where the matching logic is defined for same-origin policies in [\[RFC6454\]](#). Clients MAY further limit which authentication challenges they are willing to respond to, for example by only accepting challenges when the origin is a web site to which the user navigated.

8.3. Client Identification with Unique Keys

Client activity could be linked if an Origin and Issuer collude to have unique keys targeted at specific Clients or sets of Clients.

To mitigate the risk of a targeted ISSUER_KEY, the Mediator can observe and validate the name_key_id presented by the Client to the Issuer. As described in [Section 5](#), Mediators MUST validate that the name_key_id in the Client's AccessTokenRequest matches a known public key for the Issuer. The Mediator needs to support key rotation, but ought to disallow very rapid key changes, which could indicate that an Origin is colluding with an Issuer to try to rotate the key for each new Client in order to link the client activity.

To mitigate the risk of a targeted ORIGIN_TOKEN_KEY, the protocol expects that an Issuer has only a single valid public key for signing tokens at a time. The Client does not present the name_key_id of the token public key to the Issuer, but instead expects the Issuer to infer the correct key based on the information the Issuer knows, specifically the origin_name itself.

8.4. Collusion Among Different Entities

Collusion among the different entities in the PAT architecture can result in violation of the Client's privacy.

Issuers and Mediators should be run by mutually distinct organizations to limit information sharing. A single entity running an issuer and mediator for a single redemption can view the origins being accessed by a given client. Running the issuer and mediator in this 'single issuer/mediator' fashion reduces the privacy promises to those of the [\[I-D.ietf-privacypass-protocol\]](#); see [Appendix A](#) for more discussion. This may be desirable for a redemption flow that is limited to specific issuers and mediators, but should be avoided where hiding origins from the mediator is desirable.

If a Mediator and Origin are able to collude, they can correlate a client's identity and origin access patterns through timestamp correlation. The timing of a request to an Origin and subsequent token issuance to a Mediator can reveal the Client identity (as known to the Mediator) to the Origin, especially if repeated over multiple accesses.

9. Deployment Considerations

9.1. Origin Key Rollout

Issuers SHOULD generate a new (ORIGIN_TOKEN_KEY, ORIGIN_SECRET) regularly, and SHOULD maintain old and new secrets to allow for graceful updates. The RECOMMENDED rotation interval is two times the length of the policy window for that information. During generation, issuers must ensure the token_key_id (the 8-bit prefix of SHA256(ORIGIN_TOKEN_KEY)) is different from all other token_key_id values for that origin currently in rotation. One way to ensure this uniqueness is via rejection sampling, where a new key is generated

until its token_key_id is unique among all currently in rotation for the origin.

10. IANA Considerations

10.1. Authentication Scheme

This document registers the "PrivateAccessToken" authentication scheme in the "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry" established by [\[RFC7235\]](#).

Authentication Scheme Name: PrivateAccessToken

Pointer to specification text: [Section 4.1](#) of this document

10.2. HTTP Headers

This document registers four new headers for use on the token issuance path in the "Permanent Message Header Field Names" <<https://www.iana.org/assignments/message-headers>>.

Header Field Name	Protocol	Status	Reference
Sec-Token-Origin	http	std	This document
Sec-Token-Client	http	std	This document
Sec-Token-Nonce	http	std	This document
Sec-Token-Count	http	std	This document

Figure 2: Registered HTTP Header

10.3. Media Types

This specification defines the following protocol messages, along with their corresponding media types:

*AccessTokenRequest [Section 5](#): "message/access-token-request"

*AccessTokenResponse [Section 5](#): "message/access-token-response"

The definition for each media type is in the following subsections.

10.3.1. "message/access-token-request" media type

Type name: message

Subtype name:

access-token-request

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see [Section 5](#)

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see
Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

10.3.2. "message/access-token-response" media type

Type name: message

Subtype name: access-token-response

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations:

see [Section 5](#)

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see
Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

11. References

11.1. Normative References

- [BLINDSIG] Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-rsa-blind-signatures-02, 2 August 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-rsa-blind-signatures-02>>.
- [HPKE] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-12, 2 September 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-12>>.
- [OHTTP] Thomson, M. and C. A. Wood, "Oblivious HTTP", Work in Progress, Internet-Draft, draft-thomson-http-oblivious-02, 24 August 2021, <<https://>

datatracker.ietf.org/doc/html/draft-thomson-http-oblivious-02>.

- [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>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/rfc/rfc4648>>.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <<https://www.rfc-editor.org/rfc/rfc6454>>.
- [RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <<https://www.rfc-editor.org/rfc/rfc7235>>.
- [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>>.
- [RFC8941] Nottingham, M. and P-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <<https://www.rfc-editor.org/rfc/rfc8941>>.
- [TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [VOPRF] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-08, 25 October 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-08>>.

11.2. Informative References

- [I-D.ietf-privacypass-protocol] Celi, S., Davidson, A., and A. Faz-Hernandez, "Privacy Pass Protocol Specification", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-01, 22 February 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-01>>.

Appendix A. Related Work: Privacy Pass

Private Access Tokens has many similarities to the existing Privacy Pass protocol ([[I-D.ietf-privacypass-protocol](#)]). Both protocols allow clients to redeem signed tokens while not allowing linking between token issuance and token redemption.

There are several important differences between the protocols, however:

- *Private Access Tokens uses per-origin tokens that support rate-limiting policies. Each token can only be used with a specific origin in accordance with a policy defined for that origin. This allows origins to implement metered paywalls or mechanisms that limit the actions a single client can perform. Per-origin tokens also ensure that one origin cannot consume all of a client's tokens, so there is less need for clients to manage when they are willing to present tokens to origins.

- *Private Access Tokens employ an online challenge ([Section 4.1](#)) during token redemption. This ensures that tokens cannot be harvested and stored for use later. This also removes the need for preventing double spending or employing token expiry techniques, such as frequent signer rotation or expiry-encoded public metadata.

- *Private Access Tokens use a publically verifiable signature [[BLINDSIG](#)] to optimize token verification at the origin by avoiding a round trip to the issuer/mediator.

Authors' Addresses

Scott Hendrickson
Google LLC

Email: scott@shendrickson.com

Jana Iyengar
Fastly

Email: jri@fastly.com

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014,
United States of America

Email: tpauly@apple.com

Steven Valdez
Google LLC

Email: svaldez@chromium.org

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net