

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 14 January 2021

A. Davidson
Cloudflare
13 July 2020

Privacy Pass: The Protocol
draft-davidson-pp-protocol-01

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 14 January 2021.

Copyright Notice

Copyright (c) 2020 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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Terminology	4
3. Background	4
3.1. Motivating use-cases	4
3.2. Anonymity and security guarantees	5
3.3. Basic assumptions	5
4. Protocol description	5
4.1. Server setup	6
4.2. Client setup	6
4.3. Issuance phase	6
4.4. Redemption phase	6
4.4.1. Client info	7
4.4.2. Double-spend protection	7
4.5. Handling errors	7
5. Functionality	8
5.1. Data structures	8
5.1.1. Ciphersuite	8
5.1.2. Keys	8
5.1.3. IssuanceInput	8
5.1.4. IssuanceResponse	9
5.1.5. RedemptionToken	9
5.1.6. RedemptionRequest	9
5.1.7. RedemptionResponse	10
5.2. API functions	10
5.2.1. Generate	10
5.2.2. Issue	10
5.2.3. Process	11
5.2.4. Redeem	11
5.2.5. Verify	11
5.3. Error types	12
6. Security considerations	12
6.1. Unlinkability	12
6.2. One-more unforgeability	13
6.3. Double-spend protection	14
6.4. Additional token metadata	14
6.5. Maximum number of tokens issued	14
7. VOPRF instantiation	14
7.1. Recommended ciphersuites	15
7.2. Protocol contexts	15
7.3. Functionality	15
7.3.1. Generate	15
7.3.2. Issue	16
7.3.3. Process	16
7.3.4. Redeem	16
7.3.5. Verify	17
7.4. Security justification	17

8. Protocol ciphersuites	17
8.1. PP(OPRF2)	18
8.2. PP(OPRF4)	18
8.3. PP(OPRF5)	18
9. Extensions framework policy	18
10. References	19
10.1. Normative References	19
10.2. Informative References	19
Appendix A. Document contributors	20
Author's Address	20

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 " $(pkS, skS) = \text{KeyGen}()$ ". 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	
<div style="display: flex; align-items: center; justify-content: center;"> <div style="text-align: right; margin-right: 10px;">req</div> <div style="flex-grow: 1; border-bottom: 1px dashed black; position: relative;"> <div style="position: absolute; right: -5px; top: -5px;">></div> </div> </div>	
<div style="display: flex; align-items: center; justify-content: center;"> <div style="flex-grow: 1; border-bottom: 1px dashed black; position: relative;"> <div style="position: absolute; left: -5px; top: -5px;"><</div> </div> <div style="text-align: left; margin-left: 10px;">issueResp</div> </div>	
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 [{{sec-reqs}}](#) 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..2¹⁶-1>

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

```
struct {  
    IssuedToken tokens[m]  
    opaque proof<1..216-1>  
}
```

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..216-1>;  
    opaque issued<1..216-1>;  
} 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..216-1>;  
    opaque tag<1..216-1>;  
    opaque info<1..216-1>;  
} 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..2¹⁶-1>" 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 a 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)

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

8.2. PP(OPRF4)

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

8.3. PP(OPRF5)

- * $\text{OPRF5} = \text{OPRF}(\text{P-521}, \text{SHA-512})$
- * $\text{ID} = 0x0003$
- * 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., Sullivan, N., and C. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-03, 9 March 2020, <<http://www.ietf.org/internet-drafts/draft-irtf-cfrg-voprf-03.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)

Author's Address

Alex Davidson
Cloudflare
Lisbon
Portugal

Email: alex.davidson92@gmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 13 January 2022

S. Valdez
Google LLC
12 July 2021

Privacy Pass HTTP API
draft-ietf-privacypass-http-api-01

Abstract

This document specifies an integration for Privacy Pass over an HTTP API, along with recommendations on how key commitments are stored and accessed by HTTP-based consumers.

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 13 January 2022.

Copyright Notice

Copyright (c) 2021 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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Terminology	2
1.2. Layout	2
1.3. Requirements	3
2. Privacy Pass HTTP API Wrapping	3
3. Server key registry	3
3.1. Key Registry	4
3.2. Server Configuration Retrieval	5
4. Key Commitment Retrieval	5
5. Privacy Pass Issuance	7
6. Privacy Pass Redemption	8
6.1. Generic Token Redemption	8
6.2. Direct Redemption	9
6.3. Delegated Redemption	10
7. Security Considerations	11
8. Privacy considerations	11
9. IANA Considerations	11
9.1. Well-Known URI	11
10. Normative References	11
Author's Address	12

1. Introduction

The Privacy Pass protocol as described in [draft-ietf-privacypass-protocol] can be integrated with a number of different settings, from server to server communication to browsing the internet.

In this document, we will provide an API to use for integrating Privacy Pass with an HTTP framework. Providing the format of HTTP requests and responses needed to implement the Privacy Pass protocol.

1.1. Terminology

We use the same definition of server and client that is used in [draft-ietf-privacypass-protocol] and [draft-ietf-privacypass-architecture].

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 messages.

1.2. Layout

- * Section 2: Describes the wrapping of messages within HTTP requests/responses.

- * Section 3: Describes how HTTP clients retrieve server configurations and key commitments.
- * Section 5: Describes how issuance requests are performed via a HTTP API.
- * Section 6: Describes how redemption requests are performed via a HTTP API.

1.3. Requirements

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].

2. Privacy Pass HTTP API Wrapping

Messages from HTTP-based clients to HTTP-based servers are performed as GET and POST requests. The messages are sent via the "Sec-Privacy-Pass" header.

"Sec-Privacy-Pass" is a Dictionary Structured Header [draft-ietf-httpbis-header-structure-15]. The dictionary has two keys:

- * "type" whose value is a String conveying the function that is being performed with this request.
- * "body" whose value is a byte sequence containing a Privacy Pass protocol message.

Note that the requests may contain addition Headers, request data and URL parameters that are not specified here, these extra fields should be ignored, though may be used by the server to determine whether to fulfill the requested issuance/redemption.

3. Server key registry

A client SHOULD fetch a server's current public key information prior to performing issuance and redemption. This configuration is accessible via a "CONFIG_ENDPOINT", either provided by the server or by a global registry that provides consistency and anonymization guarantees.

3.1. Key Registry

To ensure that a server isn't providing different views of their public key material to different users, servers are expected to write their commitments to a verifiable data structure.

Using a verifiable log-backed map ([verifiable-data-structures]), the server can publish their commitments to the log in a way that clients can detect when the server is attempting to provide a split-view of their key commitments to different clients.

The key to the map is the "server_origin", with the value being:

```
struct {
    opaque public_key<1..2^16-1>;
    uint64 expiry;
    uint8 supported_methods; # 3:Issue/Redeem, 2:Redeem, 1:Issue
    opaque signature<1..2^16-1>;
} KeyCommitment;

struct {
    opaque server_id<1..2^16-1>;
    uint16 ciphersuite;
    opaque verification_key<1..2^16-1>;
    KeyCommitment commitments<1..2^16-1>;
}
```

The addition to the log is made via a signed message to the log operator, which verifies the authenticity against a public key associated with that server origin (either via the Web PKI or a out-of-band key). The signature should be computed under a long-term signing key that is associated with the server identity.

The server SHOULD then store an inclusion proof of the current key commitment so that it can present it when delivering the key commitment directly to the client or when the key commitment is being delivered by a delegated party (other registries/preloaded configuration lists/etc).

The client can then perform a request for the key commitment against either the global registry or the server as described in Section 4. Note that the signature should be verified by the client to ensure that the key material is owned by the server. This requires that the client know the public verification key that is associated with the server.

To avoid user segregation as a result of server configuration/commitment rotation, the log operator SHOULD enforce limits on how many active commitments exist and how quickly the commitments are being rotated. Clients SHOULD reject configurations/commitments that violate their requirements for avoiding user segregation. These considerations are discussed as part of [draft-ietf-privacypass-architecture].

3.2. Server Configuration Retrieval

Inputs: - "server_origin": The origin to retrieve a server configuration for.

No outputs.

1. The client makes an anonymous GET request to "CONFIG_ENDPOINT"/.well-known/privacy-pass with a message of type "fetch-config" and a body of:

```
struct {  
    opaque server_origin<1..2^16-1>;  
}
```

1. The server looks up the configuration associated with the origin "server_origin" and responds with a message of type "config" and a body of:

```
struct {  
    opaque server_id<1..2^16-1>;  
    uint16 ciphersuite;  
    opaque commitment_id<1..2^8-1>;  
    opaque verification_key<1..2^16-1>;  
}
```

1. The client then stores the associated configuration state under the corresponding "server_origin".

(TODO: This might be mergable with key commitment retrieval if server_id = server_origin)

4. Key Commitment Retrieval

The client SHOULD retrieve server key commitments prior to both an issuance and redemption to verify the consistency of the keys and to monitor for key rotation between issuance and redemption events.

Inputs: - "server_origin": The origin to retrieve a key commitment for.

No outputs.

1. The client fetches the configuration state "server_id", "ciphersuite", "commitment_id" associated with "server_origin".
2. The client makes an anonymous GET request to "CONFIG_ENDPOINT"/.well-known/privacy-pass with a message of type "fetch-commitment" and a body of:

```
struct {  
    opaque server_id<1..2^16-1> = server_id;  
    opaque commitment_id<1..2^8-1> = commitment_id;  
}
```

1. The server looks up the current configuration, and constructs a list of commitments to return, noting whether a key commitment is valid for issuance or redemption or both.
2. The server then responds with a message of type "commitment" and a body of:

```
struct {  
    opaque public_key<1..2^16-1>;  
    uint64 expiry;  
    uint8 supported_methods; # 3:Issue/Redeem, 2:Redeem, 1:Issue  
    opaque signature<1..2^16-1>;  
} KeyCommitment;
```

```
struct {  
    opaque server_id<1..2^16-1>;  
    uint16 ciphersuite;  
    opaque verification_key<1..2^16-1>;  
    KeyCommitment commitments<1..2^16-1>;  
    opaque inclusion_proofs<1..2^16-1>;  
}
```

1. The client then verifies the signature for each key commitment and stores the list of commitments to the current scope. The client SHOULD NOT cache the commitments beyond the current scope, as new commitments should be fetched for each independent issuance and redemption request. The client SHOULD verify the "inclusion_proofs" to confirm that the key commitment has been submitted to a trusted registry. Once the client receives the "ciphersuite" for the server, it should implement all Privacy Pass API functions (as detailed in [draft-ietf-privacypass-protocol]) using this ciphersuite.

5. Privacy Pass Issuance

Inputs: - "server_origin": The origin to request token issuance from.
 - "count": The number of tokens to request issuance for.

Outputs: - "tokens": A list of tokens that have been signed via the Privacy Pass protocol.

1. When a client wants to request tokens from a server, it should first fetch a key commitment from the server via the process described in Section 4 and keep the result as "commitment".
2. The client should then call the "Generate" function requesting "count" tokens storing the resulting "input" data.
3. The client then makes a POST request to <"server_origin">/.well-known/privacy-pass with a message of type "request-issuance" and a body of:

```
enum { Normal(0) } IssuanceType;
```

```
struct {
    IssuanceType type = 0;
    opaque msg<0..2^16-1> = input.msg;
}
```

1. The server, upon receipt of the "request" should call the "Issue" function with the "public_key", "secret_key" and the value of "msg" with a result of "resp".
2. The server should then respond to the POST request with a message of type "issue" and a body of:

```
struct {
    IssuanceType type = request.type;
    IssuanceResp resp = resp;
}
```

1. The client should then should call the "Process" function with the "public_key", stored "inputs" and resulting "resp", to extract a list of "redemption_tokens".
2. The client should store the "public_key" associated with these tokens and the elements of "redemption_tokens" under storage partitioned by the "server_origin", accessible only via the Privacy Pass API.

6. Privacy Pass Redemption

There are two forms of Privacy Pass redemption that could function under the HTTP API. Either passing along a token directly to the target endpoint, which would perform its own redemption Section 6.1, or the client redeeming the token and passing the result along to the target endpoint. These two methods are described below.

In the HTTP ecosystem, redemption contexts should generally be keyed by the same privacy boundary used for cookies and other local storage. Generally this is the top-level origin. Any redemption context should be built following the principles outlined in [draft-ietf-privacypass-architecture] and later in Section 8.

6.1. Generic Token Redemption

Inputs: - "context": The request context to use. - "server_id": The server ID to redeem a token against. - "ciphersuite": The ciphersuite for this token. - "public_key": The public key associated with this token. - "redemption_token": A Privacy Pass token. - "info": Additional data to bind to this token redemption.

Outputs: - "result": The result of the redemption from the server.

1. The client should check whether the "server_id" is present in the "context". If it isn't and the size of the "context" is beneath the client's limit, it should be added.
2. The client should call the "Redeem" function with "redemption_token" and additional data of "info" storing the resulting "data" and "tag".
3. The client makes a POST request to <"server_origin">/.well-known/privacy-pass with a message of type "token-redemption" and a body of:

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

1. The server, upon receipt of "request" should call the "Verify" interface with "public_key", "secret_key" and the received "data", "tag", "info" storing the resulting "resp".

2. The server should then respond to the POST request with a message of type "redemption-result" and a signed body of:

```
struct {  
    opaque info<1..2^16-1> = info;  
    uint8 result = resp;  
    // signature of info and result using  
    // the server's verification key.  
    opaque signature<1..2^16-1>;  
}
```

1. The client upon receipt of this message should verify the "signature" using the "verification_key" from the configuration and return the "result".

6.2. Direct Redemption

Inputs: - "context": The request context to use. - "server_origin": The server origin to redeem a token for. - "target": The target endpoint to send the token to. - "additional_data": Additional data to bind to this redemption request.

1. When a client wants to redeem tokens for a server, it should first fetch a key commitment from the server via the process described in Section 4 and keep the result as "commitment".
2. The client should then look up the storage partition associated with "server_origin" and fetch a "redemption_token" and "public_key".
3. The client should verify that the "public_key" is in the current "commitment". If not, it should discard the token and fail the redemption attempt.
4. As part of the request to "target", the client will include the token as part of the request in the "Sec-Privacy-Pass" header along with whatever other parameters are being passed as part of the request to "target". The header will contain a message of type "token-redemption" with a body of:

```
struct {  
    opaque server_id<1..2^16-1> = server_id;  
    uint16 ciphersuite = ciphersuite;  
    opaque public_key<1..2^16-1> = public_key;  
    RedemptionToken token<1..2^16-1> = redemption_token;  
    opaque additional_data<1..2^16-1> = additional_data;  
}
```

At this point, the "target" can perform a generic redemption as described in Section 6.1 by forwarding the message included in the request to "target".

6.3. Delegated Redemption

Inputs: - "context": The request context to use. - "server_origin": The server origin to redeem a token for. - "target": The target endpoint to send the token to. - "additional_data": Additional data to bind to this redemption request.

1. When a client wants to redeem tokens for a server, it should first fetch a key commitment from the server via the process described in Section 4 and keep the result as "commitment".
2. The client should then look up the storage partition associated with "server_origin" and fetch a "redemption_token" and "public_key".
3. The client should verify that the "public_key" is in the current "commitment". If not, it should discard the token and fail the redemption attempt.
4. The client constructs a bytestring "info" made up of the "target", the current "timestamp", and "additional_data":

```
struct {  
    opaque target<1..2^16-1>;  
    uint64 timestamp;  
    opaque additional_data<0..2^16-1>;  
}
```

1. The client then performs a token redemption as described in Section 6.1. Storing the resulting "redemption-result" message.
2. As part of the request to "target", the client will include the redemption result as part of the request in the "Sec-Privacy-Pass" header along with whatever other parameters are being passed as part of the request to "target". The header will contain a message of type "signed-redemption-result" with a body of:

```
struct {  
    opaque server_origin<1..2^16-1>;  
    opaque target<1..2^16-1>;  
    uint64 timestamp;  
    opaque additional_data<1..2^16-1> = additional_data;  
    opaque signed_redemption<1..2^16-1>;  
}
```

At this point, the "target" can verify the integrity of "signed_redemption.info" based on the values of "target", "timestamp", and "additional_data" and verify the signature of the redemption result by querying the current configuration of the Privacy Pass server. The inclusion of "target" and "timestamp" proves that the server attested to the validity of the token in relation to this particular request.

7. Security Considerations

Security considerations for Privacy Pass are discussed in [draft-ietf-privacypass-architecture].

8. Privacy considerations

General privacy considerations for Privacy Pass are discussed in [draft-ietf-privacypass-architecture].

In order to implement this API with redemption contexts, a client needs to maintain strong privacy boundaries between different redemption contexts to avoid privacy leakage from redemptions across them. Notably in the web/HTTP world, cross-site tracking and fingerprinting will need to be considered and mitigated in order to maintain these privacy boundaries.

9. IANA Considerations

9.1. Well-Known URI

This specification registers a new well-known URI.

URI suffix: "privacy-pass"

Change controller: IETF.

Specification document(s): this specification

10. Normative References

- [draft-ietf-httpbis-header-structure-15]
Nottingham, M. and P-H. Kamp, "Structured Headers for HTTP", n.d., <<https://tools.ietf.org/html/draft-ietf-httpbis-header-structure-15>>.
- [draft-ietf-privacypass-architecture]
Davidson, A., "Privacy Pass: Architectural Framework", n.d., <<https://tools.ietf.org/html/draft-ietf-privacypass-architecture-00>>.
- [draft-ietf-privacypass-protocol]
Davidson, A., "Privacy Pass: The Protocol", n.d., <<https://tools.ietf.org/html/draft-ietf-privacypass-protocol-00>>.
- [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/rfc/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/rfc/rfc8446>>.
- [verifiable-data-structures]
"Verifiable Data Structures", n.d., <<https://github.com/google/trillian/blob/master/docs/papers/VerifiableDataStructures.pdf>>.

Author's Address

Steven Valdez
Google LLC

Email: svaldez@chromium.org

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 7 October 2022

S. Celi
Cloudflare
A. Davidson
Brave Software
A. Faz-Hernandez
Cloudflare
S. Valdez
Google LLC
C. A. Wood
Cloudflare
5 April 2022

Privacy Pass Issuance Protocol
draft-ietf-privacypass-protocol-04

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 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
2. Terminology	3
3. Configuration	3
4. Token Challenge Requirements	4
5. Issuance Protocol for Privately Verifiable Tokens with Public Metadata	4
5.1. Client-to-Issuer Request	5
5.2. Issuer-to-Client Response	6
5.3. Finalization	7
5.4. Issuer Configuration	8
6. Issuance Protocol for Publicly Verifiable Tokens	8
6.1. Client-to-Issuer Request	9
6.2. Issuer-to-Client Response	10
6.3. Finalization	11
6.4. Issuer Configuration	11
7. Security considerations	11
8. IANA considerations	11
8.1. Token Type	12
8.2. Media Types	12
8.2.1. "message/token-request" media type	12
8.2.2. "message/token-response" media type	13
9. Normative References	14
Appendix A. Acknowledgements	15
Appendix B. Test Vectors	15
B.1. Issuance Protocol 1 - VOPRF(P-384, SHA-384)	15
B.2. Issuance Protocol 2 - Blind RSA, 4096	16
Authors' Addresses	20

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.

Field Name	Value
issuer-request-uri	Issuer Request URI resource URL as a JSON string

Table 1

As an example, the Issuer's JSON directory could look like:

```
{
  "issuer-request-uri": "https://issuer.example.net/example-token-request"
}
```

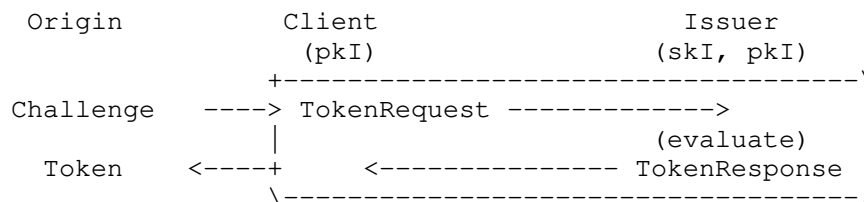
Issuer directory resources have the media type "application/json" and are located at the well-known location /.well-known/token-issuer-directory.

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:

```
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:

```
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:

```
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.

```
: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_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".

```
: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:

```
authenticator = client_context.Finalize(token_input, blind, evaluated_element, bl  
inded_element, proof)
```

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

```
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:

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

The key identifier for this specific key pair, denoted `key_id`, is computed as follows:

```
key_id = SHA256(0x0001 || SerializeElement(pkI))
```

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 `skI` and `pkI`, 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 `pkI`, 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:

```
nonce = random(32)
context = SHA256(challenge)
token_input = concat(0x0002, nonce, context, key_id)
blinded_msg, blind_inv = rsabssa_blind(pkI, token_input)
```

The rsabssa_blind function is defined in [BLINDRSA], Section 5.1.1.. The Client then creates a TokenRequest structured as follows:

```
struct {
    uint16_t token_type = 0x0002
    uint8_t token_key_id;
    uint8_t blinded_msg[Nk];
} TokenRequest;
```

The structure fields are defined as follows:

- * "token_type" is a 2-octet integer, which matches the type in the challenge.
- * "token_key_id" is the least significant byte of the key_id.
- * "blinded_msg" is the Nk-octet request defined above.

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

```
: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:

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

This is encoded and transmitted to the client in the following TokenResponse structure:

```
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".

```
: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:

```
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:

```
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 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.

Value	Name	Publicly Verifiable	Public Metadata	Private Metadata	Nk	Reference
0x0001	VOPRF (P-384, SHA-384)	N	N	N	48	Section 5
0x0002	Blind RSA, 4096	Y	N	N	512	Section 6

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

[AUTHSCHEME]

Pauly, T., Valdez, S., and C. A. Wood, "The Privacy Pass HTTP Authentication Scheme", Work in Progress, Internet-Draft, draft-pauly-privacypass-auth-scheme-00, 31 January 2022, <<https://datatracker.ietf.org/doc/html/draft-pauly-privacypass-auth-scheme-00>>.

[BLINDRSA] Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-rsa-blind-signatures-03, 2 February 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-rsa-blind-signatures-03>>.

[HTTP-Authentication]

"The Privacy Pass HTTP Authentication Scheme", n.d., <<https://datatracker.ietf.org/doc/html/draft-pauly-privacypass-auth-scheme-00>>.

[I-D.ietf-privacypass-architecture]

Davidson, A., Iyengar, J., and C. A. Wood, "Privacy Pass Architectural Framework", Work in Progress, Internet-Draft, draft-ietf-privacypass-architecture-03, 7 March 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-architecture-03>>.

[OPRF]

Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-09, 8 February 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-09>>.

- [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/rfc/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.

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.

- * token_request: 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.

```

skS: 0177781aeced893dcccdf80713d318a801e2a0498240fdcf650304bbbfd0f8d3b5c0
cf6cfee457aaa983ec02ff283b7a9
pkS: 022c63f79ac59c0ba3d204245f676a2133bd6120c90d67afa05cd6f8614294b7366
c252c6458300551b79a4911c2590a36
challenge:
a5d46383359ef34e3c4a7b8dlb3165778bffc9b70c9e6a60ddl4143e4c9c9fbd
nonce: 5d4799f8338ddc50a6685f83b8ecd264b2f157015229d12b3384c0f199efe7b8
blind: 0322fec505230992256296063d989b59cc03e83184eb6187076d264137622d202
48e4e525bdc007b80d1560e0a6f49d9
token_request: 00011a02861fd50d14be873611cfff0131d2c872c79d0260c6763498a2
a3f14ca926009c0f247653406eld52b68d61b7ed2bac9ea
token_response: 038e3625b6a769668a99680e46cf9479f5dc1e86d57164ab3b4a569d
dfc486bf1485d4916a5194fdc0518d3e8444968421ba36e8144aa7902705ff0f3cf40586
3d69451a2a7ba210cc45760c2f1a6045134d877b39e8bcbbf920e5de4a3372557debf211
765cd969976860bc039f9082d6a3e03f8e891246240173d2cf3d69a4613b0f8415979029
22e74c7a1f2e4639e4
token: 00015d4799f8338ddc50a6685f83b8ecd264b2f157015229d12b3384c0f199efe
7b8742cdfb0ed756ea680868ef109a280a393e001d2fa56b1be46ecb31fa25e76731a5b1
d698ea7ab843b8e8a71ed9b2fffa70457a43a8fc687939424b29a7554b40fde130ab7a82
2715909cb73f99a45b640ca1c85180ba9ca1a40bab8b664406a34bcbcb63b5e2e5c455cea
00001a968f7

```

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 6.1, 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_response: 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.

skS: 2d2d2d2d2d2d424547494e2050524956415445204b45592d2d2d2d0a4d49494a517
749424144414e42676b71686b6947397730424151454641415343435330776767b70416
74541416f4943415144584d4d364c5073637a6130696b0a39596c315a4e683868324a796
d504962704c383035354d52516f6463446c4c4672764d694c57666f59535241506b4e744
1546272324162654e334d454a2f67550a392f424958717a704d4256484852546c627a676
b364c7866766b54505a2b4a427832517177364f5555714b442f34426f5464765761536d6
a63644165555037760a4836376e396a383836307463367375546e67616c574d4b6d764d2
f5a6237644262543763345647386577706c6776444f6d616641353956354f78505545704
9510a586c454f4771414f43436c316f54686d33363836436c48413352374c5a4d78567a4
742386f594537583548396a70793532786a2f3242724a6861625033567a430a6f4c696c3
54f6e36487158785363466d4956573742787956495979756d75537659544e52757652777
3566c3775595446366f4d6d2b59722f5a506372594c7a510a4e666135634f747533532b6
649594456716f6f58375541415671382f716c4945747a794a744f7261616756393039327
0324f474c4f6d306a52384543666963520a3868363332572f76554d7739724c4c317a4d4
e7554424f4b74316c546c307265644a677668626b66515a5a55303541336a69366c71754
d2f47307250553030470a3369385253732f64694e625843505453746b616f45537350345
946496d526269756c786a544e2b626c5859744868494261556774306e2b566b544a77554
86e380a5367447534664e547142776567517265494e427732446b6e63576e676b5644416
867577635383669727351644c345843723376765856647a51375134586978730a465a6b6
d7763676d665142444f3469363078536c336337316954595362783359326e74584b4e6f5
a4c31537a424f595051496a6c734749454250677157546e5a0a64655a7852464f6c4d384
679794b6d30746471586271594b57716b663777494441514142416f4943414364314d707
044773645424467763558654168777252710a32726c717042492f6a6a50304e6e7057755
a302b6e787a53624a4361784d2f4f61436844676e654e586e573259655144524e7242505
84d5331344e646f4e554e0a56516c36497166445567637169636741696e7542622f4a687
a6c4d74466d5350466d2b667650726a4d2b6c4831544f384863354253633274414a52574
36468770a794a7663446349656d74646378437877754b6746485251704a5071356368526
56431535077766f50494c7831686959356c2f5175423478717a764149483873530a34673
949705a2f766169443558573541335847734a4f2b696a78717250706756655236474e4f5
33763515551716a44446967665a626a5864354a322b734d79460a4c435a6e514d6771577
03731756437366a4f4a445571563537454154534e6b56472f52633279537a554b4d6e354
c335a314a796d6d31694f584a5136535744760a6f5375426b435652436645635a746b546
f436c76646f5456666b63414a394e51343839686b3050754e466f593675315671614c5a4
a7764657a3931764f5966300a4b676f665a4a6e774976547733754b786f7559685033334
43644574a4f31763265487658767a4c744f4a54477054446f726f4c7a2f475459316b682

f484d50510a416c4f333758772f526b527752684b4d6f39545555347766642b2b5348325
4346e6730584875355254764d5339496b4266724455337a75557675434b7270356b0a377
942364473763433517932364f6367367151584a563650676b6c5770696164445778326f4
73935746578436f3346563434764a6d49327a706d5748514257630a48395331336b6d7a3
06873506274576a537535494c443061694f5630764d4c61353841685a767a32774562763
750546a494d48364d77446736326d6450744d6b0a534969517548644a38326a69316e757
8646f5678416f4942415144773431456757476e437644496239727a73704b6e61486a654
3574650385737664a6b446c4c0a61786c724a6f574e614d654c33306b646873576569765
7616730436f457656556631543455374936735a33705054675653685079374b6f4c645a4
86f757a5a390a4d774679716d694652726e574135467138544b756f4354647a3836542f5
37859753330625a476c6d6a4c432f6664453279556141486f6f5177375a2f34646b6d0a4
1693031696a36484d654e39486e5a425737656c436c4173443242436d4b5279655554793
7754b472b6b3732704630336e45694f7a42615565354d6c7a7436620a3867586b63715a5
735495478454b41726b6d486a77454379765178346a7867324573326a7653654e3041554
b624a63534a736e7a35434a65667a697561676f650a36482b3947676f62386e496a78546
34e784c7a58316a676e5236772f41302f6e657837714e6b57357773485a46644c58416f4
9424151446b734d6e75503452460a7558474a6e34316559696c43716b694e4e6b4a53753
9637456556b7772364570744230506d34556e3770535238684866767a5a43782f6b65766
b33553336546e0a67556c72416a4b72474a6a77437231457a7252726e417153466f61395
04851395a69697264626336726f474932576f63795a585859664b784a56646a4f754c440
a793947564d72575133665a3571696d7a6a394b4163304852455765784b6875514259465
554542f44323169584b4b65497568647834796b795435323857714b310a66786f4c72584
e6d5936726b77787778763334556779685a54564a75454879506e51422f54743242377a2
b342f763033665a46347361635646337738514a306c0a692b466879555a34326b63674a3
52b654c6d5558726b6d5a5959314a534332417a722f336449516a4b554655515935326a3
65775667338392f576a72516669470a31732b6c51385a7a6d4a4370416f49424151432b3
84c43686e764e574e4b37546b365431507943546b466758726351457950374a657453766
6316c4b6f654a430a304d6337692b58387a5a4e6674473478353941636163716c43376c6
96a5a55394351564f6d415159652f754d4679524371524c62453271426d79694f70357a7
00a3538487562693261513034564e554f44767644555256346468364148556e52706f534
f49356b59723079646137746f7070376a4662565165324b4c56535a742b0a746f4448384
a6c7a2f5374345774427033465a4538354747573748586a70746f756f6855344c777a464
7492f4c6d37486935783733357038716a385a6366642f0a384f756632626e6354382f674
938676b35633034307451794b483177534d4e4e6d5a496c54535943635653725369346b4
5567677684955354d726e75507647380a6256556b48584d694b7377316d63777739704d4
6373634716f6d464337586f66594d30664d6a6c4a416f494241465079565632676354534
b2b784e7976786b4c0a58577638522f2b57454569416256395674445572387a503079736
f6b34333869412b57432f32367271515a676b36446d61486d673073367356622f7a495a6
84f0a7769307a4e41446a41375751705179314f6961533333522b594b56333435656c345
35454386a433543736a79536e306559504b71396679376636614e343770570a304267664
4346d37584b454d4c66664a744d2b437a6e56536f41504c433449677257646e5a4141376
c306d57416c52576832645275664a3377703751763943770a2b315659446178785235334
e2b327930686e4b696d4b613745696970555952567834566f444a6c6d2f5a525a5769545
335796253375279514f563645334e71560a2f592f6647366563446a33674732497a50674
c4e664f3651646b556d7679364e4155386c645638754962707045446749497042684f684
a394865486a664343490a75326b4367674542414d68686e6f69312b78454a365339636e5
a327977527737433057544962662b54557230436e374f344a4d76637843544b484d62773
76a680a574c6247392f314732595966354c5673326b572f34472b2f446d586b6d4b68715

4746d6f324e50463666504b73694d64477877354149677039557a50543168550a2f6c777
26a75474d322f2b77434b70316d3645374e4d4d5659684b5841534f673473652b3676586
455356a56436f634343576162657a6c5835513262796c51480a2b2b624f6d4661504d535
5683761616d6657357573614553627366612f45506932446d3545574f456339567577525
671526143527552534632507043627279410a6b376e6b6b6746474b5a523777353053465
5366f7a776667627a2b7a33637256315535766d3076346c63794b6d524b6c4b575a51554
9624b782f5070583737640a395057536e3569594343704c432f316245372f566f4c70467
7757631656a773d0a2d2d2d2d2d454e442050524956415445204b45592d2d2d2d2d0a
pkS: 30820252303d06092a864886f70d01010a3030a00d300b060960864801650304020
2a11a301806092a864886f70d010108300b0609608648016503040202a20302013003820
20f003082020a0282020100d730ce8b3ec7336b48a4f5897564d87c87627298f21ba4bf3
4e7931142875c0e52c5aef3222d67e86124403e436d0136ebd806de37730427f814f7f04
85eace93015471d14e56f3824e8bc5f5be44cf67e241c7642ac3a39452a283ff80684ddb
66929a371d01e50fee7f1faee7f63f3ceb4b5ceacb939e06a558c2a6bccfd96fb7416d3ed
ce151bc7b0a6582f0ce99a7c0e7d5793b13d41292105e510e1aa00e082975a13866dfaf3
a0a51c0dd1ecb64cc55cc607ca1813b5f91fd8e9cb9db18ffd81ac985a6cfdd5cc2a0b8a
5e4e9falea5f149c1662155bb071c95218cae9ae4af613351baf470b1597bb984c5ea832
6f98aff64f72b60bcd035f6b970eb6edd2f9f2180d5aa8a17ed400056af3faa5204b73c8
9b4eada6a057dd3dda9d8e18b3a6d2347c1027e2711f21eb7d96fef50cc3dacb2f5ccc36
e4c138ab75953974ade74982f85b91f419654d390378e2ea5aae33f1b4acf534d06de2f1
14acfd88d6d708f4d2b646a8112b0fe181489916e2ba5c634cdf9b95762d1e120169482
dd27f959132705079fc4a00eeef353a81c1e810ade20d070d839277169e09150c08605a
fe7cea2aec41d2f85c2af7bef5d577343b4385e2c6c159926c1c8267d00433b88bad314a
5ddcef58936126f1dd8da7b5728da192f54b304e60f4088e5b0620404f82a5939d975e67
14453a533c172c8a9b4b5da976ea60a5aa91fef0203010001
challenge:
83ce743dcdadd5fc4aeb0357977bb8426635c390a15b88947f0b1c62e4a87c22
nonce: 7e0da97bfcdc4365a5f40e69262f78b81bcd2f92daf885358d9831874e3dd9d22
blind: cd6d03e332386d0166eb76b8e78522510e5cbdcf49aaac83191ea948a7719e914
0ccb6701f7301b7d445ede7adbc5e582b35edd9ac45bc4b8f794e150b2e3e407b7b7624b
6f90b33845bc255174cee0c570aa781c203dce8563afe9f48e2b49c773bba1031987fb48
d981d131876f53e264ec0609a3ea628cf2042005ed3071aeb6657472c7e7df947915b8cd
333e3f5078e456e65e5edef8f892c4f21d25a18dcd80628ed6c7d55b0b9433bc67760be0
8a4eacbdb16a4be4c5b8cab26b478fa6a36ea3c3dd1fffb420bf69feef52aab4892c9e60a
df18347b4e8256b5a0e8cbf55fc97ac62af2e7349ba98ca7462cb6a41d70b0217814a06e
1b257289c3b345be652b87d5820b06a80500880b40b8772140bf431f11497114b20fee7e
5ffc1af5cf874cc293a0c8df65d52814bcd55ae6d3701f73d140ca82c6528627129ea389
f3cbd6058f4f80b7df3818f36dd3489259b6b95df4511930ff02b5cbe643fea44306e7c4
e3d9b02f1b0559aa238b8882a6e8791bfbfd366ef4fe433fd42e5c5db208c9fceb74def
11663ce5f793c7013116995b3fef392a8633b08179a9c8309fb69359fd8486a7a8febb42
4d0726c2516b11e8b19a55fa54e9be606de6811059976473de8f9adb25af7e2862932bed
c7764b4dc50bfc9d724a4aba356a7677b5aceef21876e56b4f1b65adef0fbf8bf1636815
be01b372727e79aa6c47f41
salt: d13a47fa6466a37203e51ac34f7319831b3f04202ff74c98ab18e78088b7ac3014
06189783503227153871405c6alda0
token_request: 0002013a370077e8259098e741dcaac8184838b7c995cd82966419064
90205bf28e6745396dd9ec1761c4676e9fec3272588194c48bc60fc77c3fff19219a6b65
96523044d07d9d9e4dd88f2db9d9c369597363a12a65d244d69a1b743b365f8f5bdf8d6

```
52f2d2e249f417e9fd7da7db2626d6499d7d3856d8f9277385dfd776ffa4ffa74d7a7b17
01d87f40e525bb258a7f5b4d6c134b3b242ef46d93b32f106c84c396b1fc2772796b2473
64c48c364537708f3a8a87fb870a299ac08107a5dd3f467733d76c2359e346ee3ebcff68
c3c7e10f0f01a2b9bbdb26ffea14f81f036a71c47725b5c9f81e0320b85abfd77d5eb1c
ccfdd8eb0cf2735ea297e2a07c82dfce9b0a4d21baa81a2cfb2a72983107555035386c33
973d48f04257dd8298116d2c93298810cb6b82ad033f5b16f9f7a65d8f74b7bdcae000db
1411b40f46cb373cb69c8ab58552f98904b78229b63838ab40e833fab4ec47acf00a00db
3290c9c74ed982c64ddbaecae16fd73976ffe7da7b3b1a8e0190e95b7ce7900295f3c8c94f
2d3d85cd1825fe27073aad63fa4c530907402dc4e3a748edb300f05ce7ac5b8c1d9aeb21
66002b05dee582aeafa4f503f13bae1a51be9420e3cdcb685169bdc5ae2ebca7713ec16
666b55b097e56e5719d1b0324ecaca613af76b9f367775a90dba5e7fdd21a8da73bd80b1
31e6531117ef709ad8c7b2b3182255235acea
token_response: 061780e09bc9b851fe81e7022ee2d55b043198bcb1aa33f761d213a9
8d831abaef5417d30904a7c9d7ec531278cd9655c4b7d72f3a4e66e26565b73e5f1b9271
b541f28556543d2473c363e104a11c6ba1a1d8f99b32d3cd8f74ae07b465afdaabd21977
d6f114ea9484cd592f461e5dc4a97b86fe1463b1c69171cf734f49c240760dc2555d7cc8
9d7f882d3e1b99388bc3b561d418a7fc5770ccc66e3dd41f4a74e8267492f48e8b6aabce
592c8dc83826b1f4528f2497902a0ce4baacd4216623274057592e77091102452de115bb
d0c98ee22b14fe30fb3b1277ab17bc948b62b8adb56b67e44dce74860647718c8f4bd10e
7b022ad35e2996dd497f4a6c765689931fc11b56b06cca921fb1205e8b2302540a48da
38ed7a5e20f7aa80b55f1de9f5a6c9589f4fc23b6ed2c53ad2a6a64abb8fba67aef48e66f
f77647aa05ce96527e2d1199b7553a3b900ec510f5b765211c14d0f6cbe82f28be4d8ac
8110f13d7aee3741fc68f4abf3ca33444f790a72ef4ddec72c1d92938ee5d303f914ff12
8d9a73e867d9aa5d327934d4b2b299d7ffa32350d72d35103c027ac76c5417823e6790ee
174b9761e086bfa445f1725b20cff6a94d51abdfcfe46ff7c0f69685fd38639bdbd3917f
424949027b5a322fc217364748e544257d11f0cdcab9aal3d3282f6ac2811dfc8db5b2f99
c1ba00c366057748cabe6975fc73ed60
token: 00027e0da97bfcdc4365a5f40e69262f78b81bcd2f92daf885358d9831874e3dd9
d22895c9d3fff72e71d22cdabc11706d350ab772b0820be9f33a02e003652cb00a31501000
00000000000000000000000000000000000000000000000000000000000000000004c9e19f2a624e
e7d44ac35846749b29b1d3a784f13ea1005a2c87b9d3f939f795877d5fded823f45fd399
dc0b8730cb46c66102740c679338b7093c47e8f586e48a8b042cb0ea2b901f6981e797c4
a614f52f02ffe3ae7f83fa4f9a243e8b59621975abff94f82b3fadfb4cb305cc1c1b677
42a673204e5a0aa0c25423c604430597d0332e30dddab8855de42bcf49668410df38bb3b
fa4370df28ec59316bc1c6f3c9afc8ccfdb93f4ca60365683988f649201e1c6d6bb73d14
d0dc9a0f596a9f76502ebadc0b248985f9bb66d9d99a5aca08527aa11d555b26489299ba
5b400157a9fd47b6b4fa74315eade2b22624b29d53eb84126f64b98ea5ba45914d1fa14b
1525e2327856565054a1db9b0d778871fa6ed4d0d4c26641bf3f4faa33efaa0f5b8cec80
8d52ed3f1378273d5b7b0b0b812bfc128ef5e4924a60aebd124659d31661e9ec89f8bcb9
a51bab6a5711187639c24fdd31f14abf7d80827df91f31bfe7c4916ec4d1927ca138c5ba
9a595a9e83b5055148d19ad005c523eda76ea94006ce6315e20ed0d637fb1211b541e9ea
12c9b641d48fd2cc5f0c7f479672a4e2bf7469267c8526d734df41f2c30fb62c2aea4033
214df44a53353dc683cf72dee7b1ba39ef668478958935a0e8c9a880ae85712c401d7f09
b66fbdad05cfd69d615b229bce8818c6a6472e07a8793456f19f4f4015c507ab5c1357881
68b
```

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