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 5 April 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
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
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
7. Security considerations
8. IANA considerations
   8.1. Well-Known 'private-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/private-token-issuer-directory" media type
       8.3.2. "application/private-token-request" media type
       8.3.3. "application/private-token-response" media type
9. 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

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.

This document uses the terms Origin, Client, Issuer, and Token as defined in Section 2 of [ARCHITECTURE]. Moreover, the following additional terms are used throughout this document.

*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.

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.
Field Name | Value
---|---
issuer-request-uri | 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)
token-keys | 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 2: Issuer 'token-keys' object description

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
</table>
token-type | Integer value of the Token Type, as defined in Section 8.2, represented as a JSON number ([RFC8259], Section 6) |
token-key | The base32url-encoded [RFC4648] Public Key for use with the issuance protocol as determined by the token-type field, including padding, represented as a JSON string ([RFC8259], Section 7) |

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, since the first such key is the most likely to be valid in the given time window. 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 the following (with the "token-key" fields abbreviated):
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.

A complete "token-key" value, encoded as it would be in the Issuer directory, would look like the following (line breaks are inserted to fit within the per-line character limits):

```json
{
  "issuer-request-uri": "https://issuer.example.net/request",
  "token-keys": [
    {
      "token-type": 2,
      "token-key": "MI...AB",
      "not-before": 1686913811,
    },
    {
      "token-type": 2,
      "token-key": "MI...AQ",
    }
  ]
}
```

A complete "token-key" value, encoded as it would be in the Issuer directory, would look like the following (line breaks are inserted to fit within the per-line character limits):

```bash
$ echo MIIBUjA9BgkqhkiG9w0BAQowMKANMAsGCWCGSAFlAwQCAqEaMBgGCSqGSIb3DQEBCBglghkgBZQMEAgKiAwIBMAQCAQ8AMIIBCAQEMKGAeyoJt1pj3n7xtqAPr_DhZAPhPc8kNnR2BzdZwPTTF7KFmsS5wt-mL01at0SC-cdBuIj6WYK80qvz0AyaBuvTvW6SKCh7ZPXEq-sq5I0nthREtrYKGo113oMVPVp3sy4VHPgzd8KdzTLGzOrjiUOsSFWbfj21iaVjXJ2vdwD-430wKucYjGe0Jwi8rWx_ZkcHtav0S67Q_S1ExJe16nyRzpuuD90Qm1nxfs1Z4PhWBzt9T3Tnd3a0kF5n0pIXD6bttmTekIw_8Xx2LMis0jfJ1QL99aA-muXRFN4ZUwORf7cAcCUD_-5fh9S34FmQBGwIDAQAB \
 | sed s/-/+/g | sed s/\//\//g | openssl base64 -d \
 | openssl asn1parse -dump -inform DER
0:d=0  hl=4  l= 338 cons: SEQUENCE
 4:d=1  hl=2  l=  61 cons: SEQUENCE
  6:d=2  hl=2  l=  11 cons: OCTET STRING :rsassaPss
17:d=2  hl=2  l=  48 cons: SEQUENCE
19:d=3  hl=2  l=  13 cons: cont [ 0 ]
21:d=4  hl=2  l=  11 cons: SEQUENCE
23:d=5  hl=2  l=  9 prim: OBJECT :sha384
34:d=3  hl=2  l=  26 cons: cont [ 1 ]
36:d=4  hl=2  l=  24 cons: SEQUENCE
38:d=5  hl=2  l=  9 prim: OBJECT :mgf1
49:d=5  hl=2  l=  11 cons: SEQUENCE
51:d=6  hl=2  l=  9 prim: OBJECT :sha384
62:d=3  hl=2  l=  3 cons: cont [ 2 ]
64:d=4  hl=2  l=  1 prim: INTEGER :30
67:d=1  hl=4  l= 271 prim: OCTET STRING ...
```

... truncated public key bytes ...
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. Issuance will continue to fail until the Issuer configuration is updated.

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 Issuer Private and Public Key, denoted skI and pkI respectively, used to produce tokens as input to the protocol. See Section 5.5 for how these keys are 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 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 32-byte 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:

\[
\text{server}\_\text{context} = \text{SetupVOPRFServer}("P384-\text{SHA384}", \text{skI})
\]
\[
\text{evaluate}\_\text{element}, \text{proof} = \text{server}\_\text{context}.\text{BlindEvaluate}(\text{skI}, \text{pkI}, \text{blinded}\_\text{element})
\]

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:

\[
\text{struct }
\]
\[
\begin{array}{l}
\quad \text{uint8}_t \text{ evaluate}_\text{msg}[Ne]; \\
\quad \text{uint8}_t \text{ evaluate}_\text{proof}[Ns+Ns]; \\
\end{array}
\]
\[
\text{TokenResponse;}
\]

The structure fields are defined as follows:

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

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

The Issuer generates an HTTP response with status code 200 whose content consists of TokenResponse, with the content type set as "application/private-token-response".
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 [OPRF], Section 3.3.2. If this succeeds, the Client then constructs a Token as follows:

```c
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:

```c
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 Issuer Private and Public Keys, each denoted $sk_I$ and $pk_I$, respectively, used to produce tokens. These keys MUST NOT be reused in other protocols. A RECOMMENDED method for generating keys is as follows:

```
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 $pk_I$, 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. Collisions can cause the Issuer to use the wrong Issuer Private Key for issuance, which will in turn cause the resulting tokens to be invalid. There is no known security consequence of using the the wrong Issuer Private Key. A possible exception to this constraint would be a colliding key that is still in use but in the process of being rotated out, in which case the collision cannot reasonably be avoided but it is expected to be transient.

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 an Issuer Private and Public Key, denoted $sk_I$ and $pk_I$, respectively, used to produce tokens as input to the protocol. See Section 6.5 for how these keys are 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 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 32-byte 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:
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:

\[
\text{struct }
\begin{align*}
\quad & \text{uint8\_t blind\_sig[NK];} \\
\quad & \text{TokenResponse;}
\end{align*}
\]

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:

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

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:

\[
\text{struct }
\begin{align*}
\quad & \text{uint16\_t token\_type = 0x0002; /* Type Blind RSA (2048-bit) */} \\
\quad & \text{uint8\_t nonce[32];} \\
\quad & \text{uint8\_t challenge\_digest[32];} \\
\quad & \text{uint8\_t token\_key\_id[32];} \\
\quad & \text{uint8\_t authenticator[Nk];} \\
\quad & \text{Token;}
\end{align*}
\]

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 Issuer Private and Public Keys, each
denoted skI and pkI, respectively, used to produce tokens. Each key
SHALL be generated securely, for example as specified in FIPS 186-5
[DSS]. These keys MUST NOT be reused in other protocols.

The key identifier for an Issuer Private and Public Key (skI, pkI),
denoted token_key_id, is computed as SHA256(encoded_key), where
encoded_key is a DER-encoded SubjectPublicKeyInfo [RFC5280] (SPKI)
object carrying pkI as a DER-encoded RSAPublicKey value [RFC5756] in
the subjectPublicKey field. Additionally, the SPKI object MUST use
the id-RSASSA-PSS object identifier in the algorithm field within
the SPKI object, the parameters field MUST contain a RSASSA-PSS-
params value, and MUST include the hashAlgorithm, maskGenAlgorithm,
and saltLength values. The saltLength MUST match the output size of
the hash function associated with the public key and token type.

An example sequence of the SPKI object (in ASN.1 format, with the
actual public key bytes truncated) 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. Collisions can cause the Issuer to use the wrong Issuer Private Key for issuance, which will in turn cause the resulting tokens to be invalid. There is no known security consequence of using the the wrong Issuer Private Key. A possible exception to this constraint would be a colliding key that is still in use but in the process of being rotated out, in which case the collision cannot reasonably be avoided but it is expected to be transient.

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.

<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>private-token-issuer-directory</td>
<td>IETF</td>
<td>[this document]</td>
<td>permanent</td>
<td>None</td>
</tr>
</tbody>
</table>

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

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: N/A
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

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: IETF

8.3.2. "application/private-token-request" media type

Type name: application
Subtype name: private-token-request
Required parameters: N/A
Optional parameters: N/A
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: IETF

8.3.3. "application/private-token-response" media type

Type name: application
Subtype name: private-token-response
Required parameters: N/A
Optional parameters: N/A
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: IETF

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: 39b0d04d372459288fc5ed89bb02c2aa4e06799201d6c518871d5181
149108ee3c91bed1bff3e3c1b87d53240a
pkS: 02d45bf522425cdd2227d3f27d2459d56308829252172d34e8469290c
21da1a4d42ca38f7beabdf05c974ae1455bf
token_challenge: 0001000e6973737565722e6578616d706c65205e8a52fc
daef25ca3f65448d04e0941b924e8264acccf66c5ad451d582b3000ef72696
7696e2e6578616d706c5
nonce:
6aa422c41b59d3e44a136dd439df2454e3587ee5f3697798cd05fafe73073b8
blind: 8e7f6d97b8a809b031b81a2e22d993d83bd5597c6a4dce1496ed2b1
7ef820445e3b2d3b2c4f54c5d1c956909
token_request: 0001f4030ab3e23181e1213f24351f5775983c678ce22efff9
427610832ab3900f2cd1d665907e8a6813cf0b5b886f4cc4979
token_response: 036bb3c359d88c3527cf9f081f6e3687b86e85c93c49
94c222408d4903722a19ff272ac97e3725b947c972784ebff66eb9ea54336e43
34a9660212c0c85fbbadfbf491a1ee2446fc3739377fcc4dc510592b7c760110e
e1e2c227db0e01c9f482c00ffaa0db2e2fb58c32d2b1db69ff920a528e68d
93b3c483848e57c30542b8984fa6bfeeced67d154d53eda
token: 00016aa422c41b59d3e44a136dd439df2454e3587ee5f3697798cd05f
afe73073b850137b0944089dc462802af545ef63809581ee6f57890a12105c283
6816914bf260d0792bf7f46c9866a6d37c3032d871441587f5f6903d7fb07e
253b2f4e9a835d65288b444f73789e7edc907158b1c19792fd87375c0a7a9d
3d92540374f70733be207e721da3af40edeb

// Test vector 2
skS: 39efed331527c4d4df9722ab5cd35aeafe7c27200cfa2e2edbd298dc3
b12bc829afcc46558af1e2eacc5307d685
pkS: 038017e005904c6146b371096c2a72b95a183aaa9ed91b8d8fbeb0d393
f68033284d175e7df8984975cd67a96e8bfb4e
token_challenge: 0001000e6973737565722e6578616d706c6500000ef7269
67696e2e6578616d706c5
nonce:
7617bc802cfd5d74722ef7418dbb4f2c88403820e55f7e7ec073190c29d665
blind: 6492ec05072fa18d03d569c4246362dffe2621af95a10c3bb0109e0
f705b0437c42553272e0a5266ec379e7015e
token_request: 000133033a5fe04a39da1bbf68ccdeecd191747dd525462e
5a50a6ba53b42aa148e4e443a2e1c7f3fd5ff28a1c7cf1aeac5d
token_response: 03b28fcc6d248080d669c5cc6c88b056355c6e8e1bcfb3746cf
b9a9248a4c056f2a84876ef99a8b6b281d50f852c6fa868f4ca13c79c5bf
bdcf8bfc3926e18c712f93a4887d86dd4a45e70f5af169aa757208786f9536
92af8ff9cda72f281e4e35686e848225367946c70db097e18e3cba16193987b
c10bed3ef54c4d036c17cdd401b1130e60d7a9a27e0d
nonce: 00017617bc802cfd5d74722ef7418dbb4f2c88403820e55f7e7ec073
190c29d665c994f7d5cd2c97b01348e8eb6e6d8f9dca9a6581fb09125fdef1
34bd5a62a116477bc99e1a205cc95d6c92335ca7a5e7163b2ac020bd231c66
97f12333ef438d0d001ba5ace0f8eb483dc04cd625789b5652921cd2698c
45ea74f6c827b4e190000fa5bd039866f562
// Test vector 3
skS: 2b770959b62b784f1496ae828f65e6caeb6eefe732c86e9ae5e818c055b3d7ca3af5f2beeca859a62ff7199d35cc
pkS: 03a0de1bf3fd0a73384283b64884ba9fa5dee190f9d7ad4292c2fd49d8b4d64db6459df67f5bd7e62d475c78493ae8d
token_challenge: 001000e6973737567522e657861d6706c65000017666f2e657861d706c652c6261722e657861d6706c65
nonce: 87499b5930818d2d83ecebf92d25ca0722a11b800bbfd950537c28aa7d3a9df
blind: 1f659584d62ba15f44f3d88782e5fe4273151b85dfbfaea4253ebea306104d9b73c78141c2360e85a09420c0f0f
token_request: 0001c8042f1a9f3aac21900f3079d6809437a2b9b449c3ec645f84fbc6505ada154c258c8ecd42bdcf574daca65db671908
token_response: 03c2ab925d03e779b3a4df66b50521039f620359e1424491b8143ce6a3e5298b265662c33254e11be7277233e1a34570f7a4d1d2931e4b5ff8829e27aaf7eb2c7f9a655477d71c01d5da5e4f4d0d6b76b204710fe0f25a96e6b5095af610505978c1b8334d615008c6370556ed0c6671e69776c09a92b5ac8480475d0867c87817bdf69f1443002b1a8ea7a52
token: 00187499b5930818d2d83ecebf92d25ca0722a11b800bbfd950537c28aa7d3a9df
nonce: 02f0a206752d555a24924f2da5942a1bb4cb2d83ff473a9b262a3e98e820cd43
blind: af91d1dbcf6b46baecde7be6b035b8f7e5629199ca19c7f934b8607b90def2bc3e035ad3ce2d9f0a01efa0561d0
token_request: 001a503632e8bb003ed15b6de4557c0477f81a58668413332ad3ad7f940162d9f439ad1e8cd67d94ae7c05bc958d134
token_response: 032018bc3f180d9650e27f72de76a90b47e336ae9cb058548d851c7064fa8075d96346c15cb3d9d8083c6fb57216544c6a815c37d97269e129c9513ca20d048f4a4ed1b0f71b29e219a71874a93e53ab2f47328271d1e9cbeefc197af599a6825051fa1c6e55450042f04182b86c9cf42774a9f16849396c051fa2701e281a86e64a926a4063fe1722dd7
nonce: 000182f0a266752d555a24924f2da5942a1bb4cb2d83ff473a88b2f777f

// Test vector 4
skS: 22e237b7b9837d7774e4495aff2fc1e10422b1d955192e0ffbf2b76b18fba625fc8bb3959a9113da49c495a48fbbf777f
pkS: 028cd68715ca2ddb9b22bd0d176a04a2b9f2b0a47db65e5e763e23744fe14d74e374bc9379e53c8aa765ee21
token_challenge: 001000e6973737567522e657861d6706c65000000
nonce: 02f0a266752d555a24924f2da5942a1bb4cb2d83ff473a9b262a3e98e820cd43
blind: af91d1dbcf6b46baecde7be6b035b8f7e5629199ca19c7f934b8607b90def2bc3e035ad3ce2d9f0a01efa0561d0
token_request: 001a503632e8bb003ed15b6de4557c0477f81a58668413332ad3ad7f940162d9f439ad1e8cd67d94ae7c05bc958d134
token_response: 032018bc3f180d9650e27f72de76a90b47e336ae9cb058548d851c7064fa8075d96346c15cb3d9d8083c6fb57216544c6a815c37d97269e129c9513ca20d048f4a4ed1b0f71b29e219a71874a93e53ab2f47328271d1e9cbeefc197af599a6825051fa1c6e55450042f04182b86c9cf42774a9f16849396c051fa2701e281a86e64a926a4063fe1722dd7
nonce: 000182f0a266752d555a24924f2da5942a1bb4cb2d83ff473a88b2f777f

// Test vector 5
skS: 46f3d4f562002b85ffcfdb4d06835fb9b2e24372861ecaa11357df129f9
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: 002aa72019df951df97021ce63876fe8b0a2d1c31a12b0a2dd150
8d07827f05596f9643b4cfda5196d4aa866eb5368834f4f06de46950ed435b3b8
1bd036d44ca572f8982a9ca248a3956186322d93cac147266121d9eb5632c971f
71cd2788db6a1b533d7294b5e900fa55377dd3eb33ce4e08c9676d1e5358fd
184b0e06c637174f5206b14c7b0e724e6b6256717e5aa2ed940c51ca433d302
b23bc524681d04148f4e267b7c1b60e838f9d087629f7904cc724bcb0d
89b45c939666fcd75f9a9cd2df37f855f6f4c584d84f716864f5466966d6290e
5bd41a18119de84ff9740ba5093b2a22422d6b91eb745c084758974642a429782
01543246dddb8030ea8e722376aa82484daca9610a8f7e08e1396156462e1a03
e40ea7e128c990a911ce7080666cb2018330101ced4e910fc8e27a1be467ff786
71836a50825712a345e4e0ae2180a434bd1037713466347a8ebe46393da1970

// Test vector 2
skS: 2d2d2d2d2d424547494e2050524956415445204b45592d2d2d2d0a4d9
49457651494241444f4e26767b716866b6947397730421545641415343424b6
377676536a416745414d469424151444c477531726176583176334420a4f6b
7a3e717957355379356b6f6a41363545556b66717444774e38366a42b5a4a74f76
57245526b91a4c572877673d6453327976132633616b4745714c756440a556a
35743561496b3172417643655844644e4a590432325055707851436e699396e6
b492b6d67725769744449444873189731375866e65079596784f530a646f
655856383546f3f1a4752b23973365d586d34516a75513954596149713837
1724456567a503575812b2e4d6363793223698676324c76d423990a6a41
355334475666325a6c74878954736f4c36487327a7a58696a4e394637486271656
76f75396765b52d4584645352f2ba4a3956595a634a734a624c756570480a544f
7253a4d49d92e358b535814d4166414f45a4a454726d6dd443663566672f43473
475676a79486e4e51383733414e4b6a55176d3676574574413872514c620a4530
742b496c706641674d424145436767454147a3e436264a6931a50643584d6
b562b434c667f65531232b7266486e762764266502f5636448778275690a3270
31615358a596962653645532bd6d22f4d4655646c4850674147c731785134576
5726636e444d4736866c784c575353684732473663386f364759302a6359
366ff777424476632616847b5b5039656b2395303864a567347354736
1556e484a5852376967838456a6c666f4c6e7245516536658578734d10a2620
64487846444424d64476656777674b6f6a4f6a7063c2f3986d6555793756422
f3661326c72656576c766a362f326e4b34b745937734743647516c47460a786
4142615757835843a452f4513334c762b242656626717449397315a776a7a
265455681458356437872793251564d515751696e57684174364d7154340a5342
5354726f6c5a7a7772716a5834d504a393175614e4d6458474c63484ac93267
358a7a6734b5351a4b267514476673735055557641395a325a583953850a6d49
874d5422e4a66754559675b4b413179576e3155444ea6356a71682b7
a562f376b33947686b6830514633162713630654c39394749536914f0a535b
4f574d39454b5f2b784151326261b314d664f5931472b386a7a425855704273
9346b35338379586d466e76946776373942a35a6835666b55710a5732
306f536266b686a5264537a48326b52476972672b553774b426751445a4a6d
7e7279324578612f3345713750626f737841504d69596e6b354a415953470a7932
7a30a5a375455622b75485142f2b7504d376e433075794c494d44396c61544d4
8776e3673372f4c62476f45503157256776059482f4321346b2f526e360a6675
7724ae363249f6f397463392b41434c745542377674476179332b6752775975434
33262356564386c49696657745466b56139683075445357284174567330a6e35
6b79613251976514a5b2674646a75467a4f5a742b7467596e576e515545675385
token_challenge: 0002000e697373756572e6578616d706c65000000
nonce: 494dae41fc7e300c2d09990afcd5d5e1fc95305337dc12f78942c453400bf8e60
blind: 097cb17bcdedcfe058dff5c4e517d1e367ab8f46252b1ac1933ba378c32625c0db69f5655c2003bf9e75810796cd63675b223cf3162c571806d5e69584
cf36ecad829e74369ada38a095e36012c912b31ccde7425f9346e353f3b17525be3a8df9219da615433aaaa4505f8f218c471dfb1c12fb304158e29b6ed353bc07
9e23f1e6173c5dec4545840bbe58e5ad37cbeea0a10dca59df2781589d27c34108477b52c0d32a1370c17f703941fbb1a060a6794e7e2758709c9bf8f021eec79229b491b6ea68c1a14794e6486e6e4e00f0ae913797067aa0826366c3193e103b05653c73b52d7f825a185dccb806da700db9f53abb848554b7d4f7c28f3salt: 49912979f1b5f28e5b0228b1a328df74319dce7bdaf4521cebed0cf042a2de852c9db5b964af6493cc282504240
token_request: 0002082448d0827ca8c620f8b14caded9a199ba388c8d8541e962f68a0071535d958d18494af6dcb11ca4da8c8b33864f5a8f623b93b7c5d6348594e1a75479048a72c0ed7197b07506c09a7eb6ed3582f572f38c6f0cde11a52c5c6e7b234335b60209d661f4f0f323c9aa54307db966d4457c37542b65bb183ddeafca914fc74831698b5d2f498ee3d165685f49a8d86e39fe6c4b7ec678f5250908d25e5873c69b42236312a1421cadd6f64e90973cb9a7a3e827a0e742470f002a9d06e9470dbfd73df0ccbb96c10bb02af0df7de791ec4
11ff8e1b4929e59f3c3f19de9bda29a6d968b43083b5d4242f3448d76ada08bb8014f70b97e19
token_response: c2746ff644cfbb28a2a2c19395fa19dfb61fd1d35aa837844fbf9be66c253e64e69f53ae6dccc9f8b4833b1b5e58f571134a34f2425499c3e73419549c2911cf49f2f68f3a996d47f71e8d8d6fc5b1c074bf74fa59de4cfb32f5f08d4eaa45492f0279c3b1a8d852698edbe1651eb8e09eb223a27386cf0eb2f6a8260235edb36cf433d5a18100829b61662843b257f8fc941ea3a97eb76761b7082e09397837f74b40f0c838bce8a7f7242089dd5561f57735926cbad219fc9fefe8ae995f16ca194b7f6018c06ee02267e7267bb996432dc76973819da80e3e86947b0a4b363da972dafaa3d0e0144b325f02c67999699dc3e51390d54bc0b8c
token: 0002494dae41fc7e300c2d09990afcd5d5e1fc95305337dc12f78942c453400bf8e60
// Test vector 5
skS: 2d2d2d2d2d24547949e26505249564154545206b45592d2d2d2d20a4d49494576514942414441e4e267b718666b9473977930a24154564145343424b63776767536aa4167454141e6f94924151444c4775317261705831763343420a4f6b7a3871795735573565b6f6a41303543554b6671444774e383664a242b5a4f764
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