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 28 March 2024.

Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in
1. Introduction

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

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

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 \[\text{ARCHITECTURE}\].

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 \[\text{RFC2119}\] \[\text{RFC8174}\] 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 \[\text{ARCHITECTURE}\]. 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 \[\text{TLS13}\]. 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 ([\text{AUTHSCHEME}], Section 2.1) and use the issuance protocols to produce corresponding Token values ([\text{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 URL, or a URL relative to the directory object) 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>

Table 1: Issuer directory object description

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>token-type</td>
<td>Integer value of the Token Type, as defined in Section 8.2, represented as a JSON number ([RFC8259], Section 6)</td>
</tr>
<tr>
<td>token-key</td>
<td>The base64url-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)</td>
</tr>
</tbody>
</table>

Table 2: Issuer 'token-keys' object description

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

Beyond staging keys with the "not-before" value, Issuers MAY advertise multiple "token-keys" for the same token-type to facilitate key rotation. In this case, Issuers indicate preference for which token key to use based on the order of keys in the list, with preference given to keys earlier in the list. Clients SHOULD use the first key in the "token-keys" list that either does not have a "not-before" value or has a "not-before" value in the past, 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):

```bash
$ echo MIIBUjA9BgkqhkiG9w0BAQowMKANMAsGCWCGSAFlAwQCAqEaMBgGCSqGSIb3DQEBCBglghkgBZQMEAgKiAwIBMA0CAQ8AMIIBCgKCAQEAmyoApBzdZwPTTF7KFKms5wt-mL01at0SC-cdBuIj6WYK80vz0AyaBuvTvW6SKCh7ZPXEq-sq5I0nhtREtrYKGO113oMVPVp3sy4VHPgzd8KdzTLGz0rjJiU0sSFWb7f2i1aVjXJ2VdwdS-430wkucYjGe0Jwi8rWx_ZKcHtav0S67Q_S1ExJe16nRzpuuID90Qm1nxfs1Z4PhW8zt93Tn3aok1F5n0pIXD6bttmTekIw_8Xx2LMis0jF1J9QL99aAmuXRFN4ZUwORrF7cACUDP_-5fh9s3FmqBGwIDAQAB
| sed s/-/+/g | sed s/\//\//g | openssl base64 -d \n| openssl asn1parse -dump -inform DER
0:d=0 h=4 l= 338 cons: SEQUENCE
4:d=1 h=2 l=  61 cons: SEQUENCE
 6:d=2 h=2 l=  9 prim: OBJECT :rsassaPss
17:d=2 h=2 l=  48 cons: SEQUENCE
19:d=3 h=2 l=  13 cons: cont [ 0 ]
21:d=4 h=2 l=  11 cons: SEQUENCE
23:d=5 h=2 l=  9 prim: OBJECT :sha384
34:d=3 h=2 l=  26 cons: cont [ 1 ]
36:d=4 h=2 l=  24 cons: SEQUENCE
38:d=5 h=2 l=  9 prim: OBJECT :mgf1
49:d=5 h=2 l=  11 cons: SEQUENCE
51:d=6 h=2 l=  9 prim: OBJECT :sha384
62:d=3 h=2 l=  3 cons: cont [ 2 ]
64:d=4 h=2 l=  1 prim: INTEGER :30
67:d=1 h=4 l= 271 prim: BIT 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:

```c
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:

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

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

```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])).

The Issuer generates an HTTP response with status code 200 whose content consists of TokenResponse, with the content type set as "application/private-token-response".
HTTP/1.1 200 OK
Content-Type: application/private-token-response
Content-Length: <Length of TokenResponse>

<Bytes containing the TokenResponse>

5.3. Finalization

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

```python
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:

```python
struct {
    uint16_t token_type = 0x0001; /* Type VOPRF(P-384, SHA-384) */
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

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

5.4. Token Verification

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

```python
server_context = SetupVOPRFServer("P384-SHA384", skI)
token_authenticator_input =
    concat(Token.token_type,
           Token.nonce,
           Token.challenge_digest,
           Token.token_key_id)
token_authenticator =
    server_context.Evaluate(token_authenticator_input)
valid = (token_authenticator == Token.authenticator)
```
5.5. Issuer Configuration

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

\[
\text{seed} = \text{random}(Ns) \\
(skI, pkI) = \text{DeriveKeyPair}(\text{seed}, \text{"PrivacyPass"})
\]

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

\[
\text{token_key_id} = \text{SHA256}(.\text{SerializeElement}(pkI))
\]

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.

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 skI and pkI, 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.

```
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>
```
completes the issuance flow by computing a blinded response as follows:

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

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

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

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

HTTP/1.1 200 OK
Content-Type: application/private-token-response
Content-Length: <Length of TokenResponse>

<Bytes containing the TokenResponse>

6.3. Finalization

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

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

The \text{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:

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

The Token.nonce value is that which was sampled in Section 5.1. If the \text{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 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:
$ cat spki.bin | xxd -r -p | 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=  33 cons: OBJECT :rsassaPss
 17:d=2 hl=2 l=  48 cons: SEQUENCE
 19:d=3 hl=2 l=  19 cons: cont [ 0 ]
 21:d=4 hl=2 l=  17 cons: SEQUENCE
 23:d=5 hl=2 l=  15 cons: 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=  14 cons: OBJECT :mgf1
 49:d=5 hl=2 l=  11 cons: SEQUENCE
 51:d=6 hl=2 l=  11 cons: OBJECT :sha384
 64:d=4 hl=2 l=  13 cons: INTEGER :30
 67:d=1 hl=4 l= 271 prim: BIT STRING
... truncated public key bytes ...

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.

7. Security considerations

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

8. IANA considerations

This section contains considerations for IANA.

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

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

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

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

*Value: 0x0001

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

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

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

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

*Public Verifiability: N

*Public Metadata: N

*Private Metadata: N

*Nk: 48

*Nid: 32

*Reference: Section 5

*Notes: None

8.2.2. Token Type Blind RSA (2048-bit)

*Value: 0x0002

*Name: Blind RSA (2048-bit)

*Token Structure: As defined in Section 2.2 of [AUTHSCHEME]
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: 39b0d04d3732459288fc5edebf9b02ca2a42e06709f201d6c518871d5181
149106ee3c919bed1b9f53cf1cb87d53240a
pkS: 02d45bf522425cdd2227d3f2f7d2459d563008829252172d34e48469290c
21da1a4642ca387febadf05c974ae1455bf
token_challenge: 001006e697373756752e6578616d706c652058a52fc
daee25ca3f65448d04e404bf1924be8264acffcf6c5ad451d582b3000e6f72696
7696e26586167d06c5
nonce:
6aa422c41b59d3e44a13ded349df2454e3587ee5f3697798c05f3afe73073b8
blind: 8e7f9d08b9b0a0809b810a2e229d903d83bd597c6a4dc1496ed2b1
7ef820445ef3db2b13c45f5208d68dd
nonce_request: 001f4030ab3e23181d1e123f24315f7577983c678ce2eff9
427610832ab3900f2c1d2d629a07e8a8613cf0b5b886f44c4979
token_response: 036bb3c35c97d88c3527cf9f08f1fe63687b8678593c948
ee2c22248d490372a219ff272ac9e3725b947c972784ebf8e6eb9ea54336e43
34a966062120c08f5bfadbf491a1c2446f3c379377fccc45c10592b2c769011e
e1e2277db019c4f82c09f8a0d6ef0be2f598c328ed21be94e92a528e68dd
9b3c248384e5c730542b8984fa6bced67d15d53eda
token: 001616422c41b59d3e44a13ded349df2454e3587ee5f3697798c05f3afe73073b8
3d9254a37f47073be2f71e721da3af40ede
// Test vector 2
skS: 39efed331527cc4ddff9722ab535aafafe7c72520b0cfa2eeedbbdc298d3c
b12bc8298afcc46558af1e2eeacc5307d865
pkS: 0380i700590946146b371096c2a72b95a183aaa9ed951b8d8bf1ed9033
c68033284d175e7df89849745cd67a866bfb4e
token_challenge: 00100e697373756752e6578616d706c650000006f72696
7696e26586167d06c5
nonce:
7617bc802cbdbd57d4722ef7418dbbf4f2c88403820e55f7e7ec0d39c0aa6926d5
deblind: 6492ee50072fa1d305d6c4246362dffe2621af95a10c033bb0109e0
f705b4037c4525327e5a05266ec379e7015e
token_request: 00133033a5f0e4a39ada1bfb68ccedd1917474dd525462e
5a90a6b5b42aa1a486fe443ae2e17f3fd5ff28a1c7cf1aeac5d
token_response: 023bf8c0d624880d669c5cc6c88b056355c6e8e1bcbf3746cf
b9ab9248a4c056f2a4a876ef998a8b6b281d50f852c6fa86ff4ca13c79cbb5f
dbfbb8f3c926e1ac7c12f934a887d86dad4a4e5be70f5a169aad75720887bb69536
92af8f11a9da72f281d4e3568e84822536794c70db90e718e3ca16193987bc
10bede3e5f45d4036c17cdd4015bb113be60d7aa927e0d
nonce: 0017617bc802cbdbd57d4722ef7418dbbf4f2c88403820e55f7e7ec0d39c0aa6926d5
deblind: 6492ee50072fa1d305d6c4246362dffe2621af95a10c033bb0109e0
f705b4037c4525327e5a05266ec379e7015e
token_request: 00133033a5f0e4a39ada1bfb68ccedd1917474dd525462e
5a90a6b5b42aa1a486fe443ae2e17f3fd5ff28a1c7cf1aeac5d
token_response: 023bf8c0d624880d669c5cc6c88b056355c6e8e1bcbf3746cf
b9ab9248a4c056f2a4a876ef998a8b6b281d50f852c6fa86ff4ca13c79cbb5f
dbfbb8f3c926e1ac7c12f934a887d86dad4a4e5be70f5a169aad75720887bb69536
92af8f11a9da72f281d4e3568e84822536794c70db90e718e3ca16193987bc
10bede3e5f45d4036c17cdd4015bb113be60d7aa927e0d
// Test vector 3
skS: 2b770959b62b784f14946ae828f65e6caeba6eefe732c86e9ae50e818c055b3d7ca3af2b2eeca859a62ff7199d35cc
pkS: 03a0de1bf3fd0a73384283b648884ba9fa5dee190f9d7ad4292c2fd49d8b4d64db675496f67ff5bd7e626475c78934ae8d
token_challenge: 001000e9737375675722e6578616d706c6500017666f62e6578616d706c652c6261722e6578616d706c65
nonce: 87499b5939018d2d83ecef92d25ca0722a11b80dbbbfd950537c28aa7d3a9df
blind: 1f655958426ba15f44f3d8878e2e5fe4c7315b1b85dfbfaeaa4253ebba3061c4db97c37871c4142360e85a090420c0eff
token_request: 0001c8024b10a9f3aad21090f3079d6809437a2b94b44303c7e645f849bc6c505da1e154c258b8e4d422bdcf7574caca656b71908
token_response: 03c2ab925d03e779b34a4df6e6b50521039f620359e1424491b8143d06a3e529b2b5bb6db3c3256411be7277233e1a34570f7a4d1d2931e4b5ff8829e27aaf7eb2ccf9ab655477d71c0d1d5a5aee4f4dd0675b0924710ef025a9e6c6b50a9a5f6b771d6b08500cf6370556ed060c6d716976c09a92b5ac848047506d7867c8717bf69f1434002b18ea7a52
token: 00187499b5930918d2d83ecef92d25ca0722a11b80dbbbfd950537c28aa7d3a9df
nonce: 02f0a206752d555a2b4924f2da5942ab4cb2d83ff473aa8b8b2ca3a99e82bb02c4d3
blind: af91d1df1799f1949df455874278b8ae7e26c513c3261cddbe5722051245e4c9c911ddc0bb66575bf8ff3f0ce8accbb3a7b8e41d23a172871be4828c54582d87bc7cfc58bcedc188e0c845b000c317ed75312274a42b106ed23db8a168df2f021c23925d72cd147cd7588c03845da0d41a326

// Test vector 4
skS: 22e237b7b9837d77474e4495aff2fc1e10422b1d955192e0fbbf2b7b618fba625fc94b599a911da49c495a48bf7ff7f
pkS: 028cd68715caa20d19b2b2d0d17d6a0a42b9f2b0a47db65e5e763e23744f1e4d7437e47bbc939705e3a8a765e21
token_challenge: 001000e9737375675722e6578616d706c65000000
nonce: 02f0a266752d555a2b4924f2da5942ab4cb2d83ff473aa8b8b2ca3a99e82bb02c4d3
blind: af91d1df1799f1949df455874278b8ae7e26c513c3261cddbe5722051245e4c9c911ddc0bb66575bf8ff3f0ce8accbb3a7b8e41d23a172871be4828c54582d87bc7cfc58bcedc188e0c845b000c317ed75312274a42b106ed23db8a168df2f021c23925d72cd147cd7588c03845da0d41a326

token: 00187499b5930918d2d83ecef92d25ca0722a11b80dbbbfd950537c28aa7d3a9df
nonce: 02f0a266752d555a2b4924f2da5942ab4cb2d83ff473aa8b8b2ca3a99e82bb02c4d3
blind: af91d1df1799f1949df455874278b8ae7e26c513c3261cddbe5722051245e4c9c911ddc0bb66575bf8ff3f0ce8accbb3a7b8e41d23a172871be4828c54582d87bc7cfc58bcedc188e0c845b000c317ed75312274a42b106ed23db8a168df2f021c23925d72cd147cd7588c03845da0d41a326

// Test vector 5
skS: 46f3d4f562002b85f4cffdb4d06835fb9b2e24372861ecca11537fd1f29f9
ed26e4715549ccedeb39257f095110f0159
pkS: 02fbe9da0b7cabe3ec51c36c8487b10909142b59af030c728a5e87bb3b30
f54c06415d22e03d9212bd3d9a17d5520d4d0f
token_challenge: 0001000e6973737565722e6578616d706c65205de58a52fc
daef25ca3f65448d04e040fb1924e8264acfc6c5ad451d582b30000
nonce: 9ee54942d8a1604452a76856b1bfa1cd608e1e3fa38acfd9f13e84483c90e89
blind: 76e0938e824b6cda6c163ff55d0298d539e222ed3984f4e31bbb654a8c
59671d4e0a7e264ca758cd0f4b533e0f60c5aa
token_request: 0001e10202bc92ac516c867f39399d71976018db52fcab5403
f8534a65677ba9e17d9b1d01767d137884c86cf5fe698c2f5d8e9
token_response: 0322ea3856a71533796393229b33d33c02cd714e40d5aa4e0
71f056276f32f89c09947eca8ff119d940d9d57c2fcdbd83d2da494ddeb37dc1f6
78e5661a8e7bccc96b3477eb89d708b0ce10e9ea1b5ce0001f9332f743c0cc3d47
48233fe6a6d152fae7844821268eb96ba491f60b1a3a848849310a39e9ef59121
669aa5d5bb4b4deb532d2f907a01c5b39efaf23985000
token: 00019ee54942d8a1604452a76856b1bfa1cd608e1e3fa38acfd9f13e8
4483c99e89d4380df12a1727f4e2ca1ee0d7abea0dfb1ea9506507a4dd618f9b8
7e79f9f3521a7c9134d6722925bf622a994041c0db1b082c0f1309af32f0ce00ca
1dad63e1b03747a8a5c3b46c7c2853de5ec7af8cac7cf3e089cec9e9ed3ff05c
d24504fe4f6c52d24ac901471267d8b63b61e6b
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.
// Test vector 2
skS: 2d2d2d2d2d2d424547494e20250524956415445204b45592d2d2d2d0a4d94945675194241444444e2676b71686669479377304221545464111434246b677676736a416745414116494241514444c477551327162705837136334426a4f6b7a87179753735539735b66f6a1430354355b66b771447474e39366a42a4b5a4f747657245526b49314c527876736453327976132633616b47457141c756b44a0a55a35743561496b3b172417643655844644e445039432325055707851436969396e6b4926d67725769744444494871386139793137566e6c5079596f784f53a0a64f6558563835464f31a475b2b239733656586d34516a7551394559614971383371724450657a503575812b246e3d6373923226968676324c766d42390a6a41355334475666325a6c74878594736f4c3648723777a58969ae394637487627165676f75396765b52d584645352f2b4a3956595a634a734a624c75657849a0a44f72535ad4949492b5835814d166144f454a4547426d6d4430683566672f43473475676a7986e4e51383733414e4b6a55716d3676574574413872514c620a4530742b496c706616174d4214145436767454147a3461267a69316a506345384d6b562b43a4c6676563312b2764686e7626724665620f5663448778275690a327031615358a596962652654532b4d622f4d46555646c485067414cc773178513457657263661444373686c6784c57535636847327366386f364759320a6359366f77742447636216847b55b503456b2639530584ca5763473547361556e48a4a8523769e7834655a6c66f4c6e7245516536685778347d10a6230644878464484424d6447666577767b466f6a4f6a670352f39386d4555793756422f3661326c72656576c766a632f326e4b34b745937734743674516c47460a7a41426157538034535425f45131334c762b242656662717449397315a776a7a26545581453863473872793251564d515751696e57684174364d7154340a53425354726f6c5a777272a65384d59a4393175614ed6458474c63484c49323673574b5351b4267514476637735055557641395a325a58395350a6d49784d5422464567a66255975b4b43179576e31554d44e63556a71682b7a652f376b33794768b6830514633162713630654c393047495369414f0a54b4f574d39454b6f2b784151326261b314d664f5931472b386a7a4258557042733946b5353338759866846e68e79467763739424a385a683566b55710a53720306f53626b6b68a5264537a48326b52476972672b553774b426751445a4a4d6e7279324576861cf343571375626f737841504d69596e6b354a415953470a7932a305a37545562b7545851f4f2b758043d37e6433075794c49443496c6154d448776e363732f4c62476f455031575267760f5948f2f4231436b2f526e360a66757724e363249f6397463392b41434c745542377674476179332b675277597453433262356564386c4969665774546b65130683705445357284174567330aae356b796132513976514a4267464a75467a4f5a742b7467596e576e515545675385
bd103ea4617d2472cf58da3381e52e5be60f4acbf685e280648cef21211a796ec
d05ecbd2aa1046c40950a0ac4c4e7dd4b8c19e50408849a15667b45895b6e92
salt: c847b5d0f9a101a1e09954ac9f3e6ed6000af58936295ad2e54274e13e64
0d5f9732d07530c9c1c03c6668f30470c77ac
token_request: 00e2080f6bd84fba1822c577c8cd670f1136ca107f84dd9d
405d5ed22ad5da975538f031433ba4a2668899732927e2e929824c132389a12
2f40b639b083d6fcbced7a55fb1bd8b536d2dcaabe2f6dcda87073e6565b8a7dfd
783913a95f97d798de0cfc26c29e99a7e874a3e3355eacbd445ef6779f9a6a78
5b05ac352fdd51a116cf2be1d8e38b0bafac6a3d5a88c99f7477ce908e86b335
62691f540e3e88565292dc172cc2f78ce0f9b53312a5f2dc918bb1d1c9d9de65901
7ca9b9080c1cc755cb5437989364dc92f0e8e1af8f66d31451f6e0b263d68a41d
e1a3f9e925da5c4ca0760fc4ca28b3317e939f6573442c6d03be17cd141fa82
60d3d2d134c6b
token_response: 2dd08ce89cf4f62bc236ab7b75266e13c57c570345e3280eb
aa107537c4cbeca57f99f716950440628ea23e7db5c9c68d4f9a9656cbfb0cbbf
fb989516b1fd19a9d69cc52a28890bdcf782f56afadad88b6e861a74709c91
891c89e4293f3f7978dbd41cc8b5c68820de3d69fd3e2b22b890512245986
6aadd1eab382d293c3996f9c9b375390e19b050d5656c2b084309d98a816
14f631bf827e4588413b4a0c6d94e942fa134790b396cb71e3ed3b5575bd
0734e726fa79abcda8649703b162e629b749081d4383e0d4f825dced098c43
d3ba81c0282dd833a4fc24961f60e118d4421dce5b611d53e9ca96156a5259bf
a9af6be7
token: 0029e7a22bdc5d715628434ecb07e5ba5f3622f776a17a6d9175af
1592df6e710042e4454ac4dd5acb8f6e65c4d8dd47504f73f7463507ef9a6a472
d727747fa572988a9c2a8a39618632293a14726621d2eb6532c971f1f
71cd270815b01b0bccd5f5e9c856d2e9ffafeb0a07d33cd2d5452f6bebb01591b9
973e0dc9180eaeb1824243758df9b0ac95ae9d4af9f74ec93644ae6cd7068e
a76e2295b9b5e383ed3a9856e9f61d8afdf4c5edc5db3e4297cf39900abaca
71e3cdd6c07a437daae7ed27eb681178fb7ce5fa5dd63781cc64ace410f441c0
34b0a5cc873a2eca875e8b38c92ab563635c4ff4fa35df82ef19ef7da75a
11a50aa32e32a12bd4da41ec0a7ec7f451ca586f5b91003fcbb9ff5fa2408
c28d6807737d03da651ea9bfafcc2747a6830e19a1d160fcd5c25d2f79dad86a8
b3de8e926e08ca1addced72977ff7b56398ef59c26e725df0a976a08f2a936ca42
// Test vector 4
skS: 2d2d2d2d2d424547494e2e05052495614545402454552d2d2d0a4d49
49a4576514924414414a4e2676b716866b6947397739042415145641415343424b6
3776767536a41674541416f949242151444c4775317261705831736334420af6b
7a3871795739573966b6f6671474477e43836a424b5a4f764
57245526b49314c527876734d4653327961326333616b4745714c756b440a556a
357435614966b317214763655844464e4503443242550570851436696936e6
b492b6677576944444444413816139731375866ec5079596f784f530a646f
6558563835464f314a752b2d62937336356d566d34516a75513945596149713837
1724450567a56035758712b52a4ed63637932326968676324c76d24390a6a41
35533447566325a6c74785954736f4c364872377a58e69a4e394637486271656
76f75967654b52d584645352f2b4a3956595a634a734a624c75657480a44f
72535ad4948502b5358514d40664144f454a45472462dd4406835666672f43473
47567a679486e4e51383733414e4b6a55176d367657454413872514c620a4530
742b496c706641674d4241414543676745414c7a4362647a69316a506435838d6
token_challenge: 0002000e6973737565722e6578616d706c65000000
nonce:

949da4efc7e306c2d99990afcd5d5e1fc95305337dc12f78942c453400bfe8e6
blind: 097cb17bcedecfe058df5c4e517d1e367ab8f46252b1ac1933ba378c
32625c0d69f5655c2903b3f39e75810796cd63675b223cf3162c57186d56e958
4ccf6e6cad8297c436ada38a905e3b6012912b31cdec7425cf9346e353fb17552
be3a8df2913dca16543a33ad5058f218c471dfbc12fb304158e92b6ed355cb07
9e23f1e6173c5dec4545840b8be58e5ad37cbea0a10dca59df2781589d273c4310
8477b52c0d32a1370c17f703941fb1a907a6794e7de2758709c9bbf80f21ee7c
722b9b490eb6a8c141746ae468e6e4e4ff0ae913797067aa0826f366c3193e
103b05653c73b52d7f825a18ddc8b06da700dbf953abb84554b7d4f7c28f3
salt: 49912979f1b5f282e5b8b28ab1328df74391dce7bda4f521ecb1100dcf0
42a2def852f9db59b64af5693cc282504240
token_request: 000280244840b27ca8c620f8b14caded9a198ba388ccdc8541e
962f68a0071535d958d18494af0cb1c1da4da8c8b33864f5a8f623b697cd56348
594e1a754794048a72c0ed17fb707560c9a7ebed3582f572f38cf60fcede11a
525c5e6d7b23435b6a20ad9f666f621f4f0323ca9a5a307d0b6664457c375426b
6bb183defaca914cf7c843168b85d2f498ee3d165685f49ad86e39fe6c4b7e6b
678f5250908ed25e5873c69b422368121a4210cadd6fc649097d3cb97a3e827
a0e742470f00c2f49dc6c08e4c9470dbf73df0ccbb96c10b02af0d7de7919c
e11ff8e1b4929e593cf3f319de9bda29a6d968b430835b4d422f344876ada0b8b
014f70b97e17

token_response: c2746ff644cffb28a2a9c19395fa19dbf61fd135a8384ffbf9feb66c253e64e69f53ae5eddc09f485b1e58f571134a34f2454993e737419
549c2c911cf94f2f68f6a3996d47f71e8d8d6fc5b1c074bf74fa59de4cbf32f5f08d45aa45492f0279c3b1a8d3b52698edbe1651be8e0e9eb22a27386c0fbeb2f6a
8260235eb36cf343d5a51808296216366284325d78f9c41ea3baf7eb671b70
82e09397837f74b4f0f6c838bce8aaf7242089dd5561f57735926cbad219fc9afe85ae49a8e951f63ca194b7ff618c06ee02267e7267bb996432dc76973819da80
e3e86947b0a4b436d3a972dafa3d0b0e1044b325f02c679996d9bc3e51590d5
4bc0b8c
token: 0002494dae1fc7e300c2d99990afcd5d5e1fc95305337dc12f78942c453400bfe8e6

// Test vector 5

skS: 2d2d2d2d24254794e2e295652495641545454520a45592d2d2d2d2d0a4d9
494576514942414414e42676b7168669473977390421451456414153434246b
3776767536a41674541416f49424151444c4775317261705831736334420a4f6b
7a387179573537536b6f6a41303543554b667174477e38366a424b5a4f764
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