Privacy Pass Issuance Protocol

Abstract

This document specifies two variants of the two-message issuance protocol for Privacy Pass tokens: one that produces tokens that are privately verifiable using the issuance private key, and another that produces tokens that are publicly verifiable using the issuance public key.

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 17 March 2024.

Copyright Notice

Copyright (c) 2023 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
1. Introduction

The Privacy Pass protocol provides a privacy-preserving authorization mechanism. In essence, the protocol allows clients to provide cryptographic tokens that prove nothing other than that they have been created by a given server in the past [ARCHITECTURE].

This document describes the issuance protocol for Privacy Pass built on [HTTP]. It specifies two variants: one that is privately
verifiable using the issuance private key based on the oblivious pseudorandom function from [OPRF], and one that is publicly verifiable using the issuance public key based on the blind RSA signature scheme [BLINDRSA].

This document does not cover the Privacy Pass architecture, including choices that are necessary for deployment and application specific choices for protecting client privacy. This information is covered in [ARCHITECTURE].

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.

The following terms are used throughout this document.

*Client: An entity that runs the Issuance protocol with an Issuer to produce Tokens that can be later used for redemption (see Section 2.2 of [AUTHSCHEME]).

*Issuer: A service that provides Tokens to Clients.

*Issuer Public Key: The public key (from a private-public key pair) used by the Issuer for issuing and verifying Tokens.

*Issuer Private Key: The private key (from a private-public key pair) used by the Issuer for issuing and verifying Tokens.

This document additionally uses the terms "Origin" and "Token" as defined in [ARCHITECTURE].

Unless otherwise specified, this document encodes protocol messages in TLS notation from Section 3 of [TLS13]. Moreover, all constants are in network byte order.

3. Protocol Overview

The issuance protocols defined in this document embody the core of Privacy Pass. Clients receive TokenChallenge inputs from the redemption protocol ([AUTHSCHEME], Section 2.1) and use the issuance protocols to produce corresponding Token values ([AUTHSCHEME], Section 2.2). The issuance protocol describes how Clients and Issuers interact to compute a token using a one-round protocol consisting of a TokenRequest from the Client and TokenResponse from the Issuer. This interaction is shown below.
The TokenChallenge inputs to the issuance protocols described in this document can be interactive or non-interactive, and per-origin or cross-origin.

The issuance protocols defined in this document are compatible with any deployment model defined in Section 4 of [ARCHITECTURE]. The details of attestation are outside the scope of the issuance protocol; see Section 4 of [ARCHITECTURE] for information about how attestation can be implemented in each of the relevant deployment models.

This document describes two variants of the issuance protocol: one that is privately verifiable (Section 5) using the issuance private key based on the oblivious pseudorandom function from [OPRF], and one that is publicly verifiable (Section 6) using the issuance public key based on the blind RSA signature scheme [BLINDRSA].

4. Configuration

Issuers MUST provide two parameters for configuration:

1. Issuer Request URL: A token request URL for generating access tokens. For example, an Issuer URL might be https://issuer.example.net/request.

2. Issuer Public Key values: A list of Issuer Public Keys for the issuance protocol.

The Issuer parameters can be obtained from an Issuer via a directory object, which is a JSON object ([RFC8259], Section 4) whose values are other JSON values ([RFC8259], Section 3) for the parameters. The contents of this JSON object are defined in Table 1.
Issuer Request URL value (as an absolute URL, or a URL relative to the directory object) as a percent-encoded URL string, represented as a JSON string ([RFC8259], Section 7)

List of Issuer Public Key values, each represented as JSON objects ([RFC8259], Section 4)

Table 1: Issuer directory object description

Each "token-keys" JSON object contains the fields and corresponding raw values defined in Table 2.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>token-type</td>
<td>Integer value of the Token Type, as defined in Section 8.2, represented as a JSON number ([RFC8259], Section 6)</td>
</tr>
<tr>
<td>token-key</td>
<td>The base64url encoding of the Public Key for use with the issuance protocol, including padding, represented as a JSON string ([RFC8259], Section 7)</td>
</tr>
</tbody>
</table>

Table 2: Issuer 'token-keys' object description'

Each "token-keys" JSON object may also contain the optional field "not-before". The value of this field is the UNIX timestamp (number of seconds since January 1, 1970, UTC -- see Section 4.2.1 of [TIMESTAMP]) at which the key can be used. If this field is present, Clients SHOULD NOT use a token key before this timestamp, as doing so can lead to issuance failures. The purpose of this field is to assist in scheduled key rotations.

Beyond staging keys with the "not-before" value, Issuers MAY advertise multiple "token-keys" for the same token-type to facilitate key rotation. In this case, Issuers indicate preference for which token key to use based on the order of keys in the list, with preference given to keys earlier in the list. Clients SHOULD use the first key in the "token-keys" list that either does not have a "not-before" value or has a "not-before" value in the past. Origins can attempt to use any key in the "token-keys" list to verify tokens, starting with the most preferred key in the list. Trial verification like this can help deal with Client clock skew.

Altogether, the Issuer's directory could look like:
Clients that use this directory resource before 1686913811 in UNIX time would use the second key in the "token-keys" list, whereas Clients that use this directory after 1686913811 in UNIX time would use the first key in the "token-keys" list.

Issuer directory resources have the media type "application/private-token-issuer-directory" and are located at the well-known location /.well-known/private-token-issuer-directory; see Section 8.1 for the registration information for this well-known URI. The reason that this resource is located at a well-known URI is that Issuers are defined by an origin name in TokenChallenge structures; see Section 2.1 of [AUTHSCHEME].

The Issuer directory and Issuer resources SHOULD be available on the same origin. If an Issuer wants to service multiple different Issuer directories they MUST create unique subdomains for each so the TokenChallenge defined in Section 2.1 of [AUTHSCHEME] can be differentiated correctly.

Issuers SHOULD use HTTP cache directives to permit caching of this resource [RFC5861]. The cache lifetime depends on the Issuer's key rotation schedule. Regular rotation of token keys is recommended to minimize the risk of key compromise and any harmful effects that happen due to key compromise.

Issuers can control cache lifetime with the Cache-Control header, as follows:

Cache-Control: max-age=86400

Consumers of the Issuer directory resource SHOULD follow the usual HTTP caching [RFC9111] semantics when processing this resource. Long cache lifetimes may result in use of stale Issuer configuration information, whereas short lifetimes may result in decreased
performance. When use of an Issuer configuration results in token issuance failures, e.g., because the Issuer has invalidated its directory resource before its expiration time and issuance requests using this configuration are unsuccessful, the directory SHOULD be fetched and revalidated.

5. Issuance Protocol for Privately Verifiable Tokens

The privately verifiable issuance protocol allows Clients to produce Token values that verify using the Issuer Private Key. This protocol is based on the oblivious pseudorandom function from [OPRF].

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 URL: A URL identifying the location to which issuance requests 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 'Joint Origin and Issuer' deployment model, the Issuer Request URL 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 5.5.

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

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

The constants Ne and Ns are as defined in [OPRF], Section 4 for OPRF(P-384, SHA-384). The constant Nk, which is also equal to Nh as defined in [OPRF], Section 4, is defined by Section 8.2.1.
5.1. Client-to-Issuer Request

The Client first creates a context as follows:

client_context = SetupVOPRFClient("P384-SHA384", pkI)

Here, "P384-SHA384" is the identifier corresponding to the OPRF(P-384, SHA-384) ciphersuite in [OPRF]. SetupVOPRFClient is defined in [OPRF], 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(0x0001, // Token type field is 2 bytes long
nonce,
    challenge_digest,
    token_key_id)
blind, blinded_element = client_context.Blind(token_input)

The Blind function is defined in [OPRF], Section 3.3.2. If the Blind function fails, the Client aborts the protocol. The Client stores the nonce and challenge_digest values locally for use when finalizing the issuance protocol to produce a token (as described in Section 5.3).

The Client then creates a TokenRequest structured as follows:

```
struct {
    uint16_t token_type = 0x0001; /* Type VOPRF(P-384, SHA-384) */
    uint8_t truncated_token_key_id;
    uint8_t blinded_msg[Ne];
} TokenRequest;
```

The structure fields are defined as follows:

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

"truncated_token_key_id" is the least significant byte of the token_key_id (Section 5.5) in network byte order (in other words, the last 8 bits of token_key_id). This value is truncated so that Issuers cannot use token_key_id as a way of uniquely identifying Clients; see Section 7 and referenced information for more details.

"blinded_msg" is the Ne-octet blinded message defined above, computed as SerializeElement(blinded_element).
The values token_input and blinded_element are stored locally and used later as described in Section 5.3. The Client then generates an HTTP POST request to send to the Issuer Request URL, with the TokenRequest as the content. The media type for this request is "application/private-token-request". An example request for the Issuer Request URL "https://issuer.example.net/request" is shown below.

POST /request HTTP/1.1
Host = issuer.example.net
Accept = application/private-token-response
Content-Type = application/private-token-request
Content-Length = <Length of TokenRequest>

<Bytes containing the TokenRequest>

5.2. Issuer-to-Client Response

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

* The TokenRequest contains a supported token_type.

* The TokenRequest.truncated_token_key_id corresponds to the truncated key ID of a Public Key owned by the Issuer.

* The TokenRequest.blinded_msg is of the correct size.

If any of these conditions is not met, the Issuer MUST return an HTTP 422 (Unprocessable Content) error to the client.

If these conditions are met, the Issuer then tries to deserialize TokenRequest.blinded_msg using DeserializeElement from Section 2.1 of [OPRF], yielding blinded_element. If this fails, the Issuer MUST return an HTTP 422 (Unprocessable Content) error to the client. Otherwise, if the Issuer is willing to produce a token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:

server_context = SetupVOPRFServer("P384-SHA384", skI)
evaluate_element, proof =
server_context.BlindEvaluate(skI, pkI, blinded_element)

SetupVOPRFServer is defined in [OPRF], Section 3.2 and BlindEvaluate is defined in [OPRF], Section 3.3.2. The Issuer then creates a TokenResponse structured as follows:
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".

HTTP/1.1 200 OK
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)

The Finalize function is defined in [DPRF], Section 3.3.2. If this succeeds, the Client then constructs a Token as follows:

```
struct {
    uint16_t token_type = 0x0001; /* Type VOPRF(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 created in Section 5.1. If the Finalize function fails, the Client aborts the protocol.
5.4. Token Verification

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

```python
server_context = SetupVOPRFServer("P384-SHA384", skI)
token_authenticator_input =
    concat(Token.token_type,
          Token.nonce,
          Token.challenge_digest,
          Token.token_key_id)
token_authenticator =
    server_context.Evaluate(token_authenticator_input)
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:

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

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

```python
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 5 for producing publicly verifiable tokens using the protocol in [BLINDRSA]. In particular, this variant of the issuance protocol works for the RSABSSA-SHA384-PSS-Deterministic and RSABSSA-SHA384-PSSZERO-Deterministic blind RSA protocol variants described in Section 5 of [BLINDRSA].

The publicly verifiable issuance protocol differs from the protocol in Section 5 in that the output tokens are publicly verifiable by anyone with the Issuer Public Key. This means any Origin can select a given Issuer to produce tokens, as long as the Origin has the Issuer public key, without explicit coordination or permission from
the Issuer. This is because the Issuer does not learn the Origin that requested the token during the issuance protocol.

Beyond this difference, the publicly verifiable issuance protocol variant is nearly identical to the privately verifiable issuance protocol variant. In particular, Issuers provide a Private and Public Key, denoted skI and pkI, respectively, used to produce tokens as input to the protocol. See Section 6.5 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

*Issuer Request URL: A URL identifying the location to which issuance requests 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 URL 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].

Given this configuration and these inputs, the two messages exchanged in this protocol are described below. The constant Nk is defined by Section 8.2.2.

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:
The PrepareIdentity and Blind functions are defined in Section 4.1 of [BLINDRSA] and Section 4.2 of [BLINDRSA], respectively. The Client stores the nonce and challenge_digest values locally for use when finalizing the issuance protocol to produce a token (as described in Section 6.3).

The Client then creates a TokenRequest structured as follows:

```c
struct {
    uint16_t token_type = 0x0002; /* Type Blind RSA (2048-bit) */
    uint8_t truncated_token_key_id;
    uint8_t blinded_msg[Nk];
} TokenRequest;
```

The structure fields are defined as follows:

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

**"truncated_token_key_id" is the least significant byte of the token_key_id (Section 6.5) in network byte order (in other words, the last 8 bits of token_key_id). This value is truncated so that Issuers cannot use token_key_id as a way of uniquely identifying Clients; see Section 7 and referenced information for more details.**

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

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

```plaintext
POST /request HTTP/1.1
Host = issuer.example.net
Accept = application/private-token-response
Content-Type = application/private-token-request
Content-Length = <Length of TokenRequest>

<Bytes containing the TokenRequest>
```
6.2. Issuer-to-Client Response

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

* The TokenRequest contains a supported token_type.

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

* The TokenRequest.blinded_msg is of the correct size.

If any of these conditions is not met, the Issuer MUST return an HTTP 422 (Unprocessable Content) error to the Client. Otherwise, if the Issuer is willing to produce a token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:

\[
\text{blind\_sig} = \text{BlindSign}(skI, \text{TokenRequest.blinded\_msg})
\]

The BlindSign function is defined in Section 4.3 of [BLINDRSA]. The result is encoded and transmitted to the client in the following TokenResponse structure:

```
struct {
    uint8_t blind_sig[Nk];
} TokenResponse;
```

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".

```
HTTP/1.1 200 OK
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:

\[
\text{authenticator} = \text{Finalize}(pkI, \text{nonce}, \text{blind\_sig}, \text{blind\_inv})
\]

The Finalize function is defined in Section 4.4 of [BLINDRSA]. If this succeeds, the Client then constructs a Token as described in [AUTHSCHEME] as follows:
The Token.nonce value is that which was sampled in Section 5.1. If the Finalize function fails, the Client aborts the protocol.

### 6.4. Token Verification

Verifying a Token requires checking that Token.authenticator is a valid signature over the remainder of the token input using the Issuer Public Key. The function RSASSA-PSS-VERIFY is defined in Section 8.1.2 of [RFC8017], using SHA-384 as the Hash function, MGF1 with SHA-384 as the PSS mask generation function (MGF), and a 48-byte salt length (sLen).

```c
struct {
    uint16_t token_type = 0x0002; /* Type Blind RSA (2048-bit) */
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

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

### 6.4. Token Verification

Verifying a Token requires checking that Token.authenticator is a valid signature over the remainder of the token input using the Issuer Public Key. The function RSASSA-PSS-VERIFY is defined in Section 8.1.2 of [RFC8017], using SHA-384 as the Hash function, MGF1 with SHA-384 as the PSS mask generation function (MGF), and a 48-byte salt length (sLen).

```c
struct {
    uint16_t token_type = 0x0002; /* Type Blind RSA (2048-bit) */
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

```c
token_authenticator_input =
    concat(Token.token_type,
        Token.nonce,
        Token.challenge_digest,
        Token.token_key_id)
valid = RSASSA-PSS-VERIFY(pkI,
    token_authenticator_input,
    Token.authenticator)
```

### 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 securely, for example as specified in FIPS 186-5 [DSS]. These key pairs MUST NOT be reused in other protocols.

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. The parameters field for the digest used in the mask generation function and the digest being signed MUST be omitted.

An example sequence of the SPKI object (in ASN.1 format) for a 2048-bit key is below:
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

This document outlines how to instantiate the Issuance protocol based on the VOPRF defined in [OPRF] and blind RSA protocol defined in [BLINDRSA]. All security considerations described in the VOPRF and blind RSA documents also apply in the Privacy Pass use-case. Considerations related to broader privacy and security concerns in a multi-Client and multi-Issuer setting are deferred to the architecture document [ARCHITECTURE]. In particular, Section 4 and Section 5 of [ARCHITECTURE] discuss relevant privacy considerations influenced by the Privacy Pass deployment model, and Section 6 of [ARCHITECTURE] discusses privacy considerations that apply regardless of deployment model. Notable considerations include those pertaining to Issuer Public Key rotation and consistency, where consistency is as described in [CONSISTENCY], and Issuer selection.

8. IANA considerations

This section contains considerations for IANA.

8.1. Well-Known 'private-token-issuer-directory' URI

This document updates the "Well-Known URIs" Registry [WellKnownURIs] with the following values.
8.2. Token Type Registry Updates

This document updates the "Privacy Pass Token Type" Registry with the following entries.

8.2.1. Token Type VOPRF (P-384, SHA-384)

*Value: 0x0001

*Name: VOPRF (P-384, SHA-384)

*Token Structure: As defined in Section 2.2 of [AUTHSCHEME]

*Token Key Encoding: Serialized using SerializeElement from Section 2.1 of [OPRF]

*TokenChallenge Structure: As defined in Section 2.1 of [AUTHSCHEME]

*Public Verifiability: N

*Public Metadata: N

*Private Metadata: N

*Nk: 48

*Nid: 32

*Reference: Section 5

*Notes: None

8.2.2. Token Type Blind RSA (2048-bit)

*Value: 0x0002

*Name: Blind RSA (2048-bit)

*Token Structure: As defined in Section 2.2 of [AUTHSCHEME]
Token Key Encoding: Serialized as a DER-encoded SubjectPublicKeyInfo (SPKI) object using the RSASSA-PSS OID [RFC5756]

TokenChallenge Structure: As defined in Section 2.1 of [AUTHSCHEME]

Public Verifiability: Y

Public Metadata: N

Private Metadata: N

Nk: 256

Nid: 32

Reference: Section 6

Notes: The RSABSSA-SHA384-PSS-Deterministic and RSABSSA-SHA384-PSSZERO-Deterministic variants are supported

8.3. Media Types

The following entries should be added to the IANA "media types" registry:

"application/private-token-issuer-directory"

"application/private-token-request"

"application/private-token-response"

The templates for these entries are listed below and the reference should be this RFC.

8.3.1. "application/private-token-issuer-directory" media type

Type name: application
Subtype name: private-token-issuer-directory
Required parameters: N/A
Optional parameters: None
Encoding considerations: "binary"
Security considerations: see Section 4
Interoperability considerations: N/A
Published specification: this specification
Applications that use this media type: Services that implement the Privacy Pass issuer role, and client applications that interact with the issuer for the purposes of issuing or redeeming tokens.
Fragment identifier considerations: N/A
8.3.2. "application/private-token-request" media type

Type name: application
Subtype name: private-token-request
Required parameters: N/A
Optional parameters: None
Encoding considerations: "binary"
Security considerations: see Section 7
Interoperability considerations: N/A
Published specification: this specification
Applications that use this media type: Applications that want to issue or facilitate issuance of Privacy Pass tokens, including Privacy Pass issuer applications themselves.
Fragment identifier considerations: N/A

Additional information:
Magic number(s): N/A
Deprecated alias names for this type: N/A
File extension(s): N/A
Macintosh file type code(s): N/A
Person and email address to contact for further information: see Authors' Addresses section
Intended usage: COMMON
Restrictions on usage: N/A
Author: see Authors' Addresses section
Change controller: IESG
8.3.3. "application/private-token-response" media type

Type name: application
Subtype name: private-token-response
Required parameters: N/A
Optional parameters: None
Encoding considerations: "binary"
Security considerations: see Section 7
Interoperability considerations: N/A
Published specification: this specification
Applications that use this media type: Applications that want to issue or facilitate issuance of Privacy Pass tokens, including Privacy Pass issuer applications themselves.
Fragment identifier considerations: N/A
Additional information:
  Magic number(s): N/A
  Deprecated alias names for this type: N/A
  File extension(s): N/A
  Macintosh file type code(s): N/A
Person and email address to contact for further information: see Authors' Addresses section
Intended usage: COMMON
Restrictions on usage: N/A
Author: see Authors' Addresses section
Change controller: IESG

9. References

9.1. Normative References


9.2. Informative References
Appendix A. Acknowledgements

The authors of this document would like to acknowledge the helpful feedback and discussions from Benjamin Schwartz, Joseph Salowey, and Tara Whalen.

Appendix B. Test Vectors

This section includes test vectors for the two basic issuance protocols specified in this document. Appendix B.1 contains test vectors for token issuance protocol 1 (0x0001), and Appendix B.2 contains test vectors for token issuance protocol 2 (0x0002).

B.1. Issuance Protocol 1 - VOPRF(P-384, SHA-384)

The test vector below lists the following values:

*skS: The Issuer private Key, serialized using SerializeScalar from Section 2.1 of [OPRF] and represented as a hexadecimal string.

*pkS: The Issuer Public Key, serialized according to the encoding in Section 8.2.1.

*token_challenge: A randomly generated TokenChallenge structure, represented as a hexadecimal string.

*nonce: The 32-byte client nonce generated according to Section 5.1, represented as a hexadecimal string.
*blind: The blind used when computing the OPRF blinded message, serialized using SerializeScalar from Section 2.1 of [OPRF] and represented as a hexadecimal string.

*token_request: The TokenRequest message constructed according to Section 5.1, represented as a hexadecimal string.

*token_response: The TokenResponse message constructed according to Section 5.2, represented as a hexadecimal string.

*token: The output Token from the protocol, represented as a hexadecimal string.
// Test vector 1
skS: 39b0d04d37245928f5edb89bb02c2a4a0e6709f201d6c518871d5181
14910ee3c919ed1bffe3f3c1b87d53240a
pkS: 02d45bf522425cdd2227d3f27d2459d56308829252172d3e48469290c
21da1a46d2ca38f7beabdf05c974aee1455bf
token_challenge: 001000e6973737565722e6578616d706c65205de58a52f
daef25ca3f65448d04e040fbb1924e8264acfcf6c5ad451d582b3000e6f72696
7696e2e6578616d706c65
nonce:
6aa422c41b59d3e44a136dd439df2454e3587ee5f3697798c0df5fae73073b8
blind: 8e7f67089b8a009b31801a2e22d909d83bd5597c64dc149ed2b1
7ef82044ef3bd2ab2c4f54c5d1c956f90

// Test vector 2
skS: 39efed331527cc4ddff9722ab5cd35aeafe7c72520b0ca2e6bd2c98d3c
b12bc829afa9cc46558a96c2eeacc53097d865
pkS: 038017e005904c6146b371096c2a72b95a183aaa9ed951b8d8fb1ed9033
f68033284d175e7df8948745cd6a86bfb4e
token_challenge: 001000e6973737565722e6578616d706c65000e6f72696
7696e2e6578616d706c65
nonce:
7617bc802cfdb5d74722ef7418dbb4f2c88403820e55fe7ec07d319ac29d65
blind: 6492ee5072fa18d05d69c4246362df6e2621af95a10c03bb0109e0
f705b0437c4255327eae5266ec379e7015e
token_request: 00133033a5fe04a39da1bfb68ccdeecd197474dd525462e
5a90a6ba53b42aa1486fe443a2e17f3fd5ff028a1c7c1aeac5d
token_response: 04b8fcd624880d669c5cc688b056355c6e8e1bcbf3746c
b9ab9248a4c056f2a3a4876ef998a8b62b1d50f852c6fa68fc4fa13c79cbb5f
bd8b8f3c926e10c7c12f934a887d86da4a4e5b7e0f5a169aa7572087be69536
92a8f11f9cda7a7f281e4e3568e848225367946c70db09e718e3ca16193987b
c10bed3e3f54c4d036c17cd4015bb113be60d7aa92e0d
nonce: 0017167bc802cfdb5d74722ef7418dbb4f2c88403820e55fe7ec07d3
190c29d66569c94f75dcd2f97b13d4e8eb66e6d8f9dcaaa56851fb09125dfef1
34bd5a62a116477bc9e1a205c9a590c92335ca7a37e7163b2ac020b2d231c660
97f12333ef43b6d00801bca5ace0fab8b483dc04cd62578b95b5652921cd2698c
45ea74f6c827b4e19f01140fa5bd039686f562
// Test vector 3
skS: 2b770959b62b784f14946ae828f6e56caeb6eefe732c86e9ae50e818c0
55b3d7ca3af2beacea859a62ff7199d35cc
pkS: 03a0de1bf3fd0a73384283b648884ba9fa5dee190f9d7ad4292c2fd49db
4d64db6459df6f75bd7e626475c78934ae8d
token_challenge: 0001000e6973737565722e6578616d6500000176666f6
2e6578616706c652c6261722e6578616d65000d65
nonce: 87499b5930918d2d83ecb9f2d25ca0722aa11b80d8bbfd950537c28a7d3a9df
blind: 1f65958462ba15f44fd887b2e5fe4c273151b85dfbfaeaa4253ebba30
6104d9b73c7814c1423600e85a0942c0ff

// Test vector 4
skS: 2b770959b62b784f14946ae828f6e56caeb6eefe732c86e9ae50e818c0
55b3d7ca3af2beacea859a62ff7199d35cc
pkS: 03a0de1bf3fd0a73384283b648884ba9fa5dee190f9d7ad4292c2fd49db
4d64db6459df6f75bd7e626475c78934ae8d
token_challenge: 0001000e6973737565722e6578616d6500000176666f6
2e6578616706c652c6261722e6578616d65000d65
nonce: 87499b5930918d2d83ecb9f2d25ca0722aa11b80d8bbfd950537c28a7d3a9df
blind: 1f65958462ba15f44fd887b2e5fe4c273151b85dfbfaeaa4253ebba30
6104d9b73c7814c1423600e85a0942c0ff

// Test vector 5
skS: 46f3d4f56202b85ffcfdb4d06835fb9b2e24372861ecaa11357f1d29f9
B.2. Issuance Protocol 2 - Blind RSA, 2048

The test vector below lists the following values:

*skS: The PEM-encoded PKCS#8 RSA Issuer Private Key used for signing tokens, represented as a hexadecimal string.

*pkS: The Issuer Public Key, serialized according to the encoding in Section 8.2.2.

*token_challenge: A randomly generated TokenChallenge structure, represented as a hexadecimal string.

*nonce: The 32-byte client nonce generated according to Section 6.1, represented as a hexadecimal string.

*blind: The blind used when computing the blind RSA blinded message, represented as a hexadecimal string.

*salt: The randomly generated 48-byte salt used when encoding the blinded token request message, represented as a hexadecimal string.

*token_request: The TokenRequest message constructed according to Section 6.1, represented as a hexadecimal string.

*token_request: The TokenResponse message constructed according to Section 6.2, represented as a hexadecimal string.

*token: The output Token from the protocol, represented as a hexadecimal string.
token: 002aa72019df951df97021ce63876fe8b0a8d2c13a12b0a2dd150 8d07827f055969f643b4cfa5196d4a866b5368834f4f6de46950ed435b3b8 1bd03644ca572f8982a9ca248a3956186322d93ca14726621ddee5632c97f1 71dc2708cb6a21b53d7294b5e900fa55377d3e3b3ce4e08c9676d1e553f8fd 184b0e06c637174f5206b14c7bb0e724ebf6b56721e5aa2ed94c051ca433d302 b23bc52468010d49e248f8982a9ca248a3956186322d93ca14726621ddee5632c97f1 // Test vector 2 skS: 2d2d2d2d2d424547494e2050524956415445204b45592d2d2d2d0a4d 4945765144424144494e02676b7168e6b694739773042154564614153434246b 3677676536a41675414166f4642151444c47753172617658317633442oa4f6b 7a3871795375355739566f6a4136354355466671444774e38366a42a5f4a7f6 57245526b4913c52787764d6453237972632633616b4754714c75b644a0a55a 35743561496b317214763465584464e4a459034232505570851436e696939e6 b492b667725769444444948713e8613975866e6c5079596f78f453oa646f 655586383546f314a752b2397333656d8634516a755139545596149138337 1724450667a50357557812b5e4d636739723226986b76624c766d42390a6a41 355334475666325a6c74875954736f4c368472377a58e69ea4e94637486271656 76f755397665b525d458645352f2ba4a395659a634a734a62a4c756576480a544f 72535a4d4949525853514d166414f45a4547426d6d4436063566672f44373 475676a79486e4e5138373341eb6a55716d367675475413872514c62a4530 742b496c706641674d424145436764541c7a436264a69316a506345384d6 b562b434c667765325b7b264868e762672466502f566344787275690a3270 31613584a596962653645532b4d622f4d4655646c4850676414c7731875134576 5726636e4444376866c67845753536874327366338f364759320a6359 366f7770424476632616847b556b503465b6239538584ca576347534736 1556e484a8523769e7834345a6c666f4cc6245516536658578734d10a6230 64487868444424ed4476656777674b6f6a4f6a76032f39386d4555793756422 f3661326c7265676c766a632f326eb43a4b745937734743645716c47460a78a 4142615775384d453a42f52131334c762b42656662717449397315a776a7a 264556851483865437872793251564d515751696e57684174364d7154340a5342 5354726f6c5a77772716a5384d50a4393175614ed6458474c63484ac932367 3587a76374b53514b4267514476637735055557641395a325a583953850a6d49 784d542245467a5662550754ba4b413197576e315444e63556a711682b7 a652f376b337946786b863014633316271360654c39394749536914f0a35ab 4f574d39454bf62f78a415326214b314d664f5931472b386a7a4258557047233 9346b5353383785968d66eq7946776379342a35a683566b55710a5732 306f5362666b86a5264537a48326b52476972672b553774b426751445a4ad6 e7279324578612f3435713750626f737841504d69596e6b35a415953470a7932 7a305a3754562b27584514f2f2b7504d376e433075794c49444396c61544d4 8776e3637372f4c62476f45503157267706f59482f4321346b2f526e3609a675 77524e3632496f397463392b41434c745542377674476179332b675277597543 33262356564386c49696675745466b5613968307545435728417456733oa6e35 6b796132513976514b426746a75467a4f5a742b7467596e576e515545675385
0304f494a45484d45345554444f637743784b72485272393344aa7546320a4533 77644bbf54696937507774f59496f614a568760a50634aa6262462664b792b6 e735170351947763977644a724d6156774a637469777556376315579660a5674 4c6164676c767071333647e4d3868ae4d3058733661a6d6e6665573 974958435370684d727a4c4a6c39463036394324316f4742414e58760a7567 565872783227354316f6b636765531427367794aa5605774e5264336354a3b 9a7a061530344145431504e6b70655177486772f2b36665361546f487a0a7941 78447339685272672856216667385427b2145548378731459445665159564 d68555262546f5a6536476f6a716e54333664e66455137845a666f749a0a306c 6f4d486777657362b53494d4336f565325a63747555a633326c634961639726 248f64f3366146f4746153186385394944e74376444f43131345455509a6d30 31414a49597737416c5233756f2f524e61432b7856450555334736b75414c78 86944522f75734c45142436a6457656d64a415765e155447626e5940a5363 7752384732a46366e7245437462747973373354154676f6f465a6e63d50c45 0386c784c796265c534244454c79615a762f624173500c4d4f39624435630a4a2b 4e534261612b6f694c6c31776d4361354d43666c6330a2d2d2d2d454e44205 0249564154520b4b5592d2d2d2d0a pks: 30820152303d06092a864886f70d0101a030a0d30b060966480165 03040202a130180692a864886f70d0101080b06096648016503040202a 03020130082918f003f802910a02820100cb1aed866a95f5bce01a34ccf52 25b94b2e4a2303e4e24a07e4e7e5376a12993a1f2b1190984db471ec31d4 b6c9ad9cda09126a2ee903523e6de5a224d6b02f09e5c374d0cefe01d8f529c50 0a78a2f67908f8a6b5a2b4303c81efaf1af72d7be794fc98a313927687975ce4 53b526ef9f6e0b99798b423b9f4461aa2af37aab0cf5733s797abe44dd13c7 32db6a18116cbe607d8c6e2e065f5d996dc584eca0be87afbcd78a337d17b 1dba9e828bd08e192317144e7f89f55619709b096cbb9e474ceed264c2073f e49740c01f0e0ee109106066983d21e5f83f086e2e823c879cd43ceff700d2a352a9 babd61d03cad8b1d347b225a5f0203010001 token_challenge: 0002000e697373756572e6578616d706c65000007e6f7296796e2e6578616d706c65 none: 98c1345ff38a554b429b428b0f206cfe4f3892f8041995f2c24873d90e84488d blind: 7b8b58f89c9b83a0e2b02938b3396f06f8f3df3f0012a9f1a2cc54168a5 52994d063f3d450bea1b3bfef8d5e014316e466e2e384549c2f695ac3aff5f66 ef1eb0f113df252006bd66e4eab373a582666a8a3a75692535e1fc46d016fb6 f37c94980817e20c0b77a48570a1fc4b74325b2d23dcbe52b5d6a9e39f7ace7 3b99004ea8e8c8bfff2b4b533ea63cbfc4f7a4c95ccffcb0c4e43bc4992c1fdbe0e7 a7a74475df0894cf25125ece901abbea607a9050df98ec3d0d6341f6eab40 ee99c9c2cc0b560b8377f8543d8878c7458885fd285c7556c88fc602016707b42 2c8a3a806905196f153s32789b28fddbe3d0288ee11d7c80149753893164ea3 salt: b6b437842ab0eaa677ce3f4036fd0498edee458ad81ea519ce8bde3fcd5 ec1505d28e110d7b4d4cacc5e04eced54d11a token_request: 00020892d26a271e01064657ba10c0b5c2b287bb209d86e8002 7f96fbbf86e10f4fcb97f0f0c42649843314ee9cb84a95194fe4e40bdd3802 a121cb6b59a4ae7e032552754c4ab70f1991c82ead402660c0ef912178b0a0f6 d303e6a966079230592827b84979d6cbbf5f21ab8904e990863de7f05c4f8af8a 053c19a6609726606c6463976e4c6e0eab335222d3f96482841db4a825d5 adcc33d9259284cd1e5a3f490d581f1306705c5cc561d82373f1dbd72ae4b 2feaf7339f5d686415f59312766e3074ee4a7305f5053da82673ee7477a727a
token_request: 0002080f6b84fba1822c577c8cd670f1136ca107f84d9d465d5ed2ad15da975358f0313433ba42e688997392927efe29284d1c32389a122f40e63b0b363dfccbed7a55fbeb8d5362dcbcaef6dc0a78739e0655b98a7d7f83913a59f7d798e9cf6c262c9e90a7e8843e3c355eabcd44e5f6779feea6a78b05ac352fdd51a11de68b0bf3fcd6a3d53a88c9f7477ce908f868b33562691f540e3e88562d92c17ccf78ce6f9b53312a5f2dcb91ebd1c9d9d659017ca908001cc755cb5437989364dc92f0e8ea18f66d3141f6eb02ad68af41de1a3f9e925d5a4ca07606fc4ac28b3317e939f6573442c6d03be17cd141fa8260d32d134c6b
token_response: 2dd08ce89cf4f62bc236ab7b75266ea13c57c5705345e3280eb1a17053c74cb6e5bf990716950440628ea2e37dc59c6d684f9a9656cb9cbbfffb869516bf1d9a90d69cc52a28890bd4782f5a6eafada885b6861a747109ce91208913b9d0e7a49f3f7397bd841cc8b5c68802de3d68df903269c22b8095122458986a8dfe1ea8eb6b23c93b396e9fcb973593eob19050d5656c2b0b48a45d98a81614f631bf82a7e4588413b44a0cb6d94e94f2a134790b396cb71e3ed3b557b5bd834e72657a97abda8649763b18d0e289b794801d4383e0d4f825dce0ded98c433db81c0282dd833a4fc24961f60e118d4421dce5b611d53e9ca96156a52509bf9af9eb7e
token: 0029e7a22b2dc517d5268434ecbc07eb5fa53f62f2776a17a6d91757af1592df9e710042ee445ac4dd5ac8bf6e65c4d8d4d7f37f743d507ef9a46d727d2774f3ca572f8982a9ca2a8395616832293c1a172662121deeb6532c07ff1f71cd278015b0108bc6d955e9c856d2e9fffaefba007d33cd25452fbeb0b15919b973e0d9180aae64eb61b57e698e65363c4f84f4a53d1f82ef19edf7d7a5aa11a5038e2a32a12bd44a14e0ca7c7f451ca5f865f5910003fcb9ff5ff5fa2408c28d6087737d03a651ea9bfafcc2747a6380e19a1d160fcd5c25df79dad68ab3de8e926e08ca1addced79277f7b56398ef59c26e725df0a976a0f2a936ca42

// Test vector 4
skS: 2d2d2d2d2d424547494e2b050252495614545204b45592d2d2d0a4d9494576514942414441ae26767b1686b6947397739042151456414153434246b4776767536aa4164541416f4942451444ca4775317261705381376334240a4f6b7a38717953759375b6b66174744774e38366a42b5a4f7645724552b49314c527876734d6453327961326333616b4745714c756b440a556a357435614966b3172417643655844644e45003432530557057851436e699396e6b492b66757b9677444444471386139731375866ec5079596f784ff530a646ff6558563835464f314a752b26397336356d586d34516a7551394559614971338772450657a560335758712b524ed63637932326968763624766d42390a6a4135533447566325a6c7478594573f4c364872377a5869a4e39436748672165676f7596765b452db548645352f2b4a3956595a634a734a624c75657980a444f72535a4d4948502b5358514d466414f4f454a457426d6d440683566672f43473475676a94864e5e53387334414e6b6a557163676574547443872514620a4530742b496c706641674d4241415436767454147a3a4362647a69316a5064358346d
babd612d03cad02db134b7e225a5f0203010001
token_challenge: 0002000e697373756572e6578616d706c657373
nonce: 494dae41fc7e300c2d09990afcd5d5e1fc95305337dc12f78942c453400bfe8e6
blind: 097cb17bcdedcfec058dff5c4e517d1e367ab8f46252bad1933ba378c32625c0db69f5655c2003fb39e75810796cd63675b223cf3162c57188d56e9584cfce6cad82974379ada38a095e3b012921b31ccde7425f93464e235fb1752be3a6df2913da63a1543a3aad505bf218c471dfbbc12fb304158e29b6ed355b079e23f1e6173c5dec4545840bbe58e5ad37cbea0a10dca5d9df2781589d27c34108477b52c0d32a1370c17f703941fbbfba0a07a6794fe725787099cbbff8f21ee79229bb9491eb6a8c14764e686e64e4ff0ae913797067aa0826f366c3193e103b05653c73b52d7f852a185dccfb806da700db9f53abb848545b7d47f7c28f3salt: 49912979f1f5b28e5b8228ab13282df74319dece7bdf4521ceb100dcef042a2df8e25fc9db59b64a5f6493c28250424octoken_request: 00020824840627ca8c62098b14caded9a1998ba388c3dd8541e962fb68a0071535d958d18494afdec11da44a88c3b38645f5a8f623b697c5d6348594e1a75479408a72c0ed179b7075609ca7eb6ed3582f572df38cf60fcede1a525c5ed7b234356b020a99f66df200f4f323c9aa54307db966d4457c37542b66bb183ddeafca914fc74831868b5d52f498ee3d165685f49a8d86e39fe6c4b7ec678f5250908d25e5b873c69b42423687121aa4210cadd66fc6499073cbb97a38e827a0e742470f090c2f49dc606e849740db7f73df0cbcb96c10b02af0d7de7971ecallff8e1b4929e59f3cf319ede9bda29a6d96b843083b5d4242f3448d76ada088b8014f70b97e19token_response: 27466ff644cfffba28a2c19395fa19dfb61fd135daa837844fb9f9be6c253e6469f553ae6ddc09f4b833b15e58f571134a34f2454993e373419549c2911cf94f2f68f3996d47f71e8d8d6ec5b1c074bf74afa59ede4cb3f32f5f08d45ea45492f0279c3ba18d852698ede1b651e8e09e22ba27386c0f2eb2f6a826023ed3b36cf343a518100829631661284325787fc941ea3bafe7b6761b708e90397837f74b4f0fc8388ec8eaf7242089dd5561f57735926cbad219fc9fee85ae49a8e951f63ca194b7ff618c0e6ee02267e7267bb996432dc76973819da80e3e86947bba4b363da972dafa3ad0eb0144b3250f2c6799969db3c3e513905d4bc0b8ctoken: 0002494dae41fc7e300c2d09990afcd5d5e1fc95305337dc12f789424c53400bfe8e6b741ec1b6fd05f1e9f5f8928906ace1612896dca9753eef94ad3c9fe9237a4ca572f8982a9ca4830651863222d93ca147266121debe5632c97f171cd270a55c83d04292b5d92ad1a87b37e54f22f61c58840586f39c50b231824423378dcdcf50e69dc817d455bfad06c7f2a0ac35d2adcf72f6b0bcb9954c192b0a0ef28a2a560e3909989d3cb11667acb1716d3dd2d219dc5a3b1ea6206359d0e002d2bc4fa7e69fb07214b90addcebd2203d1e17f57fca500bccc5a13e0ac15cf942182cc2b5deaa737a71704114e357e2ec2f10047463ded02a1a0766cd34ed77219711e03ac95eb2584c116140dc903de3738ae742ab5ed8c5139fc87145a789bf989714e68f0a6682b784a9bb4c4b3d1b2667106f397b35b641d882d7b0185168946de89ef72349a44a47bb7dd6d46e9b9a9ba543d5701b65c63645c2

// Test vector 5
skS: 2d2d2d2d2d424547944e2e956524956415445240b45592d2d2d2d2d0a4d9494576514942414441e42676b717686b9473977390421514564153343424b63777677536a4174544146f49424151444c477551726170583176334420a4f6b7a3871795735379356b6f6a41303545354b6671744477e38366a242b5a4f764
Authors' Addresses

Sofía Celi
Brave Software
Lisbon
Portugal

Email: cherenkov@riseup.net

Alex Davidson
Brave Software
Lisbon
Portugal

Email: alex.davidson92@gmail.com

Steven Valdez
Google LLC

Email: svaldez@chromium.org

Christopher A. Wood
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: caw@heapingbits.net