Privacy Pass Issuance Protocol

Abstract

This document specifies two variants of the the two-message issuance protocol for Privacy Pass tokens: one that produces tokens that are privately verifiable, and another that produces tokens that are publicly verifiable. The privately verifiable issuance protocol optionally supports public metadata during the issuance flow.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 7 October 2022.

Copyright Notice

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

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

Table of Contents

1. Introduction
2. Terminology
3. Configuration
4. Token Challenge Requirements
5. Issuance Protocol for Privately Verifiable Tokens with Public Metadata
   5.1. Client-to-Issuer Request
   5.2. Issuer-to-Client Response
   5.3. Finalization
   5.4. Issuer Configuration
5. Issuance Protocol for Publicly Verifiable Tokens
   6.1. Client-to-Issuer Request
   6.2. Issuer-to-Client Response
   6.3. Finalization
   6.4. Issuer Configuration
6. Security considerations
7. IANA considerations
   8.1. Token Type
   8.2. Media Types
      8.2.1. "message/token-request" media type
      8.2.2. "message/token-response" media type
8. Normative 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, 4096
Authors' Addresses

1. Introduction

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

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

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

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following terms are used throughout this document.

*Client: An entity that provides authorization tokens to services across the Internet, in return for authorization.

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

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

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

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

3. Configuration

Issuers MUST provide one parameter for configuration:

1. Issuer Request URI: a token request URL for generating access tokens. For example, an Issuer URL might be https://issuer.example.net/example-token-request. This parameter uses resource media type "text/plain".

The Issuer parameters can be obtained from an Issuer via a directory object, which is a JSON object whose field names and values are raw values and URLs for the parameters.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>issuer-request-uri</td>
<td>Issuer Request URI resource URL as a JSON string</td>
</tr>
</tbody>
</table>

Table 1

As an example, the Issuer's JSON directory could look like:
Issuer directory resources have the media type "application/json" and are located at the well-known location /well-known/token-issuer-directory.

4. Token Challenge Requirements

Clients receive challenges for tokens, as described in [AUTHSCHEME]. The basic token issuance protocols described in this document can be interactive or non-interactive, and per-origin or cross-origin.

5. Issuance Protocol for Privately Verifiable Tokens with Public Metadata

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

Issuers provide a Private and Public Key, denoted skI and pkI, respectively, used to produce tokens as input to the protocol. See Section 5.4 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

*Issuer name, identifying the Issuer. This is typically a host name that can be used to construct HTTP requests to the Issuer.

*Issuer Public Key pkI, with a key identifier key_id computed as described in Section 5.4.

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

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

The Client first creates a context as follows:

```python
client_context = SetupVOPRFClient(0x0004, pkI)
```

Here, 0x0004 is the two-octet identifier corresponding to the OPRF(P-384, SHA-384) ciphersuite in [OPRF]. SetupVOPRFClient is defined in [OPRF], Section 3.2.

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

```python
nonce = random(32)
context = SHA256(challenge)
token_input = concat(0x0001, nonce, context, 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. Otherwise, the Client then creates a TokenRequest structured as follows:

```c
struct {
    uint16_t token_type = 0x0001;
    uint8_t 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.

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

"blinded_msg" is the Ne-octet blinded message defined above, computed as SerializeElement(blinded_element). Ne is as defined in [OPRF], Section 4.

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, with the TokenRequest as the body. The media type for this request is "message/token-request". An example request is shown below.
Upon receipt of the request, the Issuer validates the following conditions:

*The TokenRequest contains a supported token_type.

*The TokenRequest.token_key_id corresponds to a key ID of a Public Key owned by the issuer.

*The TokenRequest.blinded_request is of the correct size.

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the client.

5.2. Issuer-to-Client Response

Upon receipt of a TokenRequest, the Issuer 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 400 error to the client. Otherwise, if the Issuer is willing to produce a token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:

server_context = SetupVOPRFServer(0x0004, skI, pkI)
evaluate_element, proof = server_context.Evaluate(skI, blinded_element)

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

struct {
    uint8_t evaluate_msg[Nk];
    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])), where Ns is as defined in [OPRF], Section 4.

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

```plaintext
:status = 200
content-type = message/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 body values TokenResponse.evaluate_response and TokenResponse.evaluate_proof, yielding evaluated_element and proof. If deserialization of either value fails, the Client aborts the protocol. Otherwise, the Client processes the response as follows:

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

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

```c
struct {
    uint16_t token_type = 0x0001
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

Otherwise, the Client aborts the protocol.

5.4. Issuer Configuration

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

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

The key identifier for this specific key pair, denoted key_id, is computed as follows:
6. Issuance Protocol for Publicly Verifiable Tokens

This section describes a variant of the issuance protocol in Section 5 for producing publicly verifiable tokens. It differs from the previous variant in two important ways:

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

2. The issuance protocol does not admit public or private metadata to bind additional context to tokens.

Otherwise, this variant is nearly identical. In particular, Issuers provide a Private and Public Key, denoted $sk_I$ and $pk_I$, respectively, used to produce tokens as input to the protocol. See Section 6.4 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

*Issuer name, identifying the Issuer. This is typically a host name that can be used to construct HTTP requests to the Issuer.

*Issuer Public Key $pk_I$, with a key identifier $key_id$ computed as described in Section 6.4.

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

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

6.1. Client-to-Issuer Request

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

\[
\text{nonce} = \text{random}(32) \quad \text{context} = \text{SHA256}(\text{challenge}) \quad \text{token_input} = \text{concat}(0x0002, \text{nonce}, \text{context}, \text{key_id}) \\
\text{blinded_msg}, \text{blind_inv} = \text{rsabssa\_blind}(pk_I, \text{token_input})
\]

The $\text{rsabssa\_blind}$ function is defined in [BLINDRSA], Section 5.1.1.

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.

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

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

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

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

<Bytes containing the TokenRequest>
```

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

*The TokenRequest contains a supported token_type.

*The TokenRequest.token_key_id corresponds to a key ID of a Public Key owned by the issuer.

*The TokenRequest.blinded_msg is of the correct size.

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the Client, which will forward the error to the client.

6.2. Issuer-to-Client Response

If the Issuer is willing to produce a token token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:
This is encoded and transmitted to the client in the following TokenResponse structure:

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

The `rsabssa_blind_sign` function is defined in `BLINDRSA`, Section 5.1.2. The Issuer generates an HTTP response with status code 200 whose body consists of TokenResponse, with the content type set as "message/token-response".

```c
:status = 200
content-type = message/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 body as follows:

```c
authenticator = rsabssa_finalize(pkI, nonce, blind_sig, blind_inv)
```

The `rsabssa_finalize` function is defined in `BLINDRSA`, Section 5.1.3. If this succeeds, the Client then constructs a Token as described in [HTTP-Authentication] as follows:

```c
struct {
    uint16_t token_type = 0x0002
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

Otherwise, the Client aborts the protocol.

### 6.4. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted `skI` and `pkI`, respectively, used to produce tokens. Each key pair MUST be generated as as a valid 4096-bit RSA private key according to [TODO]. The key identifier for a keypair (`skI`, `pkI`), denoted `key_id`, is computed as SHA256(`encoded_key`), where `encoded_key` is a DER-encoded SubjectPublicKeyInfo object carrying `pkI`.
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 document also apply in the Privacy Pass use-case. Considerations related to broader privacy and security concerns in a multi-Client and multi-Issuer setting are deferred to the Architecture document [I-D.ietf-privacypass-architecture].

8. IANA considerations

8.1. Token Type

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Publicly Verifiable</th>
<th>Public Metadata</th>
<th>Private Metadata</th>
<th>Nk</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>VOPRF(P-384, SHA-384)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>48</td>
<td>Section 5</td>
</tr>
<tr>
<td>0x0002</td>
<td>Blind RSA, 4096</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>512</td>
<td>Section 6</td>
</tr>
</tbody>
</table>

Table 2: Token Types

8.2. Media Types

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

*TokenRequest: "message/token-request"

*TokenResponse: "message/token-response"

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

8.2.1. "message/token-request" media type

Type name: message

Subtype name: token-request

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 7
Interoperability considerations:  
N/A

Published specification:  this specification

Applications that use this media type:  N/A

Fragment identifier considerations:  N/A

Additional information:

Magic number(s):  N/A

Deprecated alias names for this type:  N/A

File extension(s):  N/A

Macintosh file type code(s):  N/A

Person and email address to contact for further information:  see Authors' Addresses section

Intended usage:  COMMON

Restrictions on usage:  N/A

Author:  see Authors' Addresses section

Change controller:  IESG

8.2.2. "message/token-response" media type

Type name:  message

Subtype name:  access-token-response

Required parameters:  N/A

Optional parameters:  None

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

Security considerations:  see Section 7

Interoperability considerations:  N/A

Published specification:  this specification

Applications that use this media type:  N/A

Fragment identifier considerations:  N/A
Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

9. Normative References


Appendix A. Acknowledgements

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

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 encoded OPRF private key, serialized using SerializeScalar from Section 2.1 of [OPRF] and represented as a hexadecimal string.

*pks: The encoded OPRF public key, serialized using SerializeElement from Section 2.1 of [OPRF] and represented as a hexadecimal string.

*challenge: A random challenge, 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.
**Section 5.2**, represented as a hexadecimal string.

*token: The output Token from the protocol, represented as a hexadecimal string.

skS: 0177781aeced893dccdf80713d318a801e2a0498240f4c6f650304bbfd0f8d3b5c0cf6cefee457aaa983ec02ff283b7a9
pkS: 022c63f79ac59c0ba3d204245f676a2133bd6120c90d67afa05cd6f8614294b7366c2526458300551b79a4911c2590a36
challenge: a5d46383359ef34e3c4a7b8d1b3165778fffc9b70c9e6a60dd14143e49c9f9d5
nonce: 5d4799f8338ddc50a6685f83b8ecd264b2f157015229d12b3384c0f199efe7b8
blind: 0322fec505230992256296063d89b59cc03e83184eb6187076d264137622dd0248e4e525bdc007b80d1560e0a6f49d9
token_request: 00011a02861fd50d14be873611ccf0131d2cc872c79d0260c6763498a2a3f14ca926009c0f2476534061ed52b6861b7ed2bac9ea
token_response: 038e3625b6a769668a99680e46cf9479f5dc1e86d57164ab3b4a569ddfc486bf1485d4916a5194f0c0518d3e8449468421ba3e8144a790278ff0f3c405863d69451a2a782005c45760c2f1a604134d877b39e8bcbbf920e5de4a3372557debf211765c9d6997686b0c039f9082d6a3e038e891246240173d2cf3d69a4613b0f841597902922e7c7af1e2e4639e4
token: 00015d4799f8338ddc50a6685f83b8ecd264b2f157015229d12b3384c0f199efe7b8742cdff0ed756ea6808868ef109a280a393e001d2fa56b1be46ecb31fa25e76731a5b1d698ea7ab843b8e8a71ed9b2fffa70457a43a8f687939424bb29a7554b40fde130ab7a82271590cb73f99a45b640ca1c85180ba9ca1a40ab8a664406a34b3b63b5e2e5c455cea00010a96f7

**B.2. Issuance Protocol 2 - Blind RSA, 4096**

The test vector below lists the following values:

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

*pks: The DER-encoded SubjectPublicKeyInfo object carrying the public key corresponding to skS, as described in Section 6.4, represented as a hexadecimal string.

*challenge: A random challenge, represented as a hexadecimal string.

*nonce: The 32-byte client nonce generated according to Section 6.4, 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.
challenge: 83ce743dcdadd5fc4aeb0357977bab4266353c390a15b88947f0b1c62e4a87c22
nonce: 7e09da9dfe4365a5f40e69262f788b18ed2f92daf885358d9831874e3dd9d22
blind: cd60e3332386d0166eb768e875221501e5bcdf49aaac83191ea948a7719e914
0cb6701f7301b7d445ed7ea7bc55e261e55d9ca455c4b8f79ed150b2e3e407bb7624b
6f90b338453b255174ee0c570a8171c203dce8563afe9f48e2b49c773ba1031987fb48
9d81d31876f53e264e609a3ea628cf2420005ed3071aeb66577472c7e7df947915c8cd
333e3f5708e45e6e5e5ede9f21d2f940c8d062868ed7c55bb99433c6767608e
8a4eacbdb16aae4c5b8cab26b478faa636ae3c3d1ffbf4200969eeef52aab4892c9e60a
df18a37b4e282565baa8ecbf55fc97ac62af27349ba98ca7462cbac41d700b217814a66e
1b257289c3b345be652b82d139b08a050809e0d88772140bf43f11497114b0200e7e
5ffe1af5cf874cc29a0c8df65d52814bced55ae63701f737d140ca82c6526272129ea389
f3cb6058bf4ff3b783f3483a2495b2cbf9d51990ff02b5cbe643fe4a4306ee7c4
3e9d9b02f1b0559aa238b8882a6e879b1ff366fe4fe433fd4e255d209c9f3e7c7def0416
11663e5f793c7031316995b3f3e963280b8199a9c83096f69359f9d84a6a7a8fbeb42
4d9726c25b1618e1b9a55fa5e49be600de6110597976733de8f9adb25af7e8269232bed
7c774b4dc50bf9cd72a43635a7667b5aceef2b187e56b4f1b56aefef0fb8bf1636815
be90b3727797a6a4c74f41
salt: d13a47fa6466a37283e51ac3f024022ff7c98a18e78080b7ac3014
0618978305227158709165c6da0
token_request: 0002013a370077e2859908e741dcac88184383b7c95cd82966419064
90205bf28e6745396dd9ec1761c4676e9ef8ce3272588194c48bc60f7cc3ff19219a6b6
96523044d0d79e4d88f2d9b9c365975363a1a265d244d69a17b43b3e658f5b8f88d06
52f2f2ef417ef97f7d7db2626d4997d3856d8f9277385df776fa4ff47a7b171
0d87f40e52b25bb852a7f5d6b203a4b3b08049e9b7272776962473
64a8c354377808c3a8a87fb870a299ac08107a5dd3f467733d76c2359e3463ee3bcbff68
1c3c7e10ff0f1a2bb9bzb26f6eafe14f81f036a71ac747725b5c9f81e032b8b5afbf7d756ebc1
ccfd8e8b0cf72735ea297e8a9b0a4d22ba81a2c2f2a72983107555035863c3
397d487f4027d2d8298116d2c93298810cb6b82ad033f5b16f9f7a65d8f74b7bdcaee000db
911b40f46c43b769c8ab55852f998b478229b6383ab40e833fae4ec4afca09a00dfb2
39209c7c4ed982c64dbca1e16fd7397f6f77e7a73b1a8e80199e95b7ce790299f3c89c4f
2d3d85cd1825e2f7073aad63fa4cc5309740024c4e3a748e0db309ff5ce7ac58b1d9eae2f
6d02b05dee582aaeaf4a4f053f131b1ae15be9420e3eccdb6851690d5ae5e2bca7713e6
666b55b097e5e5e5719db0324ecaca613a7f6bf367775a9d0ba5ef7dff21a8da73b80b1
31e653117f7e90ad8c7b213225r5253aceaa
token_response: 061780e99bc8b51fe8e17022ee2d55b043198bcb1aa33f761d213a9
8d831abae5f514d30904a7c97d7e5c513278cd9655c4b7d72f3a46e626565b73e5f1b927
b541f28556543d2473c363e1904a116cb1a30d8f99b32d3c8f74ae074b65afdaadb21977
6df104ea948c5df92f4615ecd4a97b86f1e463b1c69171cf734f94c240760dc2555d7cc8
Authors' Addresses

Sofía Celi  
Cloudflare  
Lisbon  
Portugal  
Email: sceli@cloudflare.com

Alex Davidson  
Brave Software  
Lisbon  
Portugal  
Email: alex.davidson92@gmail.com

Armando Faz-Hernandez  
Cloudflare  
101 Townsend St  
San Francisco,  
United States of America  
Email: armfazh@cloudflare.com

Steven Valdez  
Google LLC  
Email: svaldez@chromium.org

Christopher A. Wood  
Cloudflare  
101 Townsend St  
San Francisco,  
United States of America  
Email: caw@heapingbits.net