The Privacy Pass HTTP Authentication Scheme

draft-ietf-privacypass-auth-scheme-02

Abstract

This document defines an HTTP authentication scheme that can be used by clients to redeem Privacy Pass tokens with an origin. It can also be used by origins to challenge clients to present an acceptable Privacy Pass token.

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 6 October 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.
1. Introduction

Privacy Pass tokens are unlinkable authenticators that can be used to anonymously authorize a client (see [I-D.ietf-privacypass-architecture]). A client possessing such a token is able to prove that it was able to get a token issued by a token issuer -- based on some check from a token issuer, such as authentication or solving a CAPTCHA -- without allowing the relying party redeeming the client’s token (the origin) to link it with issuance flow.

Different types of authenticators, using different token issuance protocols, can be used as Privacy Pass tokens.

This document defines a common HTTP authentication scheme ([RFC7235]), PrivateToken, that allows clients to redeem various kinds of Privacy Pass tokens.
Clients and relying parties interact using this scheme to perform the token challenge and token redemption flow. Clients use a token issuance protocol to actually fetch tokens to redeem.

![Figure 1: Token Architectural Components](image)

In addition to working with different token issuance protocols, this scheme supports optionally associating tokens with origin-chosen contexts and specific origin names. Relying parties that request and redeem tokens can choose a specific kind of token, as appropriate for its use case. These options allow for different deployment models to prevent double-spending, and allow for both interactive (online challenges) and non-interactive (pre-fetched) tokens.

1.1. 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 document uses the terms "Client", "Origin", "Issuer", "Issuance Protocol", and "Token" as defined in [I-D.ietf-privacypass-architecture]. It additionally uses the following terms in more specific ways:

* Issuer key: Keying material that can be used with an issuance protocol to create a signed token.
* Token challenge: A requirement for tokens sent from an origin to a client, using the "WWW-Authenticate" HTTP header. This challenge is bound to a specific token issuer and issuance protocol, and may be additionally bound to a specific context or origin name.

* Token redemption: An action by which a client presents a token to an origin, using the "Authorization" HTTP header.

2. HTTP Authentication Scheme

Token redemption is performed using HTTP Authentication ([RFC7235]), with the scheme "PrivateToken". Origins challenge clients to present a token from a specific issuer (Section 2.1). Once a client has received a token from that issuer, or already has a valid token available, it presents the token to the origin (Section 2.2).

2.1. Token Challenge

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

Origins that support this authentication scheme need to handle the following tasks:

1. Select which issuer to use, and configure the issuer name and token-key to include in WWW-Authenticate challenges.

2. Determine a redemption context construction to include in the TokenChallenge, as discussed in Section 2.1.1.

3. Select the origin information to include in the TokenChallenge. This can be empty to allow fully cross-origin tokens, a single origin name that matches the origin itself, or a list of origin names containing the origin.

The TokenChallenge message has the following structure:

```c
struct {
    uint16_t token_type;
    opaque issuer_name<1..2^16-1>);
    opaque redemption_context<0..32>);
    opaque origin_info<0..2^16-1>);
} TokenChallenge;
```

The structure fields are defined as follows:
* "token_type" is a 2-octet integer, in network byte order. This type indicates the issuance protocol used to generate the token. Values are registered in an IANA registry, Section 6.2. Challenges with unsupported token_type values MUST be ignored.

* "issuer_name" is a string containing the name of the issuer. This is a hostname that is used to identify the issuer that is allowed to issue tokens that can be redeemed by this origin. The string is prefixed with a 2-octet integer indicating the length, in network byte order.

* "redemption_context" is an optional field. If present, it allows the origin to require that clients fetch tokens bound to a specific context, as opposed to reusing tokens that were fetched for other contexts. See Section 2.1.1 for example contexts that might be useful in practice. When present, this value is a 32-byte context generated by the origin. Valid lengths for this field are either 0 or 32 bytes. The field is prefixed with a single octet indicating the length. Challenges with redemption_context values of invalid lengths MUST be ignored.

* "origin_info" is an optional string containing one or more origin names, which allows a token to be scoped to a specific set of origins. The string is prefixed with a 2-octet integer indicating the length, in network byte order. If empty, any non-origin-specific token can be redeemed. If the string contains multiple origin names, they are delimited with commas "," without any whitespace.

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

* "challenge", which contains a base64url-encoded [RFC4648] TokenChallenge value. Since the length of the challenge is not fixed, the base64url data MUST include padding. This MUST be unique for every 401 HTTP response to prevent replay attacks. This attribute is required for all challenges.

* "token-key", which contains a base64url encoding of the public key for use with the issuance protocol indicated by the challenge. Since the length of the key is not fixed, the base64url data MUST include padding. This attribute MAY be omitted in deployments where clients are able to retrieve the issuer key using an out-of-band mechanism.

* "max-age", an optional attribute that consists of the number of seconds for which the challenge will be accepted by the origin.
Clients can ignore the challenge if the token-key is invalid or otherwise untrusted.

The header MAY also include the standard "realm" attribute, if desired. Issuance protocols MAY require other attributes.

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

WWW-Authenticate: PrivateToken challenge=abc..., token-key=123...

Upon receipt of this challenge, a client uses the message and keys in the issuance protocol indicated by the token_type. If the TokenChallenge has a token_type the client does not recognize or support, it MUST NOT parse or respond to the challenge. If the TokenChallenge contains a non-empty origin_info field, the client MUST validate that the name of the origin that issued the authentication challenge is included in the list of origin names. Clients MAY have further restrictions and requirements around validating when a challenge is considered acceptable or valid. For example, clients can choose to reject challenges that list origin names for which current connection is not authoritative (according to the TLS certificate).

Caching and pre-fetching of tokens is discussed in Section 2.1.2.

Note that it is possible for the WWW-Authenticate header to include multiple challenges. This allows the origin to indicate support for different token types, issuers, or to include multiple redemption contexts. For example, the WWW-Authenticate header could look like this:

WWW-Authenticate: PrivateToken challenge=abc..., token-key=123..., PrivateToken challenge=def..., token-key=234...

2.1.1. Redemption Context Construction

The TokenChallenge redemption context allows the origin to determine the context in which a given token can be redeemed. This value can be a unique per-request nonce, constructed from 32 freshly generated random bytes. It can also represent state or properties of the client session. Some example properties and methods for constructing the corresponding context are below. This list is not exhaustive.

* Context bound to a given time window: Construct redemption context as SHA256(current time window).

* Context bound to a client location: Construct redemption context as SHA256(client IP address prefix).
* Context bound to a given time window and location: Construct redemption context as SHA256(current time window, client IP address prefix).

An empty redemption context is not bound to any property of the client session. Preventing double spending on tokens requires the origin to keep state associated with the redemption context. The size of this state varies based on the size of the redemption context. For example, double spend state for unique, per-request redemption contexts does only needs to exist within the scope of the request connection or session. In contrast, double spend state for empty redemption contexts must be stored and shared across all requests until token-key expiration or rotation.

2.1.2. Token Caching

Clients can generate multiple tokens from a single TokenChallenge, and cache them for future use. This improves privacy by separating the time of token issuance from the time of token redemption, and also allows clients to avoid any overhead of receiving new tokens via the issuance protocol.

Cached tokens can only be redeemed when they match all of the fields in the TokenChallenge: token_type, issuer_name, redemption_context, and origin_info. Clients ought to store cached tokens based on all of these fields, to avoid trying to redeem a token that does not match. Note that each token has a unique client nonce, which is sent in token redemption (Section 2.2).

If a client fetches a batch of multiple tokens for future use that are bound to a specific redemption context (the redemption_context in the TokenChallenge was not empty), clients SHOULD discard these tokens upon flushing state such as HTTP cookies [COOKIES], or changing networks. Using these tokens in a context that otherwise would not be linkable to the original context could allow the origin to recognize a client.

2.2. Token Redemption

The output of the issuance protocol is a token that corresponds to the origin’s challenge (see Section 2.1). A token is a structure that begins with a two-octet field that indicates a token type, which MUST match the token_type in the TokenChallenge structure.
struct {
    uint16_t token_type;
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[Nid];
    uint8_t authenticator[Nk];
} Token;

The structure fields are defined as follows:

* "token_type" is a 2-octet integer, in network byte order. This value must match the value in the challenge (Section 2.1).

* "nonce" is a 32-octet message containing a client-generated random nonce.

* "challenge_digest" is a 32-octet message containing the hash of the original TokenChallenge, SHA256(TokenChallenge).

* "token_key_id" is an Nid-octet identifier for the the token authentication key. The value of this field is defined by the token_type and corresponding issuance protocol.

* "authenticator" is a Nk-octet authenticator that covers the preceding fields in the token. The value of this field is defined by the token_type and corresponding issuance protocol.

The authenticator value in the Token structure is computed over the token_type, nonce, context, and token_key_id fields.

When used for client authorization, the "PrivateToken" authentication scheme defines one parameter, "token", which contains the base64url-encoded Token struct. Since the length of the Token struct is not fixed, the base64url data MUST include padding. All unknown or unsupported parameters to "PrivateToken" authentication credentials MUST be ignored.

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

Authorization: PrivateToken token=abc...

For token types that support public verifiability, origins verify the token authenticator using the public key of the issuer, and validate that the signed message matches the concatenation of the client nonce and the hash of a valid TokenChallenge. For context-bound tokens, origins store or reconstruct the contexts of previous TokenChallenge structures in order to validate the token. A TokenChallenge MAY be
bound to a specific HTTP session with client, but origins can also accept tokens for valid challenges in new sessions. Origins SHOULD implement some form of double-spend prevention that prevents a token with the same nonce from being redeemed twice. This prevents clients from "replaying" tokens for previous challenges. For context-bound tokens, this double-spend prevention can require no state or minimal state, since the context can be used to verify token uniqueness.

If a client is unable to fetch a token, it MUST react to the challenge as if it could not produce a valid Authorization response.

3. Issuance Protocol Requirements

Clients initiate the issuance protocol using a challenge, a randomly generated nonce, and a public key for the issuer. The issuance protocol itself can be any interactive protocol between client, issuer, or other parties that produces a valid authenticator over the client’s input, subject to the following security requirements.

1. Unconditional input secrecy. The issuance protocol MUST NOT reveal anything about the client’s private input, including the challenge and nonce. The issuance protocol can reveal the issuer public key for the purposes of determining which private key to use in producing the issuance protocol. A result of this property is that the redemption flow is unlinkable from the issuance flow.

2. One-more forgery security. The issuance protocol MUST NOT allow malicious clients to forge tokens without interacting with the issuer directly.

3. Concurrent security. The issuance protocol MUST be safe to run concurrently with arbitrarily many clients.

4. User Interaction

When used in contexts like websites, origins that challenge clients for tokens need to consider how to optimize their interaction model to ensure a good user experience.

Tokens challenges can be performed without explicit user involvement, depending on the issuance protocol. If tokens are scoped to a specific origin, there is no need for per-challenge user interaction. Note that the issuance protocol may separately involve user interaction if the client needs to be newly validated.
If a client cannot use cached tokens to respond to a challenge (either because it has run out of cached tokens or the associated context is unique), the token issuance process can add user-perceivable latency. Origins need 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.

An origin MUST NOT use more than one redemption context value for a given token type and issuer per client request. If an origin issues a large number of challenges with unique contexts, such as more than once for each request, this can indicate that the origin is either not functioning correctly or is trying to attack or overload the client or issuance server. In such cases, a client MUST ignore redundant token challenges for the same request and SHOULD alert the user if possible.

Origins MAY include multiple challenges, where each challenge refers to a different issuer or a different token type, to allow clients to choose a preferred issuer or type.

5. Security Considerations

The security properties of token challenges vary depending on whether the challenge contains a redemption context or not, as well as whether the challenge is per-origin or not. For example, cross-origin tokens with empty contexts can be replayed from one party by another, as shown below.

```
Client          Attacker                  Origin

<-------- Challenge \

<-------- Challenge

Redemption ---->

Redemption -------->
```

Figure 2: Token Architectural Components

Token challenges that include non-empty origin_info bind tokens to one or more specific origins. As described in Section 2.1, clients only accept such challenges from origin names listed in the origin_info string. Even if multiple origins are listed, a token can only be redeemed for an origin if the challenge has an exact match.
for the origin_info. For example, if "a.example.com" issues a challenge with an origin_info string of "a.example.com,b.example.com", a client could redeem a token fetched for this challenge if and only if "b.example.com" also included an origin_info string of "a.example.com,b.example.com". On the other hand, if "b.example.com" had an origin_info string of "b.example.com" or "b.example.com,a.example.com" or "a.example.com,b.example.com,c.example.com", the string would not match and the client would need to use a different token.

Context-bound token challenges require clients to obtain matching tokens when challenged, rather than presenting a token that was obtained from a different context in the past. This can make it more likely that issuance and redemption events will occur at approximately the same time. For example, if a client is challenged for a token with a unique context at time T1 and then subsequently obtains a token at time T2, a colluding issuer and origin can link this to the same client if T2 is unique to the client. This linkability is less feasible as the number of issuance events at time T2 increases. Depending on the "max-age" token challenge attribute, clients MAY try to augment the time between getting challenged then redeeming a token so as to make this sort of linkability more difficult. For more discussion on correlation risks between token issuance and redemption, see [I-D.ietf-privacypass-architecture].

As discussed in Section 2.1, clients SHOULD discard any context-bound tokens upon flushing cookies or changing networks, to prevent an origin using the redemption context state as a cookie to recognize clients.

Applications SHOULD constrain tokens to a single origin unless the use case can accommodate such replay attacks.

All random values in the challenge and token MUST be generated using a cryptographically secure source of randomness.

6. IANA Considerations

6.1. Authentication Scheme

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

Authentication Scheme Name: PrivateToken

Pointer to specification text: Section 2 of this document
6.2. Token Type Registry

The "Token Type" registry lists identifiers for issuance protocols defined for use with the Privacy Pass token authentication scheme. These identifiers are two-byte values, so the maximum possible value is 0xFFFF = 65535.

Template:

* Value: The two-byte identifier for the algorithm
* Name: Name of the issuance protocol
* Publicly Verifiable: A Y/N value indicating if the output tokens are publicly verifiable
* Public Metadata: A Y/N value indicating if the output tokens can contain public metadata.
* Private Metadata: A Y/N value indicating if the output tokens can contain private metadata.
* Nk: The length in bytes of an output authenticator
* Nid: The length of the token key identifier
* Reference: Where this algorithm is defined

The initial contents for this registry are defined in the table below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Publicly Verifiable</th>
<th>Public Metadata</th>
<th>Private Metadata</th>
<th>Nk</th>
<th>Nid</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>(reserved)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/</td>
<td>N/</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 1: Token Types

7. References

7.1. Normative References
7.2. Informative References


Authors' Addresses

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014,
United States of America
Email: tpauly@apple.com
Privacy Pass: Centralization Problem Statement
draft-mcfadden-pp-centralization-problem-03.txt

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), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on September 7, 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 (http://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.
Abstract

This document discusses the problems associated with strict upper bounds on the number of Privacy Pass servers in the proposed Privacy Pass ecosystem. It documents a proposed problem statement.

Table of Contents

1. Introduction...................................................2
2. Key Role Definitions - Terminology.............................3
3. Potential Privacy Concerns....................................4
4. Centralization in Privacy Pass - Problem Statement.............5
   4.1. Architectural Problems....................................5
   4.2. Engineering Problems......................................6
   4.3. Practical Problems........................................6
5. Problem Statement and Potential for Mitigations................7
   5.1. Problem Statement.........................................7
   5.2. Potential Mitigations.....................................7
   5.3. Redemption Contexts as a Mitigation.......................8
   5.4. Implementation Base as a Mitigation.......................8
6. Security Considerations........................................9
7. IANA Considerations............................................9
8. References.....................................................9
   8.1. Normative References......................................9
   8.2. Informative References....................................9
9. Acknowledgments................................................9

1. Introduction

The Privacy Pass protocol provides a set of cross-domain authorization tokens that protect the client’s anonymity in message exchanges with a server. This allows clients to communicate an attestation of a previously authenticated server action, without having to reauthenticate manually. The tokens retain anonymity in the sense that the act of revealing them cannot be linked back to the session where they were initially issued.

The protocol itself is defined in [ID.ietf-privacypass-protocol-02] and the architectural framework is in [ID.ietf-privacypass-architecture-02].

The architecture document provides a concise representation of the roles in the Privacy Pass ecosystem, which is repeated for reference here:
Figure 1: Privacy Pass Roles and Flows

An important feature of the Privacy Pass Architecture is the concept of the anonymity set of each individual client. The Privacy Pass ecosystem has a set of issuers which issue tokens to clients which can then be redeemed in the Privacy Pass redemption process for authentication.

Trust is an important component in Privacy Pass. The issuers have to publish their public keys and details of the ciphersuite they are using. It is necessary to publish these in a globally consistent, tamper-proof data structure. Clients that use the same registry of server information need to coordinate in some way to validate that they have the same view of the registry and its data.

Having a large number of Issuers results in the possibility that an Origin could learn further information about a Client. The architecture draft [ID.ietf-privacypass-architecture-02] suggests that the mitigation for this should be an upper limit on the number of Issuers that are allowed in the Privacy Pass ecosystem. The motivation for limiting the number of Issuers is that there is a correlation between larger numbers of servers and dilution of privacy.

2. Key Role Definitions - 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.

A set of words that are in common use have very specific meaning in the context of Privacy Pass. This document relies entirely on the draft architecture [ID.ietf-privacypass-architecture-02] for the definition of the following:

Client - An entity that seeks authorization to an Origin.

Origin - An entity that challenges Clients for tokens.

Issuer - An entity that issues tokens to Clients for properties attested to by the Attester.

Attester - An entity that attests to properties of Client for the purposes of token issuance.

3. Potential Privacy Concerns

When a Client redeems a token in Privacy Pass, there is very little information in the token itself other than the key that was used to sign the token. A key feature of the protocol is that any Client can only remain private relative to the entire space of users using the protocol.

The architecture document, [ID.ietf-privacypass-architecture-02], provides an example where, if there are 32 Issuers, then Origins learn 32 bits of information about the Client. In certain circumstances, having that much information about the Client can lead to the client being uniquely identified and the goals of Privacy Pass thwarted. As a result, the architecture document supplies the following mitigation:

"In cases where clients can hold tokens for all servers at any given time, a strict bound SHOULD be applied to the active number of Issuers in the ecosystem. [ID.ietf-privacypass-architecture-02]."

Putting restrictions on the number of redemption tokens at the client is considered. However, establishing control of the Client, and the number of tokens it has, is far more difficult than restricting the number of active Issuers.
4. Centralization in Privacy Pass - Problem Statement

For Privacy Pass to succeed Clients must be able to acquire tokens that they can later redeem with greater privacy and anonymity. This document does not discuss the goals of privacy or anonymity. Instead, it identifies a problem related to the upper bound in number of servers that affects the Privacy Pass ecosystem.

For the purposes of this draft, "server centralization" is the strict limit or upper bound in the number of Issuers available from which a Client can acquire a token for later redemption.

The architecture draft specifies an upper limit of four for this upper bound.

The problem statement for Privacy pass can be summarized: an upper bound to available Privacy Pass Issuers creates architectural, engineering and practical problems for the deployment of the protocol. Any successful deployment of Privacy Pass must find mitigations for these problems.

4.1. Architectural Problems

Centralization is a problem space that has been exhaustively explored by others; not least of which in the IETF itself. The current draft on avoiding Internet centralization [I-D.nottingham-avoiding-internet-centralization-02] provides a useful guide to understand why centralization is undesirable. The now expired IAB draft, [I-D.arkko-arch-infrastructure-centralisation-00], discussed six separate issues related to centralization and several of them appear to apply to Privacy Pass.

Having a very limited number of servers available creates an architectural strain on avoiding single points of failure. While the Privacy Pass architecture document does specify up to four servers, this is a very small number for, potentially, billions of possible users. And this assumes that the protocol is only used in "human-to-server" applications and not in situations where the client is not a human but some other device - either acting on behalf of human or autonomously. Strict limitations on the number of servers poses the question of how the Privacy Pass architecture can scale in the presence of a large user base.

The Privacy Pass architecture, by limiting the number of servers, also concentrates information and potentially limits the ability for other competing providers of the token generating services. By concentrating the information in a small number of servers, a
problem appears when there are machine learning opportunities to collect and process data about clients requesting tokens.

A side effect of limiting the number of servers is that a significant amount of information ends up being in the control of a small number of entities. A client may trust a Privacy Pass server as send it information about itself in order to request tokens. However, the protocol itself can make no guarantee about the data handling practices of the server operator. Situations outside the control of the protocol may make it so there are pressures to misuse the data concentrated at the small number of servers.

4.2. Engineering Problems

In the event that a very limited number of servers can be provided while still supporting the goals of the protocol, there is clearly a global scaling problem that needs to be solved. Each server must publish a global, consistent and protected view of its published key and the cryptosystem in use. Without access to that view, the system appears to have no failure mode.

With a small number of servers, the ecosystem would likely be dominated by a few providers. With a dominant position in the market these Privacy Pass server operators would have a significant impact on default connectivity parameters in operating systems and browsers. As a result, a change to the way the access mechanism works for a variety of applications would have broad impacts to a wide variety of users. The relationship between engineering and how centralization affects a broad community of users and uses DNS over HTTP as an example. Another draft [I-D.draft-lazanski-consolidation-03] discusses the technical, economic and security implications of consolidation.

4.3. Practical Problems

Limits to the number of Issuers also results in practical problems outside the protocol. In the event that a small number of Issuers appear in the Privacy Pass ecosystem, and a large number of clients enter into trust relationships with those operators, what happens when those operators are acquired by other organizations that have different data handling and privacy policies than the original operator? The idea of Issuer churn is discussed in the architecture document, but is limited to discussing ensuring that trusted registries record which Issuers are active in the ecosystem.

With the requirement for a small number of Issuers, the architecture also doesn’t consider the possibility that an organization or
government could require Privacy Pass and the use of a particular set of Issuers. Such a requirement could potentially turn the goals of Privacy Pass against itself.

5. Problem Statement and Potential for Mitigations

5.1. Problem Statement

An upper bound to available Privacy Pass Issuers creates architectural, engineering and practical problems for the deployment of the protocol. Any successful deployment of Privacy Pass must find mitigations for these problems.

5.2. Potential Mitigations

The motivation for having an upper bound to available Privacy Pass Issuers is to limit the amount of information that could be gathered, because a client could be forced to redeem tokens for any issuing key. A large number of keys, means a greater about of information exposed.

One alternative to limiting the number of Issuers is to constrain the Clients so that they only possess redemption tokens for a small number of Issuers. This potential mitigation doesn’t address how the tokens might be cached, but it does discuss how the limitation might be implemented. However, there is much engineering experience to suggest that making a limitation work in a very large number of Clients is a much greater engineering and deployment problem than placing the restriction in the Issuer ecosystem.

In addition, limiting the number of Issuers increases the impact of failure in the event that there is insufficient redundancy. It is a situation that has impacted other protocols such as OAuth: when a critical provider of services (such as an Issuer) is unavailable or compromised, the impact of the failure affects services beyond the provider of services. In the case of Privacy Pass, failure at the Issuer would require the client to use another trusted Issuer. However, if the Origin trusted a limited number of Issuers, the Origin’s service could be rendered unavailable.

If the motivation for restricting the number of Issuers is essential for the success of Privacy Pass – and the mitigations at either the Issuer or Client are difficult to overcome – it is hard to understand where the mitigations for the problem statement will emerge.
5.3. Redemption Contexts as a Mitigation

Contexts are groupings of resources that have shared anonymity and privacy properties. The current architecture statement has a single, global context for redemption. It is this feature that causes the problem outlined in section 4.1 above: with N issuers in the global ecosystem, there are \(2^N\) possible anonymity sets. Adding additional metadata bits increases the number of anonymity sets.

The global redemption context results in a requirement of less than ten total issuers in order to maintain anonymity sets of 5,000.

One possible mitigation is to limit redemptions to a specific, shared context. Such an approach could limit the information available - and the potential for leakage - to a specific context. This type of solution would rely, in part, on strong security/privacy boundaries between contexts. While information about redemptions in one context wouldn’t affect information in another context, this solution depends upon there being no leakage of information between those contexts.

While this potential mitigation is suggested as a possible remediation in the Privacy Pass architecture, it is unclear whether it should be a part of the protocol design or it should be left to the application layer to implement. If left to the application layer, there is potential for the anonymity sets to be very small and not meet the privacy goals of the protocol.

What is not clear is how the consolidation considerations are affected by the development of a "symmetric mode" for Privacy Pass. The symmetric mode provides optional metadata but will not enable the use of public verifiability. The goal of this change is to remain consistent with work in the W3C. The mode will use the POPRF algorithm which does not change the architectural characteristics considered in this paper.

5.4. Implementation Base as a Mitigation

Protocol parameterization in Privacy Pass provides a guide to both architecture and implementation. The calculation of the maximum number of Issuers is based on \([0.5 \times (U/2^{2I})]\) where U is the user base.

The architecture document notes that with an implementation base of 5 million users the maximum number of allowed Issuers is about 4. Using the parameterization choices in the architecture draft,
increasing the size of the installed base to 50 million users only increases the maximum number of allowed Issuers to about 6.

6. Security Considerations

This document is all about security considerations for Privacy Pass. In particular, it addresses the very specific problem associated with centralization of Privacy Pass servers.

7. IANA Considerations

This memo contains no instructions or requests for IANA. The authors continue to appreciate the efforts of IANA staff in support of the IETF.

8. References

8.1. Normative References


8.2. Informative References


9. Acknowledgments

This document was prepared using 2-Word-v2.0.template.dot.
Authors’ Addresses

Mark McFadden
Internet policy advisors, ltd
Chepstow, Wales, United Kingdom

Email: mark@internetpolicyadvisors.com
Rate-Limited Token Issuance Protocol
draft-privacypass-rate-limit-tokens-02

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/tfpauly/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 ...................................................... 3
   1.1. Motivation .................................................. 3
   1.2. Properties and Requirements ............................... 4
2. Terminology ...................................................... 6
3. Configuration .................................................... 8
4. Token Challenge Requirements ................................... 9
5. Issuance Protocol ................................................ 10
   5.1. State Requirements ......................................... 10
      5.1.1. Client State ........................................... 11
      5.1.2. Attester State ......................................... 12
      5.1.3. Issuer State ........................................... 12
   5.2. Issuance HTTP Headers ...................................... 13
   5.3. Client-to-Attester Request ................................. 14
   5.4. Attester-to-Issuer Request ................................. 17
   5.5. Issuer-to-Attester Response ............................... 18
   5.6. Attester-to-Client Response ............................... 19
6. Encrypting Origin Token Requests and Responses ................. 20
   6.1. Client to Issuer Encapsulation ............................ 20
   6.2. Issuer to Client Encapsulation ............................ 22
7. Anonymous Issuer Origin ID Computation ........................ 23
   7.1. Client Behavior ........................................... 25
      7.1.1. Request Key .......................................... 25
      7.1.2. Request Signature .................................... 25
   7.2. Attester Behavior (Client Request Validation) ............. 26
   7.3. Issuer Behavior ........................................... 26
   7.4. Attester Behavior (Index Computation) ..................... 27
8. Security Considerations ........................................ 28
   8.1. Channel Security .......................................... 28
   8.2. Token Request Unlinkability and Unforgeability .......... 28
   8.3. Information Disclosure ................................... 29
9. Privacy Considerations ......................................... 30
   9.1. Client Token State and Origin Tracking .................... 30
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].

```plaintext
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.
Table 1

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>issuer-policy-window</td>
<td>Issuer Policy Window as a JSON number</td>
</tr>
<tr>
<td>issuer-request-uri</td>
<td>Issuer Request URI resource URL as a JSON string</td>
</tr>
<tr>
<td>issuer-encap-key-uri</td>
<td>Issuer Encapsulation Key URI resource URL as a JSON string</td>
</tr>
</tbody>
</table>

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 Attesting 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:

\[
\text{blind\_sig} = \text{rsabssa\_blind\_sign}(\text{skP}, \text{InnerTokenRequest.blinded\_msg})
\]

\text{skP is the private key corresponding to Token Key, known only to the Issuer. The Issuer then encrypts \text{blind\_sig} to the Client as described in Section 6.2, yielding \text{encrypted\_token\_response}.}

The Issuer generates an HTTP response with status code 200 whose body consists of \text{blind\_sig}, with the content type set as "message/token-response", the \text{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:

\[\text{authenticator = rsabssa_finalize}(pkI, \text{token_input}, \text{blind_sig}, \text{blind_inv})\]

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

\[
\text{struct } \\
\text{ \quad \{ } \\
\text{ \quad \quad uint16_t token_type = 0x0003; } \\
\text{ \quad \quad uint8_t nonce[32]; } \\
\text{ \quad \quad uint8_t context[32]; } \\
\text{ \quad \quad uint8_t token_key_id[Nid]; } \\
\text{ \quad \quad uint8_t authenticator[Nk]} \\
\text{ \quad \}} \text{ Token; }
\]
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 an 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
   \( N = 31 - ((L - 1) \mod 32) \)
   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:

```python
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:

```python
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(Nn, Nk), where Nn and Nk are the length of AEAD key and nonce associated with context.

2. Generate a random value of length max(Nn, Nk) 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 Nk - 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 Nn - 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.

<table>
<thead>
<tr>
<th>Client</th>
<th>Attester</th>
<th>Issuer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(request, signature)</td>
<td>--------------------</td>
<td>(request, signature)</td>
</tr>
<tr>
<td>#</td>
<td>(response)</td>
<td>#</td>
</tr>
</tbody>
</table>
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:

\[
\begin{align*}
\text{sk\_blind} &= \text{BKS-BlindKeyGen}() \\
\text{blinded\_key} &= \text{BKS-BlindPublicKey}(\text{pk\_sign}, \text{sk\_blind}) \\
\text{request\_key} &= \text{BKS-SerializePublicKey}(\text{blinded\_key}) \\
\text{request\_blind} &= \text{BKS-SerializePrivateKey}(\text{sk\_blind})
\end{align*}
\]

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:

\[
\begin{align*}
\text{sk\_blind} &= \text{BKS-BlindKeyGen}() \\
\text{blinded\_key} &= \text{BKS-BlindPublicKey}(\text{pk}, \text{sk\_blind}) \\
\text{request\_key} &= \text{BKS-SerializePublicKey}(\text{blinded\_key}) \\
\text{request\_blind} &= \text{BKS-SerializePrivateKey}(\text{sk\_blind})
\end{align*}
\]
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. Attestor Behavior (Client Request Validation)

Given a TokenRequest request containing request_key, request_signature, and request_blind, as well as Client Key pk_blind, Attestors 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

color = parse(request[.len(request)-Nsig]) // this matches context computed during signing
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:

```python
blind_key = BKS-DeserializePublicKey(request_key)
context = parse(request[..len(request)-Nsig])  // this matches context computed during signing
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:

```python
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, the 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 an 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:
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. 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.
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>.

<table>
<thead>
<tr>
<th>Header Field Name</th>
<th>Protocol</th>
<th>Status</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sec-Token-Origin</td>
<td>http</td>
<td>std</td>
<td>This document</td>
</tr>
<tr>
<td>Sec-Token-Client</td>
<td>http</td>
<td>std</td>
<td>This document</td>
</tr>
<tr>
<td>Sec-Token-Request-Blind</td>
<td>http</td>
<td>std</td>
<td>This document</td>
</tr>
<tr>
<td>Sec-Token-Request-Key</td>
<td>http</td>
<td>std</td>
<td>This document</td>
</tr>
<tr>
<td>Sec-Token-Limit</td>
<td>http</td>
<td>std</td>
<td>This document</td>
</tr>
</tbody>
</table>

Figure 1: Registered HTTP Header

12. References

12.1. Normative References

[AUTHSCHEME]


12.2. Informative References

[BASIC-ISSUANCE]
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**: 

d2653816496f400baec656f213f1345092f4406af4f2a63e164956c43c4d240ca

**issuer_encap_key**: 010020d7b6a2c10e75c4239feb9897e8d23f3f3c377d78e790361153167736a24a9c5400010001

**token_key_id**: 125

**token_type**: 3

**token_key_id**: 125

**blinded_msg**: 89da551a48270b053e53c9eb741badf89e43cb7e66366bb936e11fb2aa0d30866986a790378bb9fc6a77cf5c32b7b7584d448fafa4ced3be650e354b3136428a52ec0b27c4103c5855c2b9b4f521ad0713c800d7e6925b6c321a6f58b31d13335f468cf509b746a1679b23862d277d808706c3f8b41b21d794af8f8662f71244e81994441b19be084ec5c159b0adab433bc808d90ea2caadbdf4216e1b07155be66a048686e383c2ae5157ab80025bb4849d98ebc83cd05d045c1167cb74f4451d8f85695babb604418385464f21f9a815fb850ed83f16a966130427e5637816501f7a79c0010e06adeeba55781ceb50f56eae1520eb06f33ef880dca7ab12d

**request_key**: 0161d905e43f7515cb61f863b60e5896a9e4a17dbe238e752a144e64a5412e244f0b1f75e01083e185cac023d33cb20

**issuer_encap_key_id**: dd26de3011f187364323d3229a7a0e9defe0f9fe43f6a7c4a3ea6b16f77837

**encrypted_token_request**: 82ef7c0468506cbabc27d068a51c7ead2cbaf600b76a15e4d9df99a0da676a5a073fccc8f5ac77b25064d737903734e1b186977cface31ecbb611978c73c9ef38c9a0e8ae46881624fa6d454523a0a91d22b02b02b289190493deebd66a912a0b3391b203e92e0a681f0a10c2a2d59b668daf1e5219ed16227d707fa0e8e2918bbd57ab38b3584564ce9b6538ba823e10cfed4231a2a4f64a67285a1b9bf648e25f3eb644c88d435252daba64e6bcbaba0de3ac245e0432be6b019499427fa7e043bb7759f9e8ca5afeafbf204889d54618408a6001f6f8b276f6828c46f4fe1318e9775ec72ee47593d7373951d81952440d33756d78ca4e4b2d8218905df35a6a6c46c535211eda39da277260cb804ab7c00c6840a745e8150a6ee4899e72b6a51382f87768c05a15e891a2be07047960f0f987978d7b8b97e47ecaf90a44996d724dd3720e308abbff04f672b5a4db5732919866be19b0603f52aacb6bfa081c151c758f3092a89fc6ef591934ff4bc860896c57f83a31b237dd1b803516c

**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: f6e6a0c9de38663ca539ff2e6a04e4fca11dc569794dc405e2d17439d6ce4f67abb2b81a852e0db993b6a0452eb60d6
pk_sign: 032db7483c710673e6999a5fb2a2c6eac1d891f89bbf589d85ff168d182ad51605c4369280efabb7692f661162e683f03c
sk_origin: 85de5fbbd787da5093da0adb240eba0cc6ea90d72032fc4b6925dd7d0ab1d
request Blind: 0698a149fb9d16bcb0a856062f74f9191e82b35e91224a57abce60f5b79f03a669c6b5e093d57e647865f9fd4305b5a9
request_key: 0287b0ce6b957112638c6126af96d400bd5a9d0bf062ade15ab78944606c209470ced7086d3c418dd32bf9245fd42678
index_key: 03188bec3dc02d2382b5251b6b4fd729d0472bbddf008c5e93b7c12270d9f57d0e11c861c13be53822a1cebb604946066
anon_issuer_origin_id: 9b0f980e5c1142fddb4401e5cd2107a87d22b73753b0d5dc93f9a8f5ed2e7db78163c6a93cc41ae8158d562381c51ee

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