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, and another that produces tokens that are publicly verifiable. The privately verifiable issuance protocol optionally supports public metadata during the issuance flow.

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 4 August 2022.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in
Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction
2. Terminology
3. Issuance Protocol for Privately Verifiable Tokens with Public Metadata
  3.1. Client-to-Issuer Request
  3.2. Issuer-to-Client Response
  3.3. Finalization
  3.4. Issuer Configuration
4. Issuance Protocol for Publicly Verifiable Tokens
  4.1. Client-to-Issuer Request
  4.2. Issuer-to-Client Response
  4.3. Finalization
  4.4. Issuer Configuration
5. Security considerations
6. IANA considerations
  6.1. Token Type
  6.2. Media Types
    6.2.1. "message/token-request" media type
    6.2.2. "message/token-response" media type
7. Normative References
Appendix A. Acknowledgements
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 [I-D.ietf-privacypass-architecture].

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

This document DOES NOT cover the architectural framework required for running and maintaining the Privacy Pass protocol in the Internet setting. In addition, it DOES NOT cover the choices that are necessary for ensuring that client privacy leaks do not occur. Both of these considerations are covered in [I-D.ietf-privacypass-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 provides authorization tokens to services across the Internet, in return for authorization.

*Issuer: A service produces Privacy Pass tokens to clients.

*Private Key: The secret key used by the Issuer for issuing tokens.

*Public Key: The public key used by the Issuer for issuing and verifying tokens.

We assume that all protocol messages are encoded into raw byte format before being sent across the wire.

3. Issuance Protocol for Privately Verifiable Tokens with Public Metadata

The Privacy Pass issuance protocol is a two message protocol that takes as input a challenge from the redemption protocol and produces a token, as shown in the figure below.

Issuers provide a Private and Public Key, denoted skI and pkI, respectively, used to produce tokens as input to the protocol. See Section 3.4 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

*Issuer name, identifying 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 key_id computed as described in Section 3.4.
Challenge value challenge, an opaque byte string. For example, this might be provided by the redemption protocol in [HTTP-Authentication].

Both Client and Issuer also share a common public string called info.

Given this configuration and these inputs, the two messages exchanged in this protocol are described below.

3.1. Client-to-Issuer Request

The Client first creates a verifiable context as follows:

client_context = SetupVerifiableClient(0x0004, pkI)

Here, 0x0004 is the two-octet identifier corresponding to the OPRF(P-384, SHA-384) ciphersuite in [OPRF].

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

nonce = random(32)
context = SHA256(challenge)
token_input = concat(0x0001, nonce, context, key_id)
blind, blinded_message = client_context.Blind(nonce)

The Client then creates a TokenRequest structured as follows:

struct {
    uint16_t token_type = 0x0001;
    uint8_t 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.

**"token_key_id" is the least significant byte of the key_id.

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

The Client then generates an HTTP POST request to send to the Issuer, with the TokenRequest as the body. The media type for this request is "message/token-request". An example request is shown below, where Nk = 48.
Upon receipt of the request, the Issuer validates the following conditions:

* The TokenRequest contains a supported token_type
* The TokenRequest.token_key_id corresponds to a 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, which will forward the error to the client.

3.2. Issuer-to-Client Response

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:

server_context = SetupVerifiableServer(0x0004, skI, pkI)
evaluated_msg, proof = server_context.Evaluate(skI, TokenRequest.blinded_message, info)

The Issuer then creates a TokenResponse structured as follows:

```c
struct {
    uint8_t evaluated_msg[Nk];
    uint8_t proof[Ns+Ns];
} TokenResponse;
```

The structure fields "evaluated_msg" and "proof" are as computed above, where Ns is as defined in [OPRF], Section 4.

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

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

\[
\text{authenticator} = \text{client\_context.Finalize}(\text{context}, \text{blind}, \text{pkI, evaluated\_msg, blinded\_msg, info})
\]

If this succeeds, the Client then constructs a Token as follows:

\[
\text{struct} \{
\quad \text{uint16\_t token\_type} = 0x0001
\quad \text{uint8\_t nonce[32];}
\quad \text{uint8\_t context[32];}
\quad \text{uint8\_t key\_id[32];}
\quad \text{uint8\_t authenticator[\text{Nk}]};
\} \text{Token;}
\]

Otherwise, the Client aborts the protocol.

3.4. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted \(skI\) and \(pkI\), respectively, used to produce tokens. Each key pair MUST be generated as follows:

\[
(\text{skI, pkI}) = \text{GenerateKeyPair()}
\]

The key identifier for this specific key pair, denoted \(\text{key\_id}\), is computed as follows:

\[
\text{key\_id} = \text{SHA256}(0x0001 \mid\mid \text{SerializeElement}(pkI))
\]

4. Issuance Protocol for Publicly Verifiable Tokens

This section describes a variant of the issuance protocol in Section 3 for producing publicly verifiable tokens. It differs from the previous variant in two important ways:

1. The output tokens are publicly verifiable by anyone with the Issuer public key; and

2. The issuance protocol does not admit public or private metadata to bind additional context to tokens.
Otherwise, this variant is nearly identical. 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 4.4 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

*Issuer name, identifying 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 key_id computed as described in Section 4.4.

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

Given this configuration and these inputs, the two messages exchanged in this protocol are described below.

4.1. 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)
context = SHA256(challenge)
token_input = concat(0x0002, nonce, context, key_id)
blinded_msg, blind_inv = rsabssa_blind(pkI, token_input)

The Client then creates a TokenRequest structured as follows:

```c
struct {
    uint16_t token_type = 0x0002
    uint8_t 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.

**token_key_id** is the least significant byte of the key_id.

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

The Client then generates an HTTP POST request to send to the Issuer, with the TokenRequest as the body. The media type for this
request is "message/token-request". An example request is shown below, where Nk = 512.

```plaintext
:method = POST
:scheme = https
:authority = issuer.net
:path = /token-request
:accept = message/token-response
:cache-control = no-cache, no-store
:content-type = message/token-request
:content-length = 514

<Bytes containing the TokenRequest>

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

* The TokenRequest contains a supported token_type.

* The TokenRequest.token_key_id corresponds to a 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, which will forward the error to the client.

4.2. Issuer-to-Client Response

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:

```plaintext
blind_sig = rsabssa_blind_sign(skI, TokenRequest.blinded_rmsg)
```

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

```plaintext
:status = 200
:content-type = message/token-response
:content-length = 512

<Bytes containing the TokenResponse>
```

4.3. Finalization

Upon receipt, the Client handles the response and, if successful, processes the body as follows:
If this succeeds, the Client then constructs a Token as described in [HTTP-Authentication] as follows:

```
struct {
    uint16_t token_type = 0x0002
    uint8_t nonce[32];
    uint8_t context[32];
    uint8_t key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

Otherwise, the Client aborts the protocol.

### 4.4. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted skI and pkI, respectively, used to produce tokens. Each key pair MUST be generated as as a valid 4096-bit RSA private key according to [TODO]. The key identifier for a keypair (skI, pkI), denoted key_id, is computed as SHA256(encoded_key), where encoded_key is a DER-encoded SubjectPublicKeyInfo object carrying pkI.

### 5. 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 document 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 [I-D.ietf-privacypass-architecture].

### 6. IANA considerations

#### 6.1. Token Type

This document updates the "Token Type" Registry with the following values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Publicly Verifiable</th>
<th>Public Metadata</th>
<th>Private Metadata</th>
<th>Nk</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>OPRF(P-384, SHA-384)</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>48</td>
<td>Section 3</td>
</tr>
<tr>
<td>0x0002</td>
<td>Blind RSA, 4096</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>512</td>
<td>Section 4</td>
</tr>
</tbody>
</table>

Table 1: Token Types
6.2. Media Types

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

*TokenRequest: "message/token-request"

*TokenResponse: "message/token-response"

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

6.2.1. "message/token-request" media type

Type name: message

Subtype name: token-request

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 5

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
6.2.2. "message/token-response" media type

Type name: message

Subtype name: access-token-response

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 5

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

7. Normative References


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.

Authors' Addresses

Sofía Celi
Cloudflare
Lisbon
Portugal

Email: sceli@cloudflare.com

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