Privacy Pass
Internet-Draft
Intended status: Informational
Expires: 14 November 2026

S. Hendrickson
Google
C. A. Wood

13 May 2026

Privacy Pass Issuance Protocols with Public Metadata
draft-ietf-privacypass-public-metadata-issuance-03

Abstract

This document specifies Privacy Pass issuance protocols that encode public information visible to the Client, Attester, Issuer, and Origin into each token.

About This Document

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

The latest revision of this draft can be found at https://ietf-wg-privacypass.github.io/draft-ietf-privacypass-public-metadata-issuance/draft-ietf-privacypass-public-metadata-issuance.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-privacypass-public-metadata-issuance/.

Discussion of this document takes place on the Privacy Pass Working Group mailing list (mailto:privacy-pass@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/privacy-pass/. Subscribe at https://www.ietf.org/mailman/listinfo/privacy-pass/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-privacypass/draft-ietf-privacypass-public-metadata-issuance.

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/.

Copyright Notice

Table of Contents

1. Introduction

The basic Privacy Pass issuance protocols as specified in [BASIC-PROTOCOL] and resulting tokens convey only a single bit of information: whether or not the token is valid. However, it is possible for tokens to be issued with additional information agreed upon by Client, Attester, and Issuer during issuance. This information, sometimes referred to as public metadata, allows Privacy Pass applications to encode deployment-specific information that is necessary for their use case.

This document specifies two Privacy Pass issuance protocols that encode public information visible to the Client, Attester, Issuer, and Origin. One is based on the partially-oblivious PRF construction from [POPRF], and the other is based on the partially-blind RSA signature scheme from [PBRSA].

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.

3. Notation

The following terms are used throughout this document to describe the protocol operations in this document:

* len(s): the length of a byte string, in bytes.

* concat(x0, ..., xN): Concatenation of byte strings. For example, concat(0x01, 0x0203, 0x040506) = 0x010203040506

* int_to_bytes: Convert a non-negative integer to a byte string. int_to_bytes is implemented as I2OSP as described in Section 4.1 of [RFC8017]. Note that these functions operate on byte strings in big-endian byte order.

4. Motivation

Public metadata enables Privacy Pass deployments that share information between Clients, Attesters, Issuers and Origins. In the basic Privacy Pass issuance protocols (types 0x0001 and 0x0002), the only information available to all parties is the choice of Issuer, expressed through the TokenChallenge. If one wants to differentiate bits of information at the origin, many PrivateToken challenges must

be sent, one for each Issuer that attests to the bit required.

For example, if a deployment was built that attested to an app's published state in an app store, it requires 1 bit {published, not_published} and can be built with a single Issuer. An app version attester would require one Issuer for each app version and one TokenChallenge per Issuer.

Taken further, the limitation of one bit of information in each Privacy Pass token means that a distinct Issuer and Issuer public key is needed for each unique value one wants to express with a token. This many-key metadata deployment should provide metadata visible to all parties in the same way as the [PBRSA] proposal outlined in this document. However, it comes with practical reliability and scalability tradeoffs. In particular, many simultaneous deployed keys could be difficult to scale. Some HSM implementations have fixed per-key costs, slow key generation, and minimum key lifetimes. Quick key rotation creates reliability risk to the system, as a pause or slowdown in key rotation could cause the system to run out of active signing or verification keys. Issuance protocols that support public metadata mitigate these tradeoffs by allowing deployments to change metadata values without publishing new keys.

5. Issuance Protocol for Privately Verifiable Tokens

This section describes a variant of the issuance protocol in Section 5 of [BASIC-PROTOCOL] that supports public metadata based on the partially oblivious PRF (POPRF) from [POPRF]. Issuers provide a Private and Public Key, denoted skI and pkI respectively, used to produce tokens as input to the protocol. See Section 5.5 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

* Issuer Request URI: A URI to which token request messages are sent. This can be a URL derived from the "issuer-request-uri" value in the Issuer's directory resource, or it can be another Client-configured URL. The value of this parameter depends on the Client configuration and deployment model. For example, in the 'Split Origin, Attester, Issuer' deployment model, the Issuer Request URI might correspond to the Client's configured Attester, and the Attester is configured to relay requests to the Issuer.

* Issuer name: An identifier for the Issuer. This is typically a host name that can be used to construct HTTP requests to the Issuer.

* Issuer Public Key: pkI, with a key identifier token_key_id computed as described in Section 6.5.

* Challenge value: challenge, an opaque byte string. For example, this might be provided by the redemption protocol in [AUTHSCHEME].

* Extensions: extensions, an Extensions structure as defined in [TOKEN-EXTENSION].

Given this configuration and these inputs, the two messages exchanged in this protocol are described below. This section uses notation described in [POPRF], Section 4, including SerializeElement and DeserializeElement, SerializeScalar and DeserializeScalar, and DeriveKeyPair.

The constants Ne and Ns are as defined in [POPRF], Section 4 for OPRF(P-384, SHA-384). The constant Nk, which is also equal to Nh as defined in [POPRF], Section 4, is defined in Section 8.

5.1. Client-to-Issuer Request

The Client first creates a context as follows:

```
client_context = SetupPOPRFClient("P384-SHA384", pkI)
```

Here, "P384-SHA384" is the identifier corresponding to the OPRF(P-384, SHA-384) ciphersuite in [POPRF]. SetupPOPRFClient is defined in [POPRF], Section 3.2.

The Client then creates an issuance request message for a random value nonce with the input challenge and Issuer key identifier as described below:

```
nonce = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0xDA7B, // Token type field is 2 bytes long
                     nonce,
                     challenge_digest,
                     token_key_id)
blind, blinded_element, tweaked_key = client_context.Blind(token_input, extensions, pkI)
```

The Blind function is defined in [POPRF], Section 3.3.3. If the Blind function fails, the Client aborts the protocol. The Client stores the nonce, challenge_digest, and tweaked_key values locally for use when finalizing the issuance protocol to produce a token (as described in Section 5.3).

The Client then creates an ExtendedTokenRequest structured as follows:

```
struct {
  TokenRequest request;
  Extensions extensions;
} ExtendedTokenRequest;
```

The contents of ExtendedTokenRequest.request are as defined in Section 5 of [BASIC-PROTOCOL]. The contents of ExtendedTokenRequest.extensions match the Client's configured extensions value.

The Client then generates an HTTP POST request to send to the Issuer Request URI, with the ExtendedTokenRequest as the content. The media type for this request is "application/private-token-request". An example request is shown below:

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

<Bytes containing the ExtendedTokenRequest>
```

5.2. Issuer-to-Client Response

* The ExtendedTokenRequest.request contains a supported token_type.

* The ExtendedTokenRequest.request.truncated_token_key_id corresponds to the truncated key ID of an Issuer Public Key.

* The ExtendedTokenRequest.request.blinded_msg is of the correct size.

* The ExtendedTokenRequest.extensions value is permitted by the Issuer's policy.

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the Client, which will forward the error to the client. Otherwise, if the Issuer is willing to produce a token token to the Client for the provided extensions, the Issuer then tries to deserialize ExtendedTokenRequest.request.blinded_msg using DeserializeElement from Section 2.1 of [POPRF], yielding

blinded_element. If this fails, the Issuer MUST return an HTTP 400 error to the client. Otherwise, the Issuer completes the issuance flow by computing a blinded response as follows:

```
server_context = SetupPOPRFServer("P384-SHA384", skI, pkI)
evaluate_element, proof =
  server_context.BlindEvaluate(skI, blinded_element, ExtendedTokenRequest.extensions)
```

SetupPOPRFServer is defined in [POPRF], Section 3.2 and BlindEvaluate is defined in [POPRF], Section 3.3.3. The Issuer then creates a TokenResponse structured as follows:

```
struct {
   uint8_t evaluate_msg[Ne];
   uint8_t evaluate_proof[Ns+Ns];
} TokenResponse;
```

The structure fields are defined as follows:

* "evaluate_msg" is the Ne-octet evaluated message, computed as SerializeElement(evaluate_element).

* "evaluate_proof" is the (Ns+Ns)-octet serialized proof, which is a pair of Scalar values, computed as concat(SerializeScalar(proof[0]), SerializeScalar(proof[1])).

The Issuer generates an HTTP response with status code 200 whose content consists of TokenResponse, with the content type set as "application/private-token-response".

```
:status = 200
content-type = application/private-token-response
content-length = <Length of TokenResponse>

<Bytes containing the TokenResponse>
```

5.3. Finalization

Upon receipt, the Client handles the response and, if successful, deserializes the content values TokenResponse.evaluate_msg and TokenResponse.evaluate_proof, yielding evaluated_element and proof. If deserialization of either value fails, the Client aborts the protocol. Otherwise, the Client processes the response as follows:

```
 authenticator = client_context.Finalize(token_input, blind,
                                         evaluated_element,
                                         blinded_element,
                                         proof, extensions, tweaked_key)
```

The Finalize function is defined in [POPRF], Section 3.3.3. If this succeeds, the Client then constructs a Token as follows:

```
struct {
  uint16_t token_type = 0xDA7B; /* Type POPRF(P-384, SHA-384) */
  uint8_t nonce[32];
  uint8_t challenge_digest[32];
  uint8_t token_key_id[32];
  uint8_t authenticator[Nk];
} Token;
```

The Token.nonce value is that which was sampled in Section 5.1. If the Finalize function fails, the Client aborts the protocol.

The Client will send this Token to Origins for redemption in the "token" HTTP authentication parameter as specified in Section 2.2 of [AUTHSCHEME]. The Client also supplies its extensions value as an additional authentication parameter as specified in [TOKEN-EXTENSION].

5.4. Token Verification

Verifying a Token requires creating a POPRF context using the Issuer Private Key and Public Key, evaluating the token contents with the corresponding extensions, and comparing the result against the token authenticator value:

```
server_context = SetupPOPRFServer("P384-SHA384", skI)
token_authenticator_input =
  concat(Token.token_type,
         Token.nonce,
         Token.challenge_digest,
         Token.token_key_id)
token_authenticator =
  server_context.Evaluate(skI, token_authenticator_input, extensions)
valid = (token_authenticator == Token.authenticator)
```

5.5. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted skI and pkI, respectively, used to produce tokens. These keys MUST NOT be reused in other protocols. A RECOMMENDED method for generating key pairs is as follows:

```
seed = random(Ns)
(skI, pkI) = DeriveKeyPair(seed, "PrivacyPass-TypeDA7B")
```

The DeriveKeyPair function is defined in [POPRF], Section 3.3.1. The key identifier for a public key pkI, denoted token_key_id, is computed as follows:

token_key_id = SHA256(SerializeElement(pkI))

Since Clients truncate token_key_id in each TokenRequest, Issuers should ensure that the truncated form of new key IDs do not collide with other truncated key IDs in rotation.

6. Issuance Protocol for Publicly Verifiable Tokens

This section describes a variant of the issuance protocol in Section 6 of [BASIC-PROTOCOL] for producing publicly verifiable tokens including public metadata using cryptography specified in [PBRSA]. In particular, this variant of the issuance protocol works for the RSAPBSSA-SHA384-PSSZERO-Deterministic or RSAPBSSA-SHA384-PSS-Deterministic variant of the blind RSA protocol variants described in Section 6 of [PBRSA].

The public metadata issuance protocol differs from the protocol in Section 6 of [BASIC-PROTOCOL] in that the issuance and redemption protocols carry metadata provided by the Client and visible to the Attester, Issuer, and Origin. This means Clients can set arbitrary metadata when requesting a token, but specific values of metadata may be rejected by any of Attester, Issuer, or Origin. Similar to a token nonce, metadata is cryptographically bound to a token and cannot be altered.

Clients provide the following as input to the issuance protocol:

* Issuer Request URI: A URI to which token request messages are sent. This can be a URL derived from the "issuer-request-uri" value in the Issuer’s directory resource, or it can be another Client-configured URL. The value of this parameter depends on the Client configuration and deployment model. For example, in the ’Split Origin, Attester, Issuer’ deployment model, the Issuer Request URI might be correspond to the Client’s configured Attester, and the Attester is configured to relay requests to the Issuer.

* Issuer name: An identifier for the Issuer. This is typically a host name that can be used to construct HTTP requests to the Issuer.

* Issuer Public Key: pkI, with a key identifier token_key_id computed as described in Section 6.5.

* Challenge value: challenge, an opaque byte string. For example, this might be provided by the redemption protocol in [AUTHSCHEME].

* Extensions: extensions, an Extensions structure as defined in [TOKEN-EXTENSION].

Given this configuration and these inputs, the two messages exchanged in this protocol are described below. The constant Nk is defined as 256 for token type 0xDA7A.

6.1. Client-to-Issuer Request

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

```
nonce = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0xDA7A, // Token type field is 2 bytes long
                     nonce,
                     challenge_digest,
                     token_key_id)
blinded_msg, blind_inv = Blind(pkI, PrepareIdentity(token_input), extensions)
```

Where PrepareIdentity is defined in Section 6 of [PBRSA] and Blind is defined in Section 4.2 of [PBRSA]

The Client stores the nonce, challenge_digest, and extensions values locally for use when finalizing the issuance protocol to produce a token (as described in Section 6.3).

The Client then creates an ExtendedTokenRequest structured as follows:

```
struct {
  TokenRequest request;
  Extensions extensions;
} ExtendedTokenRequest;
```

The contents of ExtendedTokenRequest.request are as defined in Section 6 of [BASIC-PROTOCOL]. The contents of ExtendedTokenRequest.extensions match the Client’s configured extensions value.

The Client then generates an HTTP POST request to send to the Issuer Request URI, with the ExtendedTokenRequest as the content. The media type for this request is "application/private-token-request". An example request is shown below:

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

<Bytes containing the ExtendedTokenRequest>
```

6.2. Issuer-to-Client Response

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

* The ExtendedTokenRequest.request contains a supported token_type.

* The ExtendedTokenRequest.request.truncated_token_key_id corresponds to the truncated key ID of an Issuer Public Key.

* The ExtendedTokenRequest.request.blinded_msg is of the correct size.

* The ExtendedTokenRequest.extensions value is permitted by the Issuer’s policy.

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the Client, which will forward the error to the client. Otherwise, if the Issuer is willing to produce a token token to the Client for the provided extensions, the Issuer completes the issuance flow by computing a blinded response as follows:

```
blind_sig = BlindSign(skI, ExtendedTokenRequest.request.blinded_msg, ExtendedTokenRequest
.extensions)
```

Where BlindSign is defined in Section 4.3 of [PBRSA].

The result is encoded and transmitted to the client in a TokenResponse structure as defined in Section 6 of [BASIC-PROTOCOL].

The Issuer generates an HTTP response with status code 200 whose content consists of TokenResponse, with the content type set as "application/private-token-response".

```
   :status = 200
   content-type = application/private-token-response
   content-length = <Length of TokenResponse>

   <Bytes containing the TokenResponse>
```

6.3. Finalization

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

```
  authenticator = Finalize(pkI, nonce, extensions, blind_sig, blind_inv)
```

Where Finalize function is defined in Section 4.4 of [PBRSA].

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

```
struct {
  uint16_t token_type = 0xDA7A; /* Type Partially Blind RSA (2048-bit) */
  uint8_t nonce[32];
  uint8_t challenge_digest[32];
  uint8_t token_key_id[32];
  uint8_t authenticator[Nk];
} Token;
```

The Token.nonce value is that which was sampled in Section 5.1 of [BASIC-PROTOCOL]. If the Finalize function fails, the Client aborts the protocol.

The Client will send this Token to Origins for redemption in the "token" HTTP authentication parameter as specified in Section 2.2 of [AUTHSCHEME]. The Client also supplies its extensions value as an additional authentication parameter as specified in [TOKEN-EXTENSION].

6.4. Token Verification

Verifying a Token requires checking that Token.authenticator is a valid signature over the remainder of the token input with respect to the corresponding Extensions value extensions using the Augmented Issuer Public Key. This involves invoking the verification procedure described in Section 4.5 of [PBRSA] using the following token_input value as the input message, extensions as the input info (metadata), the Issuer Public Key as the input public key, and the token authenticator (Token.authenticator) as the signature.

```
token_input = concat(0xDA7A, // Token type field is 2 bytes long
                     Token.nonce,
                     Token.challenge_digest,
                     Token.token_key_id)
```

6.5. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted skI and pkI, respectively, used to produce tokens. Each key pair SHALL be generated as as specified in FIPS 186-4 [DSS], where the RSA modulus is 2048 bits in length. These key pairs MUST NOT be reused in other protocols. Each key pair MUST comply with all requirements as specified in Section 5.2 of [PBRSA].

The key identifier for a keypair (skI, pkI), denoted token_key_id, is computed as SHA256(encoded_key), where encoded_key is a DER-encoded SubjectPublicKeyInfo (SPKI) object carrying pkI. The SPKI object MUST use the RSASSA-PSS OID [RFC5756], which specifies the hash algorithm and salt size. The salt size MUST match the output size of the hash function associated with the public key and token type.

Since Clients truncate token_key_id in each TokenRequest, Issuers should ensure that the truncated form of new key IDs do not collide with other truncated key IDs in rotation.

7. Security Considerations

By design, public metadata is known to both Client and Issuer. The mechanism by which public metadata is made available to Client and Issuer is out of scope for this document. The privacy considerations in [ARCHITECTURE] offer a guide for determining what type of metadata is appropriate to include, and in what circumstances.

Each metadata use case requires careful consideration to ensure it does not regress the intended privacy properties of Privacy Pass. In general, however, metadata is meant primarily for simplfiying Privacy Pass deployments, and such simplifications require analysis so as to not invalidate Client privacy. As an example of metadata that would not regress privacy, consider the use case of metadata for differentiating keys. It is currently possible for an Issuer to assign a unique token key for each metadata value they support. This design pattern yields an increase in keys and can therefore complicate deployments. As an alternative, deployments can use one of the issuance protocols in this document with a single issuance key and different metadata values as the issuance public metadata.

8. IANA Considerations

This document extends the token type registry defined in Section 8.2.1 of [BASIC-PROTOCOL] with two new entries described in the following sub-sections.

8.1. Privately Verifiable Token Type

The contents of this token type registry entry are as follows:

* Value: 0xDA7B
* Name: Partially Oblivious PRF, OPRF(P-384, SHA-384)
* Token Structure: As defined in Section 5.3
* TokenChallenge Structure: As defined in Section 2.1 of [AUTHSCHEME]
* Publicly Verifiable: N
* Public Metadata: Y
* Private Metadata: N
* Nk: 48
* Nid: 32
* Notes: N/A

8.2. Publicly Verifiable Token Type

The contents of this token type registry entry are as follows:

* Value: 0xDA7A
* Name: Partially Blind RSA (2048-bit)
* Token Structure: As defined in Section 6.3
* TokenChallenge Structure: As defined in Section 2.1 of [AUTHSCHEME]
* Publicly Verifiable: Y
* Public Metadata: Y

* Private Metadata: N

* Nk: 256

* Nid: 32

* Notes: The RSAPBSSA-SHA384-PSS-Deterministic and RSAPBSSA-SHA384-PSSZERO-Deterministic variants are supported; see Section 6 of [PBRSA]

9. References

9.1. Normative References

[AUTHSCHEME] Pauly, T., Valdez, S., and C. A. Wood, "The Privacy Pass HTTP Authentication Scheme", RFC 9577, DOI 10.17487/RFC9577, June 2024, <https://www.rfc-editor.org/rfc/rfc9577>.

[BASIC-PROTOCOL] Celi, S., Davidson, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocols", RFC 9578, DOI 10.17487/RFC9578, June 2024, <https://www.rfc-editor.org/rfc/rfc9578>.

[PBRSA] Amjad, G. A., Hendrickson, S., Wood, C. A., and K. W. L. Yeo, "Partially Blind RSA Signatures", Work in Progress, Internet-Draft, draft-amjad-cfrg-partially-blind-rsa-03, 15 August 2024, <https://datatracker.ietf.org/doc/html/draft-amjad-cfrg-partially-blind-rsa-03>.

[POPRF] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups", RFC 9497, DOI 10.17487/RFC9497, December 2023, <https://www.rfc-editor.org/rfc/rfc9497>.

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

[RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters", RFC 5756, DOI 10.17487/RFC5756, January 2010, <https://www.rfc-editor.org/rfc/rfc5756>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

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

[TOKEN-EXTENSION] "The PrivateToken HTTP Authentication Scheme Extensions Parameter", n.d., <https://ietf-wg-privacypass.github.io/draft-ietf-privacypass-auth-scheme-extensions/draft-ietf-privacypass-auth-scheme-extensions.html>.

9.2. Informative References

[ARCHITECTURE] Davidson, A., Iyengar, J., and C. A. Wood, "The Privacy Pass Architecture", RFC 9576, DOI 10.17487/RFC9576, June 2024, <https://www.rfc-editor.org/rfc/rfc9576>.

[DSS] "Digital signature standard (DSS)", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.186-4, 2013, <https://doi.org/10.6028/nist.fips.186-4>.

Acknowledgments

This work benefited from input from Ghous Amjad and Kevin Yeo.

Authors' Addresses

Scott Hendrickson
Google
Email: scott@shendrickson.com

Christopher A. Wood
Email: caw@heapingbits.net