

Workgroup: Network Working Group
Internet-Draft:
draft-privacypass-rate-limit-tokens-02
Published: 2 May 2022
Intended Status: Informational
Expires: 3 November 2022
Authors: S. Hendrickson J. Iyengar T. Pauly
 Google LLC Fastly Apple Inc.
 S. Valdez C. A. Wood
 Google LLC Cloudflare

Rate-Limited Token Issuance Protocol

Abstract

This document specifies a variant of the Privacy Pass issuance protocol that allows for tokens to be rate-limited on a per-origin basis. This enables origins to use tokens for use cases that need to restrict access from anonymous clients.

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 3 November 2022.

Copyright Notice

Copyright (c) 2022 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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

- [1. Introduction](#)
 - [1.1. Motivation](#)
 - [1.2. Properties and Requirements](#)
- [2. Terminology](#)
- [3. Configuration](#)
- [4. Token Challenge Requirements](#)
- [5. Issuance Protocol](#)
 - [5.1. State Requirements](#)
 - [5.1.1. Client State](#)
 - [5.1.2. Attester State](#)
 - [5.1.3. Issuer State](#)
 - [5.2. Issuance HTTP Headers](#)
 - [5.3. Client-to-Attester Request](#)
 - [5.4. Attester-to-Issuer Request](#)
 - [5.5. Issuer-to-Attester Response](#)
 - [5.6. Attester-to-Client Response](#)
- [6. Encrypting Origin Token Requests and Responses](#)
 - [6.1. Client to Issuer Encapsulation](#)
 - [6.2. Issuer to Client Encapsulation](#)
- [7. Anonymous Issuer Origin ID Computation](#)
 - [7.1. Client Behavior](#)
 - [7.1.1. Request Key](#)
 - [7.1.2. Request Signature](#)
 - [7.2. Attester Behavior \(Client Request Validation\)](#)
 - [7.3. Issuer Behavior](#)
 - [7.4. Attester Behavior \(Index Computation\)](#)
- [8. Security Considerations](#)
 - [8.1. Channel Security](#)
 - [8.2. Token Request Unlinkability and Unforgeability](#)
 - [8.3. Information Disclosure](#)
- [9. Privacy Considerations](#)
 - [9.1. Client Token State and Origin Tracking](#)
 - [9.2. Origin Verification](#)
 - [9.3. Client Identification with Unique Keys](#)
 - [9.4. Origin Identification](#)
 - [9.5. Collusion Among Different Entities](#)

- [10. Deployment Considerations](#)
 - [10.1. Token Key Management](#)
- [11. IANA considerations](#)
 - [11.1. Token Type](#)
 - [11.1.1. ECDSA-based Token Type](#)
 - [11.1.2. Ed25519-based Token Type](#)
 - [11.2. HTTP Headers](#)
- [12. References](#)
 - [12.1. Normative References](#)
 - [12.2. Informative References](#)
- [Appendix A. Acknowledgements](#)
- [Appendix B. Test Vectors](#)
 - [B.1. Origin Name Encryption Test Vector](#)
 - [B.2. Anonymous Origin ID Test Vector](#)
- [Authors' Addresses](#)

1. Introduction

This document specifies a variant of the Privacy Pass issuance protocol (as defined in [\[ARCH\]](#)) that allows for tokens to be rate-limited on a per-origin basis. This enables origins to use tokens for use cases that need to restrict access from anonymous clients.

The base Privacy Pass issuance protocol [\[ISSUANCE\]](#) defines stateless anonymous tokens, which can either be publicly verifiable or not.

This variant build upon the publicly verifiable issuance protocol that uses RSA Blind Signatures [\[BLINDSIG\]](#), and allows tokens to be rate-limited on a per-origin basis. This means that a client will only be able to receive a limited number of tokens associated with a given origin server within a fixed period of time.

This issuance protocol registers the Rate-Limited Blind RSA token type ([Section 11.1](#)), to be used with the PrivateToken HTTP authentication scheme defined in [\[AUTHSCHEME\]](#).

1.1. Motivation

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.

Privacy Pass tokens in general allow clients to provide anonymous attestation of various properties. The tokens generated by the base issuance protocol ([\[ISSUANCE\]](#)) can be used to verify that a client meets a particular bar for attestation, but cannot be used by a redeeming server to rate-limit specific clients.

There are several use cases for rate-limiting anonymous clients that are common on the Internet. These routinely use client IP address tracking, among other characteristics, to implement rate-limiting.

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

Another example of this use case is a metered paywall, where an origin limits the number of page requests from 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.

For some applications, the basic issuance protocol from [[BASIC-ISSUANCE](#)] could be used to implement rate limits. In particular, the 'Joint Attester and Issuer' model from [[ARCH](#)] could be used to restrict the number of tokens issued to individual clients over a time window. However, in this deployment model, the Attester and Issuer would learn all origins used by a participating client. In some cases this might be a significant portion of browsing history. The issuance protocol defined in this document employs the 'Split Origin, Attester, Issuer' model to combat this, where the issuer would know all per-origin policies, and the attester would maintain per-client state without knowing all origins a client visits.

1.2. Properties and Requirements

For rate-limited token issuance, the Attester, Issuer, and Origin as defined in [[ARCH](#)] 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 Attester 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), the number of tokens the Issuer is willing to issue within the current policy window, and the number of tokens issued to a given Client for the claimed Origin in the policy window. The Attester does not know the identity of the Origin the Client is trying to access (Origin Name), but knows a Client-anonymized identifier for it (Anonymous Origin ID).

*The Issuer knows the Origin's secret (Issuer Origin Secret) and policy about client access, and learns the Origin's identity (Origin Name) during issuance. The Issuer does not learn the Client's identity or information about the Client's access pattern.

*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 Attester 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 protocol that the Attester not learn the Origin's identity so that, despite knowing the Client's identity, an Attester cannot track and concentrate information about Client activity.

An Issuer expects an Attester to verify its Clients' identities correctly, but an Issuer cannot confirm an Attester's efficacy or the Attester-Client relationship directly without learning the Client's identity. Similarly, an Origin does not know the Attester's identity, but ultimately relies on the Attester 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 Attesters; the Issuer can employ its own methods to determine the reputation of a Attester.

An Attester is expected to employ a stable Client identifier, such as an IP address, a device identifier, or an account at the Attester, 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 Attester or a third-party to store these secrets, but it is a requirement of this protocol that the Attester not be able to learn these secrets.

The privacy guarantees of this issuance protocol, 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 Attesters, and Origins choose and share policy with Issuers; however, the Attester is generally expected to not be

colluding with Issuers or Origins. If this occurs, it can become possible for an Attester 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 9.5](#).

2. 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 otherwise specified, 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 in [[ARCH](#)] and are used throughout this document:

- *Client: An entity that provides authorization tokens to services across the Internet, in return for authorization.
- *Issuer: An entity that produces Privacy Pass tokens to clients.
- *Attester: An entity that can attest to properties about the client, including previous patterns of access.
- *Origin: The server from which the client can redeem tokens.
- *Issuance Protocol: The protocol exchange that involves the client, attester, and issuer, used to generate tokens.

The following terms are defined in [[AUTHSCHEME](#)], which defines the interactions between clients and origins:

- *Issuer Name: The name that identifies the Issuer, which is an entity that can generate tokens for a Client using one or more issuance protocols.
- *Token Key: Keying material that can be used with an issuance protocol to create a signed token.

*Origin Name: The name that identifies the Origin, as included in a TokenChallenge.

Additionally, this document defines several terms that are unique to the rate-limited issuance protocol:

*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 Attester 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.

*Issuer Encapsulation Key: The public key used to encrypt values such as Origin Name in requests from Clients to the Issuer, so that Attesters cannot learn the Origin Name value. Each Issuer Encapsulation Key is used across all requests on the Issuer, for different Origins.

*Anonymous Origin ID: An identifier that is generated by the Client and marked on requests to the Attester, which represents a specific Origin anonymously. The Client generates a stable Anonymous Origin ID for each Origin Name, to allow the Attester to count token access without learning the Origin Name.

*Client Key: A public key chosen by the Client and shared only with the Attester; see [Section 8.2](#) for more details about this restriction.

*Client Secret: The secret key used by the Client during token issuance, whose public key (Client Key) is shared with the Attester.

*Issuer Origin Secret: A per-origin secret key used by the Issuer during token issuance, whose public key is not shared with anyone.

*Anonymous Issuer Origin ID: An identifier that is generated by Issuer based on an Issuer 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 Policy Window: a uint64 of seconds as defined in [Section 2](#).
2. Issuer Request URI: a token request URL for generating access tokens. For example, an Issuer URL might be https://

issuer.example.net/token-request. This parameter uses resource media type "text/plain".

3. Issuer Encapsulation Key: a EncapsulationKey structure as defined below to use when encapsulating information, such as the origin name, to the Issuer in issuance requests. This parameter uses resource media type "application/issuer-encap-key". The Npk parameter corresponding to the HpkeKdfId can be found in [HPKE].

```
opaque HpkePublicKey[Npk]; // defined in RFC9180
uint16 HpkeKemId;          // defined in RFC9180
uint16 HpkeKdfId;          // defined in RFC9180
uint16 HpkeAeadId;         // defined in RFC9180
```

```
struct {
    uint8 key_id;
    HpkeKemId kem_id;
    HpkePublicKey public_key;
    HpkeKdfId kdf_id;
    HpkeAeadId aead_id;
} EncapsulationKey;
```

The Issuer 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-policy-window	Issuer Policy Window as a JSON number
issuer-request-uri	Issuer Request URI resource URL as a JSON string
issuer-encap-key-uri	Issuer Encapsulation Key URI resource URL as a JSON string

Table 1

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

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

Issuer directory resources have the media type "application/json" and are located at the well-known location /.well-known/token-issuer-directory.

4. Token Challenge Requirements

Clients receive challenges for tokens, as described in [[AUTHSCHEME](#)].

For the rate-limited token issuance protocol described in this document, the name of the origin is sent in an encrypted message from the Client to the Issuer. If the `TokenChallenge.origin_info` field contains a single origin name, that origin name is used. If the `origin_info` field contains multiple origin names, the client selects the single origin name that presented the challenge. If the `origin_info` field is empty, the encrypted message is the empty string "".

The HTTP authentication challenge also SHOULD contain the following additional attribute:

"issuer-encap-key", which contains a base64url encoding of a `EncapsulationKey` as defined in [Section 3](#) to use when encrypting the Origin Name in issuance requests.

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, Attester, and Issuer, with the following steps:

1. The Client sends a token request containing a token request, encrypted origin name, and one-time-use public key and signature to the Attester
2. The Attester validates the request contents, specifically checking the request signature, and proxies the request to the Issuer
3. The Issuer validates the request against the signature, and processes its contents, and produces a token response sent back to the Attester
4. The Attester verifies the response and proxies the response to the Client

The Issuance protocol is designed such that Client, Attester, and Issuer learn only what is necessary for completing the protocol; see [Section 8.3](#) for more details.

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

- *RSA Blind Signatures [[BLINDSIG](#)], for issuing and constructing Tokens. This support is the same as used in the base publicly verifiable token issuance protocol [[ISSUANCE](#)]
- *[\[HPKE\]](#), for encrypting the origin server name in transit between Client and Issuer across the Attester.
- *Signatures with key blinding, as described in [[KEYBLINDING](#)], for verifying correctness of Client requests.

Clients and Issuers are required to implement all of these dependencies, whereas Attesters are required to implement signature with key blinding 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, a hostname referring to the Origin [[RFC6454](#)]. This is the name of the Origin that issued the token challenge. One or more names can be listed in the TokenChallenge.origin_info field. Rate-limited token issuance relies on the client selecting a single origin name from this list if multiple are present.
- *Token Key, a blind signature public key corresponding to the Issuer identified by the TokenChallenge.issuer_name.
- *Issuer Encapsulation Key, a public key used to encrypt request information corresponding to the Issuer identified by TokenChallenge.issuer_name.

Clients maintain a stable Client Key that they use for all communication with a specific Attester. Client Key is a public key, where the corresponding private key Client Secret is known only to the client.

If the client loses this (Client Key, Client Secret), they may generate a new tuple. The Attester will enforce if a client is allowed to use this new Client Key. See [Section 5.1.2](#) for details on this enforcement.

Clients also need to be able to generate an Anonymous Origin ID value that corresponds to the Origin Name, to send in requests to the Attester.

Anonymous 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 an Attester rejects a request upon detecting two Anonymous 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 Anonymous 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., Anonymous Origin ID = HKDF(secret=Client Secret, salt="", info=Origin Name).

5.1.2. Attester State

An Attester is required to maintain state for every authenticated Client. The mechanism of identifying a Client is specific to each Attester, and is not defined in this document. As examples, the Attester could use device-specific certificates or account authentication to identify a Client.

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

Attesters 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, an Attester maintains a policy window start and end time for each Issuer from which a Client requests a token.

For each tuple of (Client Key, Anonymous Origin ID, policy window), the Attester maintains the following state:

- *A counter of successful tokens issued

- *Whether or not a previous request was rejected by the Issuer

*The last received Anonymous Issuer Origin ID value for this Anonymous Origin ID, if any

5.1.3. Issuer State

Issuers maintain a stable Issuer Origin Secret that they use in calculating values returned to the Attester 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. See [Section 10.1](#) for details about generating and rotating the Issuer Origin Secret.

Issuers are expected to have the private key that corresponds to Issuer Encapsulation Key, which allows them to decrypt the Origin Name values in requests.

Issuers also need to know the set of valid 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 Token Key regularly to ensure that Client requests do not fail after 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, Attesters, and Issuers (see [Section 11.2](#)).

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-Attester requests ([Section 5.3](#)) and on Issuer-to-Attester 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-Attester requests ([Section 5.3](#)), and contains the bytes of Client Key. Its ABNF is:

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

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

```
Sec-Token-Request-Blind = sf-binary
```

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

```
Sec-Token-Request-Key = sf-binary
```

The "Sec-Token-Limit" is an Item Structured Header [[RFC8941](#)]. Its value MUST be an Integer. This header is sent on Issuer-to-Attester responses ([Section 5.5](#)), and contains the number of times a Client can retrieve a token for the requested Origin within a policy window, as set by the Issuer. Its ABNF is:

```
Sec-Token-Limit = sf-integer
```

5.3. Client-to-Attester Request

The Client and Attester MUST use a secure and Attester-authenticated HTTPS connection. They MAY use mutual authentication or mechanisms such as TLS certificate pinning, to mitigate the risk of channel compromise; see [Section 8](#) for additional about this channel.

Requests to the Attester need to indicate the Issuer Name to which issuance requests will be forwarded. Attesters SHOULD provide Clients with a URI template that contains one variable that contains the Issuer Name, "issuer", using Level 3 URI template encoding as defined in Section 1.2 of [[RFC6570](#)].

An example of an Attester URI templates is shown below:

```
https://attester.net/token-request{?issuer}
```

Attesters and Clients MAY agree on other mechanisms to specify the Issuer Name in requests.

The Client first creates an issuance request message for a random value nonce using the input TokenChallenge challenge and the Issuer key identifier key_id as follows:

```
nonce = random(32)
context = SHA256(challenge)
token_input = concat(0x0003, nonce, context, key_id)
blinded_msg, blind_inv = rsabssa_blind(pkI, token_input)
```

The Client then uses Client Key to generate its one-time-use request public key request_key and blind request_blind as described in [Section 7.1](#).

The Client then constructs a `InnerTokenRequest` value, denoted `origin_token_request`, combining `blinded_msg`, `request_key`, and the origin name as follows:

```
struct {
  uint8_t blinded_msg[Nk];
  uint8_t request_key[Npk];
  uint8_t padded_origin_name<0..2^16-1>;
} InnerTokenRequest;
```

This structure is initialized and then encrypted using Issuer Encryption Key, producing `encrypted_token_request`, as described in [Section 6](#).

Finally, the Client uses Client Secret to produce `request_signature` as described in [Section 7.1.2](#).

The Client then constructs a `TokenRequest` structure. This `TokenRequest` structure is based on the publicly verifiable token issuance path in [[ISSUANCE](#)], adding fields for the encrypted origin name and request signature.

```
struct {
  uint16_t token_type = 0x0003;
  uint8_t token_key_id;
  uint8_t issuer_encap_key_id[32];
  uint8_t encrypted_token_request<1..2^16-1>;
  uint8_t request_signature[Nsig];
} TokenRequest;
```

The structure fields are defined as follows:

`"token_type"` is a 2-octet integer, which matches the type in the challenge.

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

`"issuer_encap_key_id"` is a collision-resistant hash that identifies the Issuer Encryption Key, generated as `SHA256(EncapsulationKey)`.

`"encrypted_token_request"` is an encrypted structure that contains an `InnerTokenRequest` value, calculated as described in [Section 6](#).

`"request_signature"` is computed as described in [Section 7.1.2](#).

The Client then generates an HTTP POST request to send through the Attester to the Issuer, with the `TokenRequest` as the body. The media

type for this request is "message/token-request". The Client includes the "Sec-Token-Origin" header, whose value is Anonymous Origin ID; the "Sec-Token-Client" header, whose value is Client Key; the "Sec-Token-Request-Blind" header, whose value is request_blind; and the "Sec-Token-Request-Key" header, whose value is request_key. The Client sends this request to the Attester's proxy URI. An example request is shown below, where the Issuer Name is "issuer.net" and the Attester URI template is "https://attester.net/token-request{?issuer}"

```
:method = POST
:scheme = https
:authority = attester.net
:path = /token-request?issuer=issuer.net
accept = message/token-response
cache-control = no-cache, no-store
content-type = message/token-request
content-length = <Length of TokenRequest>
sec-token-origin = Anonymous Origin ID
sec-token-client = Client Key
sec-token-request-blind = request_blind
sec-token-request-key = request_key
```

<Bytes containing the TokenRequest>

If the Attester detects a token_type in the TokenRequest that it does not recognize or support, it MUST reject the request with an HTTP 400 error.

The Attester also checks to validate that the issuer_encap_key_id in the client's TokenRequest matches a known Issuer Encapsulation Key public key for the Issuer. For example, the Attester 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 Attester rejects the request with an HTTP 400 error. Note that this can lead to failures in the event of Issuer Issuer Encapsulation Key rotation; see [Section 9](#) for considerations.

The Attester finally validates the Client's stable mapping request as described in [Section 7.2](#). If this fails, the Attester MUST return an HTTP 400 error to the Client.

If the Attester 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 Anonymous Origin ID. See [Section 5.1.2](#) for more details.

If the Attester has stored state that a previous request for this Anonymous 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 Attester detects this Client has changed their Client Key more frequently than allowed as described in [Section 5.1.2](#), it SHOULD reject the request without forwarding it to the Issuer.

5.4. Attester-to-Issuer Request

Assuming all checks in [Section 5.3](#) succeed, the Attester generates an HTTP POST request to send to the Issuer with the Client's TokenRequest as the body. The Attester MUST NOT add information that will uniquely identify a Client, or associate the request with a small set of possible Clients. Extensions to this protocol MAY allow Attesters to add information that can be used to separate large populations, such as providing information about the country or region to which a Client belongs. An example request is shown below.

```
:method = POST
:scheme = https
:authority = issuer.net
:path = /token-request
accept = message/token-response
cache-control = no-cache, no-store
content-type = message/token-request
content-length = <Length of TokenRequest>
```

<Bytes containing the TokenRequest>

The Attester and the Issuer MUST use a secure and Issuer-authenticated HTTPS connection. Also, Issuers MUST authenticate Attesters, 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 8](#) for additional about this channel.

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

- *The TokenRequest contains a supported token_type
- *The TokenRequest.token_key_id and TokenRequest.issuer_encap_key_id correspond to known Token Keys and Issuer Encapsulation Keys held by the Issuer.
- *The TokenRequest.encrypted_token_request can be decrypted using the Issuer's private key (the private key associated with Issuer Encapsulation Key), and contains a valid InnerTokenRequest whose

unpadded origin name matches an Origin Name that is served by the Issuer. The Origin name associated with the InnerTokenRequest value might be the empty string "", as described in [Section 6](#), in which case the Issuer applies a cross-origin policy if supported. If a cross-origin policy is not supported, this condition is not met.

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

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

5.5. Issuer-to-Attester Response

If the Issuer is willing to give a token to the Client, the Issuer decrypts TokenRequest.encrypted_token_request to discover a InnerTokenRequest value. If this fails, the Issuer rejects the request with a 400 error. Otherwise, the Issuer validates and processes the token request with Issuer Origin Secret corresponding to the designated Origin as described in [Section 7.3](#). If this fails, the Issuer rejects the request with a 400 error. Otherwise, the output is index_key.

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

```
blind_sig = rsabssa_blind_sig(skP, InnerTokenRequest.blinded_msg)
```

skP is the private key corresponding to Token Key, known only to the Issuer. The Issuer then encrypts blind_sig to the Client as described in [Section 6.2](#), yielding encrypted_token_response.

The Issuer generates an HTTP response with status code 200 whose body consists of blind_sig, with the content type set as "message/token-response", the index_key set in the "Sec-Token-Origin" header, and the limit of tokens allowed for a Client for the Origin within a policy window set in the "Sec-Token-Limit" header. This limit SHOULD NOT be unique to a specific Origin, such that the Attester could use the value to infer which Origin the Client is accessing (see [Section 9](#)).

```
:status = 200
content-type = message/token-response
content-length = <Length of blind_sig>
sec-token-origin = index_key
sec-token-limit = Token limit

<Bytes containing the encrypted_token_response>
```

5.6. Attester-to-Client Response

Upon receipt of a successful response from the Issuer, the Attester extracts the "Sec-Token-Origin" header, and uses the value to determine Anonymous Issuer Origin ID as described in [Section 7.4](#).

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

The Attester also extracts the "Sec-Token-Limit" header, and compares the limit against the previous count of accesses for this Client for the Anonymous Origin ID. If the count is greater than or equal to the limit, the Attester drops the token and responds to the client with an HTTP 429 (Too Many Requests) error.

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

When the Attester detects successful token issuance, it MUST increment the counter in its state for the number of tokens issued to the Client for the Anonymous Origin ID.

Upon receipt, the Client decrypts the blind_sig from encrypted_token_response as described in [Section 6.2](#). If successful, the Client then processes the response as follows:

```
authenticator = rsabssa_finalize(pkI, token_input, blind_sig, blind_inv)
```

If this succeeds, the Client then constructs a token as described in [\[AUTHSCHEME\]](#) as follows:

```
struct {
    uint16_t token_type = 0x0003
    uint8_t nonce[32];
    uint8_t context[32];
    uint8_t token_key_id[Nid];
    uint8_t authenticator[Nk]
} Token;
```

6. Encrypting Origin Token Requests and Responses

Clients encapsulate token request information to the Issuer using the Issuer Encapsulation Key. Issuers decrypt the token request using their corresponding private key. This process yields the decrypted token request as well as a shared encryption context between Client and Issuer. Issuers encapsulate their token response to the Client using an ephemeral key derived from this shared encryption context. This process ensures that the Attester learns neither the token request or response information.

Client to Issuer encapsulation is described in [Section 6.1](#), and Issuer to Client encapsulation is described in [Section 6.2](#).

6.1. Client to Issuer Encapsulation

Given a EncapsulationKey (Issuer Encapsulation Key), Clients produce `encrypted_token_request` using the following values:

- *the one octet key identifier from the Name Key, `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 following inputs defined in [Section 5.3](#): `token_key_id`, `blinded_msg`, `request_key`, `origin_name`, and `issuer_encap_key_id`.

Together, these are used to encapsulate an `InnerTokenRequest` and produce an encrypted token request (`encrypted_token_request`).

`origin_name` contains the name of the origin that initiated the challenge, as taken from the `TokenChallenge.origin_info` field. If the `TokenChallenge.origin_info` field is empty, `origin_name` is set to the empty string `""`.

The process for generating `encrypted_token_request` from `blinded_msg`, `request_key`, and `origin_name` values is 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 `TokenRequest` structure.
3. Pad `origin_name` with `N` zero bytes, where $N = 31 - ((L - 1) \% 32)$ and `L` is the length of `origin_name`. If `L` is 0, `N = 32`. Denote this padding process as the function `pad`.
4. Encrypt (seal) the padded `origin_name` with `aad` as associated data using context, yielding ciphertext `ct`.
5. Concatenate the values of `aad`, `enc`, and `ct`, yielding `encrypted_token_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, "InnerTokenRequest")
aad = concat(encode(1, keyID),
            encode(2, kemID),
            encode(2, kdfID),
            encode(2, aeadID),
            encode(2, token_type),
            encode(1, token_key_id),
            encode(32, issuer_encap_key_id))
padded_origin_name = pad(origin_name)
input = concat(encode(Nk, blinded_msg),
              encode(49, request_key),
              encode(len(padded_origin_name), padded_origin_name))
ct = context.Seal(aad, input)
encrypted_token_request = concat(enc, ct)
```

Issuers reverse this procedure to recover the `InnerTokenRequest` value by computing the AAD as described above and decrypting `encrypted_token_request` with their private key `skI` (the private key corresponding to `pkI`). The `origin_name` value is recovered from `InnerTokenRequest.padded_origin_name` by stripping off padding bytes. In pseudocode, this procedure is as follows:

```

enc, ct = parse(encrypted_token_request)
aad = concat(encode(1, keyID),
             encode(2, kemID),
             encode(2, kdfID),
             encode(2, aeadID),
             encode(2, token_type),
             encode(1, token_key_id),
             encode(32, issuer_encap_key_id))
context = SetupBaseR(enc, skI, "TokenRequest")
origin_token_request, error = context.Open(aad, ct)

```

The `InnerTokenRequest.blinded_msg` and `InnerTokenRequest.request_key` values, along with the unpadded `origin_name` value, are used by the Issuer as described in [Section 5.4](#).

6.2. Issuer to Client Encapsulation

Given an HPKE context `context` computed in [Section 6.1](#), Issuers encapsulate their token response `blind_sig`, yielding an encrypted token response `encrypted_token_response`, to the Client as follows:

1. Export a secret `secret` from `context`, using the string `"OriginTokenResponse"` as context. The length of this secret is $\max(N_n, N_k)$, where N_n and N_k are the length of AEAD key and nonce associated with `context`.
2. Generate a random value of length $\max(N_n, N_k)$ bytes, called `response_nonce`.
3. Extract a pseudorandom key `prk` using the Extract function provided by the KDF algorithm associated with `context`. The `ikm` input to this function is `secret`; the salt input is the concatenation of `enc` (from `enc_request`) and `response_nonce`.
4. Use the Expand function provided by the same KDF to extract an AEAD key `key`, of length N_k - the length of the keys used by the AEAD associated with `context`. Generating key uses a label of `"key"`.
5. Use the same Expand function to extract a nonce `nonce` of length N_n - the length of the nonce used by the AEAD. Generating nonce uses a label of `"nonce"`.
6. Encrypt `blind_sig`, passing the AEAD function `Seal` the values of `key`, `nonce`, empty `aad`, and a `pt` input of `request`, which yields `ct`.
7. Concatenate `response_nonce` and `ct`, yielding an Encapsulated Response `enc_response`. Note that `response_nonce` is of fixed-

length, so there is no ambiguity in parsing either response_nonce or ct.

In pseudocode, this procedure is as follows:

```
secret = context.Export("OriginTokenResponse", Nk)
response_nonce = random(max(Nn, Nk))
salt = concat(enc, response_nonce)
prk = Extract(salt, secret)
aead_key = Expand(prk, "key", Nk)
aead_nonce = Expand(prk, "nonce", Nn)
ct = Seal(aead_key, aead_nonce, "", blind_sig)
encrypted_token_response = concat(response_nonce, ct)
```

Clients decrypt encrypted_token_response by reversing this process. That is, they first parse enc_response into response_nonce and ct. They then follow the same process to derive values for aead_key and aead_nonce.

The client uses these values to decrypt ct using the Open function provided by the AEAD. Decrypting might produce an error, as follows:

```
blind_sig, error = Open(aead_key, aead_nonce, "", ct)
```

7. Anonymous Issuer Origin ID Computation

This section describes the Client, Attester, and Issuer behavior in computing Anonymous Issuer Origin ID, the stable mapping based on client identity and origin name. At a high level, this functionality computes $y = F(x, k)$, where x is a per-Client secret and k is a per-Origin secret, subject to the following constraints:

- *The Attester only learns y if the Client in possession of x engages with the protocol;
- *The Attester prevents a Client with private input x from running the protocol for input x' that is not equal to x ;
- *The Issuer does not learn x , nor does it learn when two requests correspond to the same private value x ; and
- *Neither the Client nor Attester learn k .

The interaction between Client, Attester, and Issuer in computing this functionality is shown below.



The protocol for computing this functionality is divided into sections for each of the participants. [Section 7.1](#) describes Client behavior for initiating the computation with its per-Client secret, [Section 7.2](#) describes Attester behavior for verifying Client requests, [Section 7.3](#) describes Issuer behavior for computing the mapping with its per-Origin secret, and [Section 7.4](#) describes the final Attester step for computing the client-origin index.

The index computation is based on a signature scheme with key blinding and unblinding support, denoted BKS, as described in [[KEYBLINDING](#)]. Such a scheme has the following functions:

- *BKS-KeyGen(): Generate a random private and public key pair (sk, pk).
- *BKS-BlindKeyGen(): Generate a random blinding key bk.
- *BKS-BlindPublicKey(pk, bk): Produce a blinded public key based on the input public key pk and blind key bk according to [[KEYBLINDING](#)], Section 6.1.
- *BKS-Verify(pk, msg, sig): Verify signature sig over input message msg against the public key pk, producing a boolean value indicating success.
- *BKS-BlindKeySign(sk_sign, sk_blind, msg): Sign input message msg with signing key sk_sign and blind key sk_blind according to [[KEYBLINDING](#)], Section 6.2, and produce a signature of size Nsig bytes.
- *BKS-SerializePrivatekey(sk): Serialize a private key to a byte string of length Nsk.
- *BKS-DeserializePrivatekey(buf): Attempt to deserialize a private key from an Nsk-byte string buf. This function can fail if buf does not represent a valid private key.
- *BKS-SerializePublicKey(pk): Serialize a public key to a byte string of length Npk.

*BKS-DeserializePublicKey(buf): Attempt to deserialize a public key of length Npk. This function can fail if buf does not represent a valid public key.

Additionally, each BKS scheme has a corresponding hash function, denoted Hash. The implementation of each of these functions depends on the issuance protocol token type. See [Section 11.1](#) for more details.

7.1. Client Behavior

This section describes the Client behavior for generating an one-time-use request key and signature. Clients provide their Client Secret as input to the request key generation step, and the rest of the token request inputs to the signature generation step.

7.1.1. Request Key

Clients produce request_key by masking Client Key and Client Secret with a randomly chosen blind. Let pk_sign and sk_sign denote Client Key and Client Secret, respectively. This process is done as follows:

1. Generate a random blind key, sk_blind.
2. Blind pk_sign with sk_blind to compute a blinded public key, request_key.
3. Output the blinded public key.

In pseudocode, this is as follows:

```
sk_blind = BKS-BlindKeyGen()  
blinded_key = BKS-BlindPublicKey(pk_sign, sk_blind)  
request_key = BKS-SerializePublicKey(blinded_key)  
request_blind = BKS-SerializePrivatekey(sk_blind)
```

7.1.2. Request Signature

Clients produce signature of their request based on the following inputs defined in [Section 5.3](#): token_key_id, blinded_msg, request_key, issuer_encap_key_id, encrypted_token_request. This process requires the blind value sk_blind produced during the [Section 7.1.1](#) process. As above, let pk and sk denote Client Key and Client Secret, respectively. Given these values, this signature process works as follows:

1. Concatenate all signature inputs to yield a message to sign.

2. Compute a signature with the blind `sk_blind` over the input message using Client Secret, `sk_sign` as the signing key.
3. Output the signature.

In pseudocode, this is as follows:

```
context = concat(token_type,  
                token_key_id,  
                issuer_encap_key_id,  
                encrypted_token_request)  
request_signature = BKS-BlindKeySign(sk_sign, sk_blind, context)
```

7.2. Attester Behavior (Client Request Validation)

Given a `TokenRequest` request containing `request_key`, `request_signature`, and `request_blind`, as well as Client Key `pk_blind`, Attesters verify the signature as follows:

1. Check that `request_key` is a valid public key. If this fails, abort.
2. Check that `request_blind` is a valid private key. If this fails, abort.
3. Blind the Client Key `pk_sign` by blind `sk_blind`, yielding a blinded key. If this does not match `request_key`, abort.
4. Verify `request_signature` over the contents of the request, excluding the signature itself, using `request_key`. If signature verification fails, abort.

In pseudocode, this is as follows:

```
blind_key = BKS-DeserializePublicKey(request_key)  
sk_blind = BKS-DeserializePrivatekey(request_blind)  
pk_blind = BKS-BlindPublicKey(pk_sign, sk_blind)  
if pk_blind != blind_key:  
    raise InvalidParameterError  
  
context = parse(request[..len(request)-Nsig]) // this matches context co  
valid = BKS-Verify(blind_key, context, request_signature)  
if not valid:  
    raise InvalidSignatureError
```

7.3. Issuer Behavior

Given an Issuer Origin Secret (denoted `sk_origin`) and a `TokenRequest`, from which `request_key` and `request_signature` are

parsed, Issuers verify the request signature and compute a response as follows:

1. Check that `request_key` is a valid public key. If this fails, abort.
2. Verify `request_signature` over the contents of the request, excluding the signature itself, using `request_key`. If signature verification fails, abort.
3. Blind `request_key` by Issuer Origin Secret, `sk_origin`, yielding an index key.
4. Output the index key.

In pseudocode, this is as follows:

```
blind_key = BKS-DeserializePublicKey(request_key)
context = parse(request[..len(request)-Nsig]) // this matches context co
valid = BKS-Verify(blind_key, context, request_signature)
if not valid:
    raise InvalidSignatureError

evaluated_key = BKS-BlindPublicKey(request_key, sk_origin)
index_key = BKS-SerializePublicKey(evaluated_key)
```

7.4. Attester Behavior (Index Computation)

Given an Issuer response `index_key`, Client blind `sk_blind`, and Client Key (denoted `pk_sign`), Attesters complete the Anonymous Issuer Origin ID computation as follows:

1. Check that `index_key` is a valid public key. If this fails, abort.
2. Unblind the `index_key` using the Client blind `sk_blind`, yielding the index result.
3. Run HKDF [[RFC5869](#)] with the hash function corresponding to the BKS scheme, using the index result as the secret, Client Key `pk_sign` as the salt, and ASCII string "anon_issuer_origin_id" as the info string, yielding Anonymous Issuer Origin ID.

In pseudocode, this is as follows:

```
evaluated_key = BKS-DeserializePublicKey(index_key)
unblinded_key = BKS-UnblindPublicKey(evaluated_key, sk_blind)

index_result = BKS-SerializePublicKey(unblinded_key)
pk_encoded = BKS-SerializePublicKey(pk_sign)

anon_issuer_origin_id = HKDF-Hash(secret=index_result,
    salt=pk_encoded, info="anon_issuer_origin_id")
```

8. Security Considerations

This section describes security considerations relevant to the use of this protocol.

8.1. Channel Security

An attacker that can act as an intermediate between Attester 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 an Attester 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.

8.2. Token Request Unlinkability and Unforgeability

Client token requests are constructed such that an Issuer cannot distinguish between any two token requests from the same Client and two requests from different Clients. We refer to this property as issuance unlinkability. This property is achieved by the way the tokens are constructed. In particular, `TokenRequest.request_key` and `TokenRequest.request_signature` are the only value in a `TokenRequest` that is derived from per-Client information, i.e., the Client Secret.

`TokenRequest.request_key` is computed using a freshly generated blind for each token request. As a result, the value of `TokenRequest.request_key` in one token request is statistically independent from Client Key. Similarly, `TokenRequest.request_signature` is computed using the same freshly generated blind as `TokenRequest.request_key` for each token request, and the resulting signature is therefore independent from signatures produced using Client Secret. More details about this unlinkability property can be found in [[KEYBLINDING](#)].

This unlinkability property is only intended for requests observed by the Issuer. In contrast, the Attester is required to link requests from the same Client together for the purposes of enforcing rate limits. This Attester does this by observing the Client Key.

Importantly, the Client Key is not sent to the Issuer during the issuance flow, as doing this would allow the Issuer to trivially link two requests to the same Client.

The token request signature is also required to be unforgeable. Informally, unforgeability means that no entity can produce a valid (message, signature) pair for any blinding key without access to the private signing key. Importantly, this means the Attester cannot forge signatures on behalf of a given Client in an attempt to learn the origin name.

8.3. Information Disclosure

The protocol in this document is designed such that information pertaining to issuance of a token is limited to parties that need it for completing the protocol. In particular, honest-but-curious Attesters learn only the Anonymous Issuer Origin ID as described in [Section 7](#), any per-Client information necessary for attestation, and the target Issuer for a given token request. The Attester does not directly learn the origin name associated with a given token request, though it does learn the distribution of tokens across Client interactions. This auxiliary information could be used to infer the Origin for a given token. For example, if an Issuer has only two configured Origins, each with a different token request pattern, then the distribution of Client tokens might reveal the Origin associated with a given token.

Malicious or otherwise compromised Attesters can choose to not follow the protocol described in this specification, allowing, for example, Clients to bypass rate limits imposed by Origins. Moreover, malicious Attesters could reveal the per-request blind (request_blind) to Issuers, breaking the unlinkability property described in [Section 8.2](#).

Honest-but-curious Issuers only learn the Attester that vouches for a particular Client's token request and the origin name associated with a token request. Issuers do not learn the Anonymous Issuer Origin ID or any per-Client information used when creating a token request.

Conversely, malicious Issuers that do not follow the protocol can choose to not validate the token request signature, thereby allowing others to forge token requests in an attempt to learn the origin name. Malicious Issuers can also rotate token signing keys or Issuer Origin Secret values frequently in an attempt to bypass Attester-enforced rate limits. Both of these are detectable by the Attester, though. Issuers can also lie about per-origin rate limits without detection, e.g., by increasing the limit to a value well beyond any

configured limit by an Origin, or return different limits for different origins to the Attester.

Clients learn the output token. They do not learn the Anonymous Issuer Origin ID, though the security of the protocol does not depend on keeping this value secret from Clients. Moreover, even malicious Clients cannot tamper with per-Client state stored on the Attester for other Clients, as doing so requires knowledge of their unique Client Secret.

9. Privacy Considerations

This section describes privacy considerations relevant to use of this protocol.

9.1. 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.

9.2. Origin Verification

Rate-limited 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 name of the origin that is providing the HTTP authentication challenge is present in the `TokenChallenge.origin_info` list ([\[AUTHSCHEME\]](#)), 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.

9.3. 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 Encapsulation Key, the Attester can observe and validate the token_key_id presented by the Client to the Issuer. As described in [Section 5](#), Attesters MUST validate that the token_key_id in the Client's TokenRequest matches a known public key for the Issuer. The Attester 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.

9.4. Origin Identification

As stated in [Section 1.2](#), the design of this protocol is such that Attesters cannot learn the identity of origins that Clients are accessing. The Origin Name itself is encrypted in the request between the Client and the Issuer, so the Attester cannot directly learn the value. However, in order to prevent the Attester from inferring the value, additional constraints need to be added:

*Each Issuer SHOULD serve tokens to a large number of Origins. A one-to-one relationship between Origin and Issuer would allow an Attester to infer which Origin is accessed simply by observing the Issuer identity.

*Issuers SHOULD NOT return rate-limit values that are specific to Origins, such that an Attester can infer which Origin is accessed by observing the rate limit. This can be mitigated by having many Origins share the same rate-limit value.

Some deployments MAY choose to relax these requirements, such as in cases where the origins being accessed are ubiquitous or do not correspond to user-specific behavior.

9.5. Collusion Among Different Entities

Collusion among the different entities in the Privacy Pass architecture can result in exposure of a client's per-origin access patterns.

For this issuance protocol, Issuers and Attesters should be run by mutually distinct organizations to limit information sharing. A single entity running an Issuer and Attester for a single token issuance flow can view the origins being accessed by a given client. Running the Issuer and Attester in this 'single Issuer/Attester' fashion reduces the privacy promises of no one entity being able to learn Client browsing patterns. This may be desirable for a

redemption flow that is limited to specific Issuers and Attesters, but should be avoided where hiding origin names from the Attester is desirable.

If a Attester 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 Attester can reveal the Client identity (as known to the Attester) to the Origin, especially if repeated over multiple accesses.

10. Deployment Considerations

10.1. Token Key Management

Issuers SHOULD generate a new (Token Key, Issuer 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(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.

11. IANA considerations

11.1. Token Type

This document updates the "Token Type" Registry ([\[AUTHSCHEME\]](#)) with the following value:

Value	Name	Publicly Verifiable	Public Metadata	Private Metadata	Nk	Nid	Reference
0x0003	Rate-Limited Blind RSA(SHA-384, 2048-bit) with ECDSA(P-384, SHA-384)	Y	N	N	512	32	This document
0x0004	Rate-Limited Blind RSA(SHA-384, 2048-bit) with Ed25519(SHA-512)	Y	N	N	512	32	This document

Table 2: Token Types

The details of the signature scheme with key blinding and unblinding functions for each token type above are described in the following sections.

11.1.1.1. ECDSA-based Token Type

This section describes the implementation details of the signature scheme with key blinding and unblinding functions introduced in [Section 7](#) using [\[ECDSA\]](#) with P-384 as the underlying elliptic curve and SHA-384 as the corresponding hash function.

*BKS-KeyGen(): Generate a random ECDSA private and public key pair (sk, pk).

*BKS-BlindKeyGen(): Generate a random ECDSA private key bk.

*BKS-BlindPublicKey(pk, bk): Produce a blinded public key based on the input public key pk and blind bk according to [\[KEYBLINDING\]](#), Section 6.1.

*BKS-Verify(pk, msg, sig): Verify the DER-encoded [\[X690\]](#) BKS-Sig-Value signature sig over input message msg against the ECDSA public key pk, producing a boolean value indicating success.

*BKS-BlindKeySign(sk_sign, sk_blind, msg): Sign input message msg with signing key sk_sign and blind sk_blind according to [\[KEYBLINDING\]](#), Section 6.2, and serializes the resulting signature pair (r, s) in "raw" form, i.e., as the concatenation of two 48-byte, big endian scalars, yielding an Nsig=96 byte signature.

*BKS-SerializePrivatekey(sk): Serialize an ECDSA private key using the Field-Element-to-Octet-String conversion according to [\[SECG\]](#).

*BKS-DeserializePrivatekey(buf): Attempt to deserialize an ECDSA private key from a 48-byte string buf using Octet-String-to-Field-Element from [\[SECG\]](#). This function can fail if buf does not represent a valid private key.

*BKS-SerializePublicKey(pk): Serialize an ECDSA public key using the compressed Elliptic-Curve-Point-to-Octet-String method according to [\[SECG\]](#).

*BKS-DeserializePublicKey(buf): Attempt to deserialize a public key using the compressed Octet-String-to-Elliptic-Curve-Point method according to [\[SECG\]](#), and then performs partial public-key validation as defined in section 5.6.2.3.4 of [\[KEYAGREEMENT\]](#). This validation includes checking that the coordinates are in the correct range, that the point is on the curve, and that the point is not the point at infinity.

11.1.2. Ed25519-based Token Type

This section describes the implementation details of the signature scheme with key blinding and unblinding functions introduced in [Section 7](#) using Ed25519 as described in [\[RFC8032\]](#).

*BKS-KeyGen(): Generate a random Ed25519 private and public key pair (sk, pk), where sk is randomly generated 32 bytes (See [\[RFC4086\]](#) for information about randomness generation) and pk is computed according to [\[RFC8032\]](#), [Section 5.1.5](#).

*BKS-BlindKeyGen(): Generate and output 32 random bytes.

*BKS-BlindPublicKey(pk, bk): Produce a blinded public key based on the input public key pk and blind bk according to [\[KEYBLINDING\]](#), [Section 5.1](#).

*BKS-Verify(pk, msg, sig): Verify the signature sig over input message msg against the Ed25519 public key pk, as defined in [\[RFC8032\]](#), [Section 5.1.7](#), producing a boolean value indicating success.

*BKS-BlindKeySign(sk_sign, sk_blind, msg): Sign input message msg with signing key sk_sign and blind sk_blind according to [\[KEYBLINDING\]](#), [Section 5.2](#), yielding an Nsig=64 byte signature.

*BKS-SerializePrivateKey(sk): Identity function which outputs sk as an Nsk=32 byte buffer.

*BKS-DeserializePrivateKey(buf): Identity function which outputs buf interpreted as sk.

*BKS-SerializePublicKey(pk): Identity function which outputs pk as an Npk=32 byte buffer.

*BKS-DeserializePublicKey(buf): Identity function which outputs buf interpreted as pk.

11.2. 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>](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-Request-Blind	http	std	This document
Sec-Token-Request-Key	http	std	This document
Sec-Token-Limit	http	std	This document

Figure 1: Registered HTTP Header

12. References

12.1. Normative References

- [ARCH] Davidson, A., Iyengar, J., and C. A. Wood, "Privacy Pass Architectural Framework", Work in Progress, Internet-Draft, draft-ietf-privacypass-architecture-03, 7 March 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-architecture-03>>.
- [AUTHSCHEME] Pauly, T., Valdez, S., and C. A. Wood, "The Privacy Pass HTTP Authentication Scheme", Work in Progress, Internet-Draft, draft-ietf-privacypass-auth-scheme-02, 4 April 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-auth-scheme-02>>.
- [BLINDSIG] Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-rsa-blind-signatures-03, 2 February 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-rsa-blind-signatures-03>>.
- [ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry - The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.
- [HPKE] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <<https://www.rfc-editor.org/rfc/rfc9180>>.
- [ISSUANCE] Celi, S., Davidson, A., Faz-Hernandez, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-

protocol-04, 5 April 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-04>>.

- [**KEYAGREEMENT**] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar3, April 2018, <<https://doi.org/10.6028/nist.sp.800-56ar3>>.
- [**KEYBLINDING**] Denis, F., Eaton, E., and C. A. Wood, "Key Blinding for Signature Schemes", Work in Progress, Internet-Draft, draft-dew-cfrg-signature-key-blinding-01, 7 March 2022, <<https://datatracker.ietf.org/doc/html/draft-dew-cfrg-signature-key-blinding-01>>.
- [**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>>.
- [**RFC5869**] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/rfc/rfc5869>>.
- [**RFC6454**] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <<https://www.rfc-editor.org/rfc/rfc6454>>.
- [**RFC6570**] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <<https://www.rfc-editor.org/rfc/rfc6570>>.
- [**RFC8032**] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/

RFC8032, January 2017, <<https://www.rfc-editor.org/rfc/rfc8032>>.

- [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>>.
- [SECG] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", 2009, <<https://secg.org/sec1-v2.pdf>>.
- [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>>.
- [X690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8824-1:2021 , February 2021.

12.2. Informative References

- [BASIC-ISSUANCE] Celi, S., Davidson, A., Faz-Hernandez, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-04, 5 April 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-04>>.
- [RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/rfc/rfc4086>>.

Appendix A. Acknowledgements

The authors of this document would like to acknowledge feedback from contributors to the Privacy Pass working group for their help in improving this document. The authors also thank Frank Denis and David Schinazi for their contributions.

Appendix B. Test Vectors

This section includes test vectors for Origin Name encryption in [Section 6](#) and Anonymous Origin ID computation in [Section 7](#). Test vectors for the token request and response protocol can be found in [\[ISSUANCE\]](#).

B.1. Origin Name Encryption Test Vector

The test vector below for the procedure in [Section 6](#) lists the following values:

*origin_name: The Origin Name to encrypt, represented as a hexadecimal string.

*kem_id, kdf_id, aead_id: The HPKE algorithms comprising the ciphersuite DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, AES-128-GCM.

*issuer_encap_key_seed: The seed used to derive the private key corresponding to Issuer Encapsulation Key via the DeriveKeyPair function as defined in [Section 7.1.3.](#) of [[HPKE](#)], represented as a hexadecimal string.

*issuer_encap_key: The public Issuer Encapsulation Key, represented as a hexadecimal string.

*token_type: The type of the protocol specified in this document.

*token_key_id: The ID of Token Key computed as in [Section 5.3](#), a single octet.

*blinded_msg: A random blinded_msg value, represented as a hexadecimal string.

*request_key: A random request_key value, represented as a hexadecimal string.

*issuer_encap_key_id: The Issuer Encapsulation Key ID computed as in [Section 5.3](#), represented as a hexadecimal string.

*encrypted_token_request: The encrypted InnerTokenRequest, represented as a hexadecimal string.

origin_name: 746573742e6578616d706c65
kem_id: 32
kdf_id: 1
aead_id: 1
issuer_encap_key_seed:
d2653816496f400baec656f213f1345092f4406af4f2a63e164956c4c3d240ca
issuer_encap_key: 010020d7b6a2c10e75c4239feb9897e8d23f3c377d78e7903611
53167736a24a9c5400010001
token_type: 3
token_key_id: 125
blinded_msg: 89da551a48270b053e53c9eb741badf89e43cb7e66366bb936e11fb2aa0
d30866986a790378bb9fc6a7cf5c32b7b7584d448ffa4ced3be650e354b3136428a52ec0
b27c4103c5855c2b9b4f521ad0713c800d7e6925b6c62e1a6f58b31d13335f468cf509b7
46a16e79b23862d277d0880706c3fb84b127d94faf8d6d2f3e124e681994441b19be084e
c5c159bcd0abab433bbc308d90ea2cabdf4216e1b07155be66a048d686e383ca1e517ab8
0025bb4849d98beb8c3d05d045c1167cb74f4451d8f85695babb604418385464f21f9a81
5fb850ed83fd16a966130427e5637816501f7a79c0010e06adeba55781ceb50f56eae152
ebd06f3cef80dc7ab121d
request_key: 0161d905e4e37f515cb61f863b60e5896aa9e4a17dbe238e752a144c64a
5412e244f0b1f75e010831e185cac023d33cb20
issuer_encap_key_id:
dd2c6de3091f1873643233d229a7a0e9defe0f9fe43f6a7c42ae3a6b16f77837
encrypted_token_request: 82ef7c068506bcabc27d068a51c7ead2cbaf600b76a15e4
d9df99a0da676da5a073fcc8f5ac77b25064d7379037b4e1b186977cface31eceb611978
c73c9aef38c9a0e8ae846881624fa6d454523a0a91d22b02b022891d0469deebd66a912a
a1ab3391b203e92e0a681f0a10c2a2d59b668daf1e5219ed16227d707fa0e8e29188bd58
7ab38b3584564ce9b6538ba82e301cfed4231a2fa4f64a67285a1b9bf648e25f3eb1644c
88d43552bdea6e4bfcbaef0de3ac245e0432be6b019494927fde0743b775f9efe8ca5fef
afbf2048890d54618d408a6001fc8fb276f6828c46f4fe1381e9775eec72ee47593df738
95d18952440d33756d78caea4bd8218950d35afa6c46c535211eda39da277260cb8dab7c
00c6840a745e8150a6ee4893e72b6a51382f877f8c05a15e891a2bde07049760f0f09879
78d78b97e47ecaf90a44996d724dd3720e308abbbf04f672bc5a4db573291986be191b06
03ff521accb6fa081c151c758f3092a89fc6ef591934ff4bc860896c57f83a31b237dd1b
803516c

B.2. Anonymous Origin ID Test Vector

The test vector below for the procedure in [Section 7](#) lists the following values:

*sk_client: Client Secret, serialized and represented as a hexadecimal string.

*pk_client: Client Key, serialized and represented as a hexadecimal string.

*sk_origin: Origin Secret, serialized and represented as a hexadecimal string.

*request_blind: The request_blind value computed in [Section 7.1.1](#), represented as a hexadecimal string.

*index_key: The index_key value computed in [Section 7.3](#), represented as a hexadecimal string.

*anon_issuer_origin_id: The anon_issuer_origin_id value computed in [Section 7.4](#), represented as a hexadecimal string.

sk_sign: f6e6a0c9de38663ca539ff2e6a04e4fca11dc569794dc405e2d17439d6ce4f67abb2b81a1852e0db993b6a0452eb60d6
pk_sign: 032db7483c710673e6999a5fb2a2c6eac1d891f89bbf58d985ff168d182ad51605c4369280efabb7692f661162e683f03c
sk_origin: 85de5fbbd787da5093da0adb240eba0cc6ea90d72032fc4b6925dd7d0ab1da1e5ae0be27fe9f59e9ec7e1f1b15b28696
request_blind: 0698a149fb9d16bcb0a856062f74f9191e82b35e91224a57abce60f5b79f03a669c6b5e093d57e647865f9fd4305b5a9
request_key: 0287b0ce6b957111263d8c6126af96d400bd5a9d0b0f62ade15ab78944606c209470ced7086d3c418dd32bf9245fd42678
index_key: 03188bec3dc02d2382b5251b6b4fd729d0472bbddf008c5e93b7c12270d9f57dde111c861c13be53822a1cebb604946066
anon_issuer_origin_id: 9b0f980e5c1142fd4401e5cd2107a87d22b73753b0d5dc93f9a8f5ed2ee7db78163c6a93cc41ae8158d562381c51ee

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