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 3 August 2023.

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
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
2. Terminology
3. Protocol Overview
4. Configuration
5. Issuance Protocol for Privately Verifiable Tokens
   5.1. Client-to-Issuer Request
   5.2. Issuer-to-Client Response
   5.3. Finalization
   5.4. Token Verification
   5.5. Issuer Configuration
5.6. Issuance Protocol for Publicly Verifiable Tokens
   6.1. Client-to-Issuer Request
   6.2. Issuer-to-Client Response
   6.3. Finalization
   6.4. Token Verification
   6.5. Issuer Configuration
6. Security considerations
7. IANA considerations
   8.1. Well-Known 'token-issuer-directory' URI
   8.2. Token Type Registry Updates
      8.2.1. Token Type VOPRF (P-384, SHA-384)
      8.2.2. Token Type Blind RSA (2048-bit)
   8.3. Media Types
      8.3.1. "application/token-issuer-directory" media type
      8.3.2. "application/private-token-request" media type
      8.3.3. "application/private-token-response" media type
8. References
   9.1. Normative References
   9.2. Informative References
Appendix A. Acknowledgements
Appendix B. Test Vectors
   B.1. Issuance Protocol 1 - VOPRF(P-384, SHA-384)
   B.2. Issuance Protocol 2 - Blind RSA, 2048
9. Authors' Addresses

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 [BLindrRSA].

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.
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>issuer-request-uri</td>
<td>Issuer Request URL value (as an absolute or relative URL) as a percent-encoded URL string, represented as a JSON string ([RFC8259], Section 7)</td>
</tr>
<tr>
<td>token-keys</td>
<td>List of Issuer Public Key values, each represented as JSON objects ([RFC8259], Section 4)</td>
</tr>
</tbody>
</table>

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>

Issuers MAY advertise multiple token-keys for the same token-type to support 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.

Altogether, the Issuer's directory could look like:

```json
{
    "issuer-request-uri": "https://issuer.example.net/request",
    "token-keys": [
        {
            "token-type": 2,
            "token-key": "MI...AB",
        },
        {
            "token-type": 2,
            "token-key": "MI...AQ",
        }
    ]
}
```

Issuer directory resources have the media type "application/token-issuer-directory" and are located at the well-known location /.well-known/token-issuer-directory; see Section 8.1 for the registration information for this well-known URI.

Issuers SHOULD use HTTP caching 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.

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 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 configuration information is too stale, 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 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 'Joint Origin and 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 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 is defined by Section 8.2.1.

5.1. Client-to-Issuer Request

The Client first creates a context as follows:

\[
\text{client\_context} = \text{SetupVOPRFClient}(0x0004, \text{pkI})
\]

Here, 0x0004 is the two-octet 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:

\[
\begin{aligned}
\text{nonce} &= \text{random}(32) \\
\text{challenge\_digest} &= \text{SHA256}(\text{challenge}) \\
\text{token\_input} &= \text{concat}(0x0001, // \text{Token type field is 2 bytes long} \\
& \quad \text{nonce,} \\
& \quad \text{challenge\_digest,} \\
& \quad \text{token\_key\_id}) \\
\text{blind, blinded\_element} &= \text{client\_context.Blind}(\text{token\_input})
\end{aligned}
\]

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:

\[
\text{struct} \{
\text{uint16\_t token\_type} = 0x0001; /* \text{Type VOPRF(P-384, SHA-384)} */ \\
\text{uint8\_t truncated\_token\_key\_id}; \\
\text{uint8\_t blinded\_msg[Ne]};
\} \text{TokenRequest;}
\]

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).

"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 URI, with the
TokenRequest as the content. The media type for this request is
"application/private-token-request". An example request is shown
below.

```plaintext
.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 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 400 error to the client. The Issuer then tries to deserialized
TokenRequest.blinded_msg using DeserializeElement from Section 2.1
of [OPRF], yielding blinded_element. If this fails, the Issuer MUST
return an HTTP 400 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:

```plaintext
server_context = SetupVOPRFServer(0x0004, skI, pkI)
evaluate_element, proof =
    server_context.Evaluate(skI, blinded_element)
```
SetupVOPRFServer is in [OPRF], Section 3.2 and Evaluate is defined in [OPRF], Section 3.3.2. The Issuer then creates a TokenResponse structured as follows:

```c
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])).

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:

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

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

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:

server_context = SetupVOPRFServer(0x0004, skI, pkI)
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:

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

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 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 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].

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:

nonce = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0x0002, // Token type field is 2 bytes long
  nonce,
  challenge_digest,
  token_key_id)

blinded_msg, blind_inv =
  Blind(pkI, PrepareIdentity(token_input))

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).

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

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

```c
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".

>: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, blind_sig, 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:

```
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;
```

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).

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 as as specified in FIPS 186-4 [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.

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, the privacy considerations in Section 4 and Section 5 of [ARCHITECTURE], particularly those pertaining to Issuer Public Key rotation and consistency (where consistency is as described in [CONSISTENCY]) and Issuer selection, are relevant for implementations of the protocols in this document.

8. IANA considerations

This section contains considerations for IANA.

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

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

<table>
<thead>
<tr>
<th>URI Suffix</th>
<th>Change Controller</th>
<th>Reference</th>
<th>Status</th>
<th>Related information</th>
</tr>
</thead>
<tbody>
<tr>
<td>token-issuer-directory</td>
<td>IETF</td>
<td>[this document]</td>
<td>permanent</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 3: 'token-issuer-directory' Well-Known URI

8.2. Token Type Registry Updates

This document updates the "Token Type" Registry from [AUTHSCHEME], Section 5.2 with the following entries.

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

*Value: 0x0001
8.2.2. Token Type Blind RSA (2048-bit)

*Value: 0x0002

*Name: Blind RSA (2048-bit)

*Publicly Verifiable: 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

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

*Token issuer directory: "application/token-issuer-directory"

*TokenRequest: "application/private-token-request"

*TokenResponse: "application/private-token-response"

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

8.3.1. "application/token-issuer-directory" media type
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: N/A
Fragment identifier considerations: N/A
Additional information:
    Magic number(s): N/A
    Deprecated alias names for this type: N/A
    File extension(s): N/A
    Macintosh file type code(s): N/A
Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON
Restrictions on usage: N/A
Author: see Authors' Addresses section
Change controller: IESG

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 8
Interoperability considerations: N/A
Published specification: this specification
Applications that use this media type: N/A
Fragment identifier considerations: N/A
Additional information:
    Magic number(s): N/A
    Deprecated alias names for this type: N/A
    File extension(s): N/A
    Macintosh file type code(s): N/A
Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON
Restrictions on usage: N/A
Author: see Authors' Addresses section
Change controller: IESG
### Private-Token-Response

**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:** N/A  
**Fragment identifier considerations:** N/A  
**Additional information:**  
  - Magic number(s): N/A  
  - Deprecated alias names for this type: N/A  
  - File extension(s): N/A  
  - Macintosh file type code(s): N/A  
**Person and email address to contact for further information:** see Authors' Addresses section  
**Intended usage:** COMMON  
**Restrictions on usage:** N/A  
**Author:** see Authors' Addresses section  
**Change controller:** IESG

### References

#### 9. References

**9.1. Normative References**


**[RFC2119]** Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/
9.2. Informative References


[CONSISTENCY] "*** BROKEN REFERENCE ***".

[DSS] Laboratory, I. T. and National Institute of Standards and Technology, "Digital Signature Standard (DSS)", DOI
Appendix A. Acknowledgements

The authors of this document would like to acknowledge the helpful feedback and discussions from Benjamin Schwartz, Joseph Salowey, Sofia Celi, 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 using SerializeElement from Section 2.1 of [OPRF] and represented as a hexadecimal string.

*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.

skS: 08f572b675c83bf83c8037e503816119409a21d26e097414678eb44c625f
     cdd9db2e4eb16dbbcc9755ae745fffa3f4fa
pkS: 0371b63695ddf7965ff770ced74c17938d60c9cb9d8b9537614072b001ff
     c608e80f310cdb4475487736f0f9d1406c7c9
token_challenge: 0001000e697373756772e6578616d0065000000e6f7269
     67696e2e6578616d0065c6
nonce:
     1a177bae66ea3341c367c160c635aa52daef9f105bb1240d06a063ae12e9798a
blind: 1e46366a7b619ae77e24d2b853f5ddc64524eb5a78f4e3af108f0291
     9827cbdee2f8d753869ab9229aeb7fe9988763
token_request: 00017f023d788d4089a5f76f908ce26d18bb3b8ee826223b8a
     1df70a652e092aaf235c44c6f1e57f81d17d31632d090d260dc531
nonce: 03c1854b0cb631ceff11079299fde5c8d9f94c67d6dbc62
     b259916a4db969e39ac38817fafa9ae48842c610d41bf0bb6f93ae6e3025acf22
     38c0ef62e0b628437944cd8db0207c8eb9c3025fcaacdb0e520576c7ad9bb9cc18
     4668716e7c5226bdfd0c898e980d5d90eb60e5533045358e3063b63a24cc2f5
     5891c1ded1a7642ef945bcecc888e92e15d5eccb431f6c6d
token: 00011a177bae66ea3341c367c160c635aa52daef9f105bb1240d06a063
     ae12e9798ac994f7d5cd2cfbb978b13d4e8eb6e6d8f9dcdaa65851fb091025de1
     34bd5a62a7f13956db752669425e8eb1128273c17972b5f16a9bc835a9c9f357
     72a2add9f5e1bb3ab71770ada81fa1af0fbd4a476fc92a3ff25fac14639b7fe3
     4365118ae2ff55a2399e15800ec9aa759659317

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 DER-encoded SubjectPublicKeyInfo object carrying the Issuer Public Key, as described in Section 6.5, represented as a hexadecimal string.

*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.
pkS: 30820156304106092a8648866f78d01010a3034a00f300d06096086480165
030402020500a11c0a660992a4648866f70d010108300d060960864801650304
2020500a203020130382010f003082010a202820100cb1aed6b9a9f5b1ce01
3a4ccfca2b59b4e2a30425067ea4b3c0df9a8c12993af12b1119864d47
1bec31d4b6c9ad9cda90912a2ee903523e6de5a224d6b02f09e5c374d0cfe01d
8f52c500a782a679f0fa682b5a2b430c81eaf1af7d27b5e794f989a31392768
797574c5b26e9f8b66e9b843239b0f4461a223f37a0b0f573f7597ab
e44d31c732db68a181c69cbe6078c0e5e2605f59966c584e0c8e087abf7cd8
a337d17bf1db9e828bbd81e391371144e7ff89f5619798b096cbb9e474ead2
64c2073fe49740c1f00e019106066983d21ef8f8086e2e823c879cd43fcef0
02a352a9bab661203cad02db134be225a5f020301001
token_challenge:
d0bea78c3b452a4cdd4484e4c6f1e73d3c949be58a5a81c7c97f12fe5f9b03c
nonce:
23b29061bc9d3cf5e673e47abe5122d35569d18a3fcf249f4d4dd4ae7d61
blind: 1260c6f526412ac47163292fe2b14232e369eaf05ce7593ca6426c415
b44f9d44d6e6a4f57993beb75a8a06a7d7a45b9c27ab930fdef46a3c5f
d46fed54849f4d03fd6f5062b33ed5e70d6266a74dd667563ca0bdf0f5fae416df
b86b32ab152d460c5add6b82273c75599f50aabe1b9ae9e6a1384a50738
5e492de216f6bd94776aba4a6843d8068fddd33b3014aca2aa127045bc99b0b43011c
ed53f628896b6d222b9bc1b9a1e0afe6b1e355af27f00b189d371422976ebb
9c52da0372c31c7b5030ab736a289372e555dcd5970f0ce79cddd10abf37dd60
ad2e1168081aa5159041346cf7b4fa8052dc167cfc83fc367bfac2111ae3e00
salt: 660138f93e216110f271680079b11f2dafa8ef31ab539eccee166c012085
79a61f043c7c8253f830ad3b06bb8aa4314aa
token_request: 0002f84040a6a43a8297dc0677f7c51fe71ab33ba77d113345f
12dc6224376bbae08af328e8df057f0c890cf108ed59de5f87cd7811bfc3d6f6a
9c680447b2b8d38cd379776c86dc1667bf80636321374d15cd728321fcd939ea
ef01ebcfe52a7e0c1fecd4474443f7ca8e6c304793eb075a83140ca9e409e7687
e5e2e0c918e8e3913fe6a121b2588a4ec90d7f722243b13c50d58e
52d78916ac4bb4d3597c7c816ed2d997383f5b90e565c36b9c19b63dee07538a
c9e5d60e3e28bb3b651ed6b6b3b6c647b327b1536f1176ee62e1f3708a94e6cd
447610b2c8347
token_response: 67393299cc66da3b450f1c73836a918e54392ce11cc9a8a
a26eb2dbb8a3f182527b7b9e7f90d8fd97bd2773a39d1d8ca66f597e458548d6b
bc60826bb776c8b6be017913d284be7d04069c667db22ce49fe87b5e936e5b6
8f6f634d782f281ea2a5a913cf9b74b5e60153f9f2fcfebe4099c3bcbf9d0c
bd63197bef30a451c9f7677665c46112a2296b5e359789bcb6c13d2bd6a49b985
89b3580225bad2014fc433c18013281cf064fe894cb307613dccc3a4f197d1
119b572f9f3c2d41ce2008e0baafad393ff4f95b9d16624a7a97f731ce998
7aa87810f586b93a14b996ea949ada84e1d56de7a31a5bd99e21f3652a4d6ab
d76f2d6d
token: 000223b29061bc9d3cf5e673e47abe5122d35569d18a3fcf249f4d4
4dd4ae7d61a982f3054c8c8d7d827eb066c7bb05b1f65a8e6e2968895c0ed1
Authors' Addresses

Sofía Celi
Brave Software
Lisbon
Portugal

Email: cherenkov@riseup.net

Alex Davidson
Brave Software
Lisbon
Portugal

Email: alex.davidson92@gmail.com

Armando Faz-Hernandez
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: armfazh@cloudflare.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