Network Working Group
Internet-Draft
Intended status: Informational
Expires: 9 July 2021

S. Celi
Cloudflare
A. Davidson
LIP
A. Faz-Hernandez
Cloudflare

5 January 2021

Privacy Pass Protocol Specification
draft-ietf-privacypass-protocol-00

Abstract

This document specifies the Privacy Pass protocol. This protocol provides anonymity-preserving authorization of clients to servers. In particular, client re-authorization events cannot be linked to any previous initial authorization. Privacy Pass is intended to be used as a performant protocol in the application-layer.

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 9 July 2021.

Table of Contents

1. Introduction

A common problem on the Internet is providing an effective mechanism for servers to derive trust from clients that they interact with. Typically, this can be done by providing some sort of authorization challenge to the client. But this also negatively impacts the experience of clients that regularly have to solve such challenges.

To mitigate accessibility issues, a client that correctly solves the challenge can be provided with a cookie. This cookie can be presented the next time the client interacts with the server, instead of performing the challenge. However, this does not solve the problem of reauthorization of clients across multiple domains. Using current tools, providing some multi-domain authorization token would allow linking client browsing patterns across those domains, and severely reduces their online privacy.

The Privacy Pass protocol provides a set of cross-domain authorization tokens that protect the client's anonymity in message exchanges with a server. This allows clients to communicate an attestation of a previously authenticated server action, without having to reauthenticate manually. The tokens retain anonymity in the sense that the act of revealing them cannot be linked back to the session where they were initially issued.

This document lays out the generic description of the protocol, along with the data and message formats. We detail an implementation of the protocol functionality based on the description of a verifiable oblivious pseudorandom function [I-D.irtf-cfrg-voprf].

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 a separate document [draft-davidson-pp-architecture]. In addition, [draft-svaldez-pp-http-api] provides an instantiation of this protocol intended for the HTTP setting.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

The following terms are used throughout this document.

* Server: A service that provides the server-side functionality required by the protocol. May be referred to as the issuer.

* Client: An entity that seeks authorization from a server that supports interactions in the Privacy Pass protocol.

* Key: The secret key used by the server for authorizing client data.

We assume that all protocol messages are encoded into raw byte format before being sent. We use the TLS presentation language [RFC8446] to describe the structure of protocol data types and messages.

3. Background

We discuss the core motivation behind the protocol along with the guarantees and assumptions that we make in this document.

3.1. Motivating use-cases

The Privacy Pass protocol was originally developed to provide anonymous authorization of Tor users. In particular, the protocol allows clients to reveal authorization tokens that they have been issued without linking the authorization to the actual issuance event. This means that the tokens cannot be used to link the browsing patterns of clients that reveal tokens.

Beyond these uses-cases, the Privacy Pass protocol is used in a number of practical applications. See [DGSTV18], [TrustTokenAPI], [PrivateStorage], [OpenPrivacy], and [Brave] for examples.

3.2. Anonymity and security guarantees

Privacy Pass provides anonymity-preserving authorization tokens for clients. Throughout this document, we use the terms "anonymous", "anonymous-preserving" and "anonymity" to refer to the core security guarantee of the protocol. Informally, this guarantee means that any token issued by a server key and subsequently redeemed is indistinguishable from any other token issued under the same key.

Privacy Pass also prohibits clients from forging tokens, as otherwise the protocol would have little value as an authorization protocol. Informally, this means any client that is issued "N" tokens under a given server key cannot redeem more than "N" valid tokens.

Section 6 elaborates on these protocol anonymity and security requirements.

3.3. Basic assumptions

We make only a few minimal assumptions about the environment of the clients and servers supporting the Privacy Pass protocol.

* At any one time, we assume that the server uses only one configuration containing their ciphersuite choice along with their secret key data. This ensures that all clients are issued tokens under the single key associated with any given epoch.

* We assume that the client has access to a global directory of the current public parts of the configurations used the server.

The wider ecosystem that this protocol is employed in is described in [draft-davidson-pp-architecture].

4. Protocol description

The Privacy Pass protocol is split into two phases that are built upon the functionality described in Section 5 later.

The first phase, "issuance", provides the client with unlinkable tokens that can be used to initiate re-authorization with the server in the future. The second phase, "redemption", allows the client to redeem a given re-authorization token with the server that it interacted with during the issuance phase. The protocol must satisfy two cryptographic security requirements known as "unlinkability" and "unforgeability". These requirements are covered in Section 6.

4.1. Server setup

Before the protocol takes place, the server chooses a ciphersuite and generates a keypair by running "(pkS, skS) = KeyGen()". This configuration must be available to all clients that interact with the server (for the purpose of engaging in a Privacy Pass exchange). We assume that the server has a public (and unique) identity that the client uses to retrieve this configuration.

4.2. Client setup

The client initialises a global storage system "store" that allows it store the tokens that are received during issuance. The storage system is a map of server identifiers ("server.id") to arrays of stored tokens. We assume that the client knows the server public key "pkS" ahead of time. The client picks a value "m" of tokens to receive during the issuance phase. In [draft-davidson-pp-architecture] we discuss mechanisms that the client can use to ensure that this public key is consistent across the entire ecosystem.

4.3. Issuance phase

The issuance phase allows the client to receive "m" anonymous

authorization tokens from the server.

```
  Client(pkS, m)                                Server(skS, pkS)
  ------------------------------------------------------------
  cInput = Generate(m)
  req = cInput.req

                             req
                      ------------------->

                            serverResp = Issue(pkS, skS, req)

                          issueResp
                      <-------------------

  tokens = Process(pkS, cInput, issueResp)
  store[server.id].push(tokens)
```

4.4. Redemption phase

The redemption phase allows the client to anonymously reauthenticate to the server, using data that it has received from a previous issuance phase.

```
  Client(info)                                  Server(skS, pkS)
  ------------------------------------------------------------
  token = store[Issue.id].pop()
  req = Redeem(token, info)

                             req
                       ------------------>

                             if (dsIdx.includes(req.data)) {
                               raise ERR_DOUBLE_SPEND
                             }
                             resp = Verify(pkS, skS, req)
                             if (resp.success) {
                               dsIdx.push(req.data)
                             }

                              resp
                       <------------------
  Output resp
```

4.4.1. Client info

The client input "info" is arbitrary byte data that is used for linking the redemption request to the specific session. We RECOMMEND that "info" is constructed as the following concatenated byte-encoded data:

```
len(aux) || aux || len(server.id) || server.id || current_time()
```

where "len(x)" is the length of "x" in bytes, and "aux" is arbitrary auxiliary data chosen by the client. The usage of "current_time()" allows the server to check that the redemption request has happened in an appropriate time window.

4.4.2. Double-spend protection

To protect against clients that attempt to spend a value "req.data" more than once, the server uses an index, "dsIdx", to collect valid inputs it witnesses. Since this store needs to only be optimized for storage and querying, a structure such as a Bloom filter suffices. The storage should be parameterized to live as long as the server keypair that is in use. See Section 6 for more details.

4.5. Handling errors

It is possible for the API functions from Section 5.2 to return one of the errors indicated in Section 5.3 rather than their expected value. In these cases, we assume that the entire protocol aborts.

5. Functionality

This section details the data types and API functions that are used to construct the protocol in Section 4.

We provide an explicit instantiation of the Privacy Pass API in Section 7.3, based on the public API provided in [I-D.irtf-cfrg-voprf].

5.1. Data structures

The following data structures are used throughout the Privacy Pass protocol and are written in the TLS presentation language [RFC8446]. It is intended that any of these data structures can be written into widely-adopted encoding schemes such as those detailed in TLS [RFC8446], CBOR [RFC7049], and JSON [RFC7159].

5.1.1. Ciphersuite

The "Ciphersuite" enum provides identifiers for each of the supported ciphersuites of the protocol. Some initial values that are supported by the core protocol are described in Section 8. Note that the list of supported ciphersuites may be expanded by extensions to the core protocol description in separate documents.

5.1.2. Keys

We use the following types to describe the public and private keys used by the server.

```
opaque PublicKey<1..2^16-1>
```

```
opaque PrivateKey<1..2^16-1>
```

5.1.3. IssuanceInput

The "IssuanceInput" struct describes the data that is initially generated by the client during the issuance phase.

Firstly, we define sequences of bytes that partition the client input.

```
opaque Internal<1..2^16-1>
opaque IssuanceRequest<1..2^16-1>
```

These data types represent members of the wider "IssuanceInput" data type.

```
struct {
  Internal data[m]
  IssuanceRequest req[m]
} IssuanceInput;
```

Note that a "IssuanceInput" contains equal-length arrays of "Internal" and "IssuanceRequest" types corresponding to the number of tokens that should be issued.

5.1.4. IssuanceResponse

Firstly, the "IssuedToken" type corresponds to a single sequence of bytes that represents a single issued token received from the server.

```
opaque IssuedToken<1..2^16-1>
```

Then an "IssuanceResponse" corresponds to a collection of "IssuedTokens" as well as a sequence of bytes "proof".

```
struct {
  IssuedToken tokens[m]
  opaque proof<1..2^16-1>
}
```

The value of "m" is equal to the length of the "IssuanceRequest" vector sent by the client.

5.1.5. RedemptionToken

The "RedemptionToken" struct contains the data required to generate the client message in the redemption phase of the Privacy Pass protocol.

```
struct {
  opaque data<1..2^16-1>;
  opaque issued<1..2^16-1>;
```

```
} RedemptionToken;
```

5.1.6. RedemptionRequest

The "RedemptionRequest" struct consists of the data that is sent by the client during the redemption phase of the protocol.

```
struct {
  opaque data<1..2^16-1>;
  opaque tag<1..2^16-1>;
  opaque info<1..2^16-1>;
} RedemptionRequest;
```

5.1.7. RedemptionResponse

The "RedemptionResponse" struct corresponds to a boolean value that indicates whether the "RedemptionRequest" sent by the client is valid. It can also contain any associated data.

```
struct {
  boolean success;
  opaque ad<1..2^16-1>;
} RedemptionResponse;
```

5.2. API functions

The following functions wrap the core of the functionality required in the Privacy Pass protocol. For each of the descriptions, we essentially provide the function signature, leaving the actual contents to be defined by specific instantiations or extensions of the protocol.

5.2.1. Generate

A function run by the client to generate the initial data that is used as its input in the Privacy Pass protocol.

Inputs:

* "m": A "uint8" value corresponding to the number of Privacy Pass tokens to generate.

Outputs:

* "input": An "IssuanceInput" struct.

5.2.2. Issue

A function run by the server to issue valid redemption tokens to the client.

Inputs:

* "pkS": A server "PublicKey".

* "skS": A server "PrivateKey".

* "req": An "IssuanceRequest" struct.

Outputs:

* "resp": An "IssuanceResponse" struct.

5.2.3. Process

Run by the client when processing the server response in the issuance phase of the protocol.

Inputs:

* "pkS": An server "PublicKey".

* "input": An "IssuanceInput" struct.

* "resp": An "IssuanceResponse" struct.

Outputs:

* "tokens": A vector of "RedemptionToken" structs, whose length is equal to length of the internal "ServerEvaluation" vector in the "IssuanceResponse" struct.

Throws:

* "ERR_PROOF_VALIDATION" (Section 5.3)

5.2.4. Redeem

Run by the client in the redemption phase of the protocol to generate the client's message.

Inputs:

* "token": A "RedemptionToken" struct.

* "info": An "opaque<1..2^16-1>" type corresponding to data that is linked to the redemption. See Section 4.4.1 for advice on how to construct this.

Outputs:

* "req": A "RedemptionRequest" struct.

5.2.5. Verify

Run by the server in the redemption phase of the protocol.
Determines whether the data sent by the client is valid.

Inputs:

* "pkS": An server "PublicKey".

* "skS": An server "PrivateKey".

* "req": A "RedemptionRequest" struct.

Outputs:

* "resp": A "RedemptionResponse" struct.

5.3. Error types

* "ERR_PROOF_VALIDATION": Error occurred when a client attempted to verify the proof that is part of the server's response.

* "ERR_DOUBLE_SPEND": Error occurred when a client has attempted to redeem a token that has already been used for authorization.

6. Security considerations

We discuss the security requirements that are necessary to uphold when instantiating the Privacy Pass protocol. In particular, we focus on the security requirements of "unlinkability", and "unforgeability". Informally, the notion of unlinkability is required to preserve the anonymity of the client in the redemption phase of the protocol. The notion of unforgeability is to protect against an adversarial client that may look to subvert the security of the protocol.

Both requirements are modelled as typical cryptographic security games, following the formats laid out in [DGSTV18] and [KLOR20].

Note that the privacy requirements of the protocol are covered in the architectural framework document [draft-davidson-pp-architecture].

6.1. Unlinkability

Formally speaking the security model is the following:

* The adversary runs the server setup and generates a keypair "(pkS, skS)".

* The adversary specifies a number "Q" of issuance phases to initiate, where each phase "i in range(Q)" consists of "m_i" Issue evaluations.

* The adversary runs "Issue" using the keypair that it generated on

each of the client messages in the issuance phase.

* When the adversary wants, it stops the issuance phase, and a random number "l" is picked from "range(Q)".

* A redemption phase is initiated with a single token with index "i" randomly sampled from "range(m_l)".

* The adversary guesses an index "l_guess" corresponding to the index of the issuance phase that it believes the redemption token was received in.

* The adversary succeeds if "l == l_guess".

The security requirement is that the adversary has only a negligible probability of success greater than "1/Q".

6.2. One-more unforgeability

The one-more unforgeability requirement states that it is hard for any adversarial client that has received "m" valid tokens from the issuance phase to redeem "m+1" of them. In essence, this requirement prevents a malicious client from being able to forge valid tokens based on the Issue responses that it sees.

The security model roughly takes the following form:

* The adversary specifies a number "Q" of issuance phases to initiate with the server, where each phase "i in range(Q)" consists of "m_i" server evaluation. Let "m = sum(m_i)" where "i in range(Q)".

* The adversary receives "Q" responses, where the response with index "i" contains "m_i" individual tokens.

* The adversary initiates "m_adv" redemption sessions with the server and the server verifies that the sessions are successful (return true), and that each request includes a unique token. The adversary succeeds in "m_succ =< m_adv" redemption sessions.

* The adversary succeeds if "m_succ > m".

The security requirement is that the adversarial client has only a negligible probability of succeeding.

Note that [KLOR20] strengthens the capabilities of the adversary, in comparison to the original work of [DGSTV18]. In [KLOR20], the adversary is provided with oracle access that allows it to verify that the server responses in the issuance phase are valid.

6.3. Double-spend protection

All issuing servers should implement a robust, global storage-query mechanism for checking that tokens sent by clients have not been spent before. Such tokens only need to be checked for each server individually. This prevents clients from "replaying" previous requests, and is necessary for achieving the unforgeability requirement.

6.4. Additional token metadata

Some use-cases of the Privacy Pass protocol benefit from associating a limited amount of metadata with tokens that can be read by the server when a token is redeemed. Adding metadata to tokens can be used as a vector to segment the anonymity of the client in the protocol. Therefore, it is important that any metadata that is added is heavily limited.

Any additional metadata that can be added to redemption tokens should be described in the specific protocol instantiation. Note that any additional metadata will have to be justified in light of the privacy concerns raised above. For more details on the impacts associated with segmenting user privacy, see [draft-davidson-pp-architecture].

Any metadata added to tokens will be considered either "public" or "private". Public metadata corresponds to unmodifiable bits that a client can read. Private metadata corresponds to unmodifiable private bits that should be obscured to the client.

Note that the instantiation in Section 7 provides randomized redemption tokens with no additional metadata for an server with a single key.

6.5. Maximum number of tokens issued

Servers SHOULD impose a hard ceiling on the number of tokens that can be issued in a single issuance phase to a client. If there is no limit, malicious clients could abuse this and cause excessive computation, leading to a Denial-of-Service attack.

7. VOPRF instantiation

In this section, we show how to instantiate the functional API in Section 5 with the VOPRF protocol described in [I-D.irtf-cfrg-voprf]. Moreover, we show that this protocol satisfies the security requirements laid out in Section 6, based on the security proofs provided in [DGSTV18] and [KLOR20].

7.1. Recommended ciphersuites

The RECOMMENDED server ciphersuites are as follows: detailed in [I-D.irtf-cfrg-voprf]:

* OPRF(curve448, SHA-512) (ID = 0x0002);

* OPRF(P-384, SHA-512) (ID = 0x0004);

* OPRF(P-521, SHA-512) (ID = 0x0005).

We deliberately avoid the usage of smaller ciphersuites (associated with P-256 and curve25519) due to the potential to reduce security to unfavourable levels via static Diffie Hellman attacks. See [I-D.irtf-cfrg-voprf] for more details.

7.2. Protocol contexts

Note that we must run the verifiable version of the protocol in [I-D.irtf-cfrg-voprf]. Therefore the "server" takes the role of the "Server" running in "modeVerifiable". In other words, the "server" runs "(ctxtI, pkS) = SetupVerifiableServer(suite)"; where "suite" is one of the ciphersuites in Section 7.1, "ctxt" contains the internal VOPRF server functionality and secret key "skS", and "pkS" is the server public key. Likewise, run "ctxtC = SetupVerifiableClient(suite)" to generate the Client context.

7.3. Functionality

We instantiate each functions using the API functions in [I-D.irtf-cfrg-voprf]. Note that we use the framework mentioned in the document to allow for batching multiple tokens into a single VOPRF evaluation. For the explicit signatures of each of the functions, refer to Section 5.

7.3.1. Generate

```
def Generate(m):
  tokens = []
  blindedTokens = []
  for i in range(m):
    x = random_bytes()
    (token, blindedToken) = Blind(x)
    token[i] = token
    blindedToken[i] = blindedToken
  return IssuanceInput {
           internal: tokens,
           req: blindedTokens,
         }
```

7.3.2. Issue

For this functionality, note that we supply multiple tokens in "req" to "Evaluate". This allows batching a single proof object for multiple evaluations. While the construction in [I-D.irtf-cfrg-voprf] only permits a single input, we follow the advice for providing vectors of inputs.

```
def Issue(pkS, skS, req):
  Ev = Evaluate(skS, pkS, req)
  return IssuanceResponse {
           tokens: Ev.elements,
           proof: Ev.proof,
         }
```

7.3.3. Process

Similarly to "Issue", we follow the advice for providing vectors of inputs to the "Unblind" function for verifying the batched proof object.

```
Process(pkS, input, resp):
  unblindedTokens = Unblind(pkS, input.data, input.req, resp)
  redemptionTokens = []
  for bt in unblindedTokens:
    rt = RedemptionToken { data: input.data, issued: bt }
    redemptionTokens[i] = rt
  return redemptionTokens
```

7.3.4. Redeem

```
def Redeem(token, info):
  tag = Finalize(token.data, token.issued, info)
  return RedemptionRequest {
           data: data,
           tag: tag,
           info: info,
         }
```

7.3.5. Verify

```
def Verify(pkS, skS, req):
  resp = VerifyFinalize(skS, pkS, req.data, req.info, req.tag)
  Output RedemptionResponse {
           success: resp
         }
```

7.4. Security justification

The protocol devised in Section 4, coupled with the API instantiation in Section 7.3, are equivalent to the protocol description in [DGSTV18] and [KLOR20] from a security perspective. In [DGSTV18], it is proven that this protocol satisfies the security requirements of unlinkability (Section 6.1) and unforgeability (Section 6.2).

The unlinkability property follows unconditionally as the view of the adversary in the redemption phase is distributed independently of the issuance phase. The unforgeability property follows from the one-more decryption security of the ElGamal cryptosystem [DGSTV18]. In [KLOR20] it is also proven that this protocol satisfies the stronger

notion of unforgeability, where the adversary is granted a verification oracle, under the chosen-target Diffie-Hellman assumption.

Note that the existing security proofs do not leverage the VOPRF primitive as a black-box in the security reductions. Instead, it relies on the underlying operations in a non-black-box manner. Hence, an explicit reduction from the generic VOPRF primitive to the Privacy Pass protocol would strengthen these security guarantees.

8. Protocol ciphersuites

The ciphersuites that we describe for the Privacy Pass protocol are derived from the core instantiations of the protocol (such as in Section 7).

In each of the ciphersuites below, the maximum security provided corresponds to the maximum difficulty of computing a discrete logarithm in the group. Note that the actual security level MAY be lower. See the security considerations in [I-D.irtf-cfrg-voprf] for examples.

8.1. PP(OPRF2)

* OPRF2 = OPRF(curve448, SHA-512)
* ID = 0x0001
* Maximum security provided: 224 bits

8.2. PP(OPRF4)

* OPRF4 = OPRF(P-384, SHA-512)
* ID = 0x0002
* Maximum security provided: 192 bits

8.3. PP(OPRF5)

* OPRF5 = OPRF(P-521, SHA-512)
* ID = 0x0003
* Maximum security provided: 256 bits

9. Extensions framework policy

The intention with providing the Privacy Pass API in Section 5 is to allow new instantiations of the Privacy Pass protocol. These instantiations may provide either modified VOPRF constructions, or simply implement the API in a completely different way.

Extensions to this initial draft SHOULD be specified as separate documents taking one of two possible routes:

* Produce new VOPRF-like primitives that use the same public API provided in [I-D.irtf-cfrg-voprf] to implement the Privacy Pass API, but with different internal operations.

* Implement the Privacy Pass API in a different way to the proposed implementation in Section 7.

If an extension requires changing the generic protocol description as described in Section 4, then the change may have to result in changes to the draft specification here also.

Each new extension that modifies the internals of the protocol in either of the two ways MUST re-justify that the extended protocol still satisfies the security requirements in Section 6. Protocol extensions MAY put forward new security guarantees if they are applicable.

The extensions MUST also conform with the extension framework policy as set out in the architectural framework document. For example, this may concern any potential impact on client anonymity that the extension may introduce.

10. References

10.1. Normative References

[draft-davidson-pp-architecture]
Davidson, A., "Privacy Pass: Architectural Framework", n.d., <https://github.com/alxdavids/privacy-pass-ietf/tree/master/drafts/draft-davidson-pp-architecture>.

[draft-svaldez-pp-http-api]
Valdez, S., "Privacy Pass: HTTP API", n.d., <https://github.com/alxdavids/privacy-pass-ietf/tree/master/drafts/draft-davidson-pp-architecture>.

[I-D.irtf-cfrg-voprf]
Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-05, 2 November 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-voprf-05.txt>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

10.2. Informative References

[Brave] "Brave Rewards", n.d., <https://brave.com/brave-rewards/>.

[DGSTV18] "Privacy Pass, Bypassing Internet Challenges Anonymously", n.d., <https://petsymposium.org/2018/files/papers/issue3/popets-2018-0026.pdf>.

[KLOR20] "Anonymous Tokens with Private Metadata Bit", n.d., <https://eprint.iacr.org/2020/072>.

[OpenPrivacy] "Token Based Services - Differences from PrivacyPass", n.d., <https://openprivacy.ca/assets/towards-anonymous-prepaid-services.pdf>.

[PrivateStorage] Steininger, L., "The Path from S4 to PrivateStorage", n.d., <https://medium.com/least-authority/the-path-from-s4-to-privatestorage-ae9d4a10b2ae>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/info/rfc7159>.

[TrustTokenAPI] WICG, ., "Trust Token API", n.d., <https://github.com/WICG/trust-token-api>.

Appendix A. Document contributors

* Alex Davidson (alex.davidson92@gmail.com)

* Sofia Celi (cherenkov@riseup.net)

* Christopher Wood (caw@heapingbits.net)

Authors' Addresses

Sofía Celi
Cloudflare
Lisbon
Portugal

Email: sceli@cloudflare.com

Alex Davidson
LIP
Lisbon
Portugal

Email: alex.davidson92@gmail.com

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

Email: armfazh@cloudflare.com