

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

A. Davidson  
Cloudflare Portugal  
13 July 2020

Privacy Pass: Architectural Framework  
draft-davidson-pp-architecture-01

Abstract

This document specifies the architectural framework for constructing secure and anonymity-preserving instantiations of the Privacy Pass protocol. It provides recommendations on how the protocol ecosystem should be constructed to ensure the privacy of clients, and the security of all participating entities.

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 . . . . .	3
3. Ecosystem participants . . . . .	4
3.1. Servers . . . . .	5
3.2. Clients . . . . .	6
3.2.1. Client identifying information . . . . .	6
4. Key management framework . . . . .	6
4.1. Public key registries . . . . .	7
4.2. Key rotation . . . . .	8
4.3. Ciphersuites . . . . .	8
5. Server running modes . . . . .	9
5.1. Single-Verifier . . . . .	9
5.2. Delegated-Verifier . . . . .	9
5.3. Asynchronous-Verifier . . . . .	10
5.4. Public-Verifier . . . . .	10
5.5. Bounded number of servers . . . . .	10
6. Client-Server trust relationship . . . . .	11
7. Privacy considerations . . . . .	12
7.1. Server key rotation . . . . .	12
7.2. Large numbers of servers . . . . .	13
7.2.1. Allowing larger number of servers . . . . .	13
7.3. Partitioning of server key material . . . . .	14
7.4. Additional token metadata . . . . .	14
7.5. Tracking and identity leakage . . . . .	15
7.6. Client incentives for anonymity reduction . . . . .	15
8. Security considerations . . . . .	15
8.1. Double-spend protection . . . . .	16
8.2. Token exhaustion . . . . .	16
8.3. Avoiding server centralization . . . . .	16
9. Protocol parametrization . . . . .	16
9.1. Justification . . . . .	17
9.2. Example parameterization . . . . .	18
9.3. Allowing more servers . . . . .	19
10. Extension integration policy . . . . .	19
11. Existing applications . . . . .	19
11.1. Cloudflare challenge pages . . . . .	19
11.2. Trust Token API . . . . .	20
11.3. Zero-knowledge Access Passes . . . . .	20
11.4. Basic Attention Tokens . . . . .	20
11.5. Token Based Services . . . . .	20
12. References . . . . .	20
12.1. Normative References . . . . .	20
12.2. Informative References . . . . .	21
Appendix A. Contributors . . . . .	21
Author's Address . . . . .	21

## 1. Introduction

The Privacy Pass protocol provides an anonymity-preserving mechanism for authorization of clients with servers. The protocol is detailed in [draft-davidson-pp-protocol] and is intended for use in the application-layer.

The way that the ecosystem around the protocol is set up can have significant impacts on the stated privacy and security guarantees of the protocol. For instance, the number of servers issuing Privacy Pass tokens, along with the number of registered clients, determines the anonymity set of each individual client. Moreover, this can be influenced by other factors, such as: the key rotation policy used by each server; and, the number of supported ciphersuites. There are also client behavior patterns that can reduce the effective security of the server.

In this document, we will provide a structural framework for building the ecosystem around the Privacy Pass protocol. The core of the document also includes policies for the following considerations.

- \* How server key material should be managed and accessed.
- \* Compatible server issuance and redemption running modes and associated expectations.
- \* How clients should evaluate server trust relationships.
- \* Security and privacy properties of the protocol.
- \* A concrete assessment and parametrization of the privacy budget associated with different settings of the above policies.
- \* The incorporation of potential extensions into the wider ecosystem.

Finally, we will discuss existing applications that make use of the Privacy Pass protocol, and highlight how these may fit with the proposed framework.

## 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.

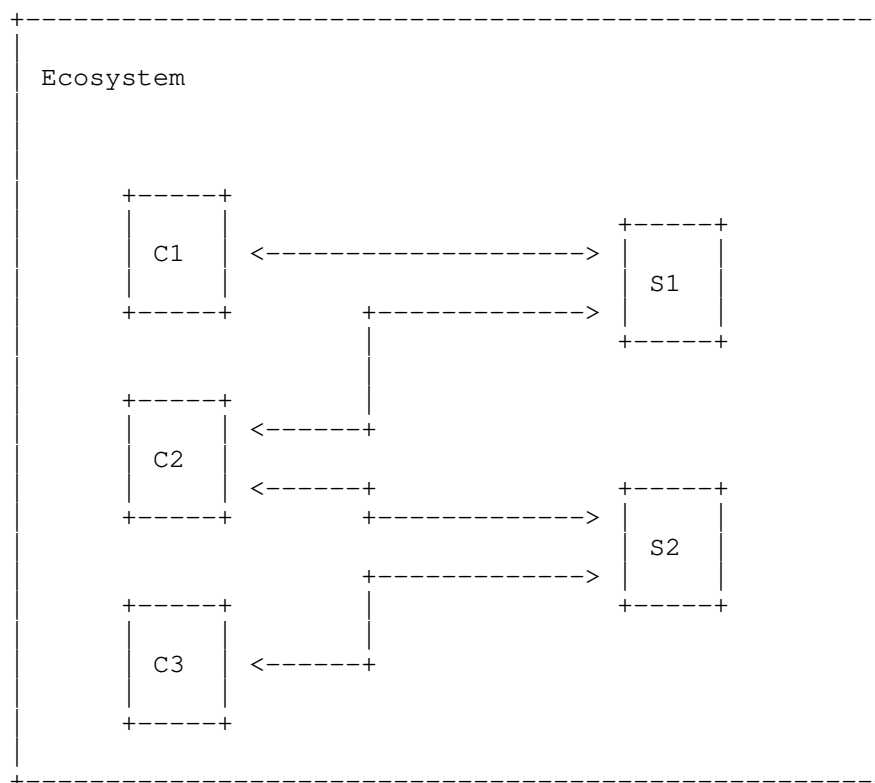
- \* Server: An entity that issues anonymous tokens to clients. In symmetric verification cases, the server must also verify tokens. Also referred to as the server.
- \* Client: An entity that seeks authorization from a server.

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 the data that is communicated and stored.

### 3. Ecosystem participants

The Privacy Pass ecosystem refers to the global framework in which multiple instances of the Privacy Pass protocol operate. This refers to all servers that support the protocol, or any extension of it, along with all of the clients that may interact with these servers.

The ecosystem itself, and the way it is constructed, is critical for evaluating the privacy of each individual client. We assume that a client's privacy refers to fraction of users that it represents in the anonymity set that it belongs to. We discuss this more in Section 7.



In the above diagram, the arrows indicate the open channels between a client and a server. An open channel indicates that a client accepts Privacy Pass tokens from this server.

If no channel exists, this means that the client chooses not to accept tokens from (or redeem tokens with) that particular server. We discuss the roles of servers and clients further in Section 3.1 and Section 3.2, respectively.

### 3.1. Servers

Generally, servers in the Privacy Pass ecosystem are entities whose primary function is to undertake the role of the "server" in [draft-davidson-pp-protocol]. To facilitate this, the server **MUST** hold a Privacy Pass protocol keypair at any given time. The server public key **MUST** be made available to all clients in such a way that key rotations and other updates are publicly visible. The server **MAY** also require additional state for ensuring this. We provide a wider discussion in Section 4.

Note that, in the core protocol instantiation from [draft-davidson-pp-protocol], the redemption phase is a symmetric protocol. This means that the server is the same server that ultimately processes token redemptions from clients. However, plausible extensions to the protocol specification may allow public verification of tokens by entities which do not hold the secret Privacy Pass keying material. We highlight possible client and server configurations in Section 5.

The server must be uniquely identifiable by all clients with a consistent identifier.

### 3.2. Clients

Clients in the Privacy Pass ecosystem are entities whose primary function is to undertake the role of the "Client" in [draft-davidson-pp-protocol]. Clients are assumed to only store data related to the tokens that it has been issued by the server. This storage is used for constructing redemption requests.

Clients MAY choose not to accept tokens from servers that they do not trust. See Section 6 for a wider discussion.

#### 3.2.1. Client identifying information

Privacy properties of this protocol do not take into account other possibly identifying information available in an implementation, such as a client's IP address. Servers which monitor IP addresses may use this to track client redemption patterns over time. Clients cannot check whether servers monitor such identifying information. Thus, clients SHOULD minimize or remove identifying information where possible, e.g., by using anonymity-preserving tools such as Tor to interact with servers.

### 4. Key management framework

The key material and protocol configuration that a server uses to issue tokens corresponds to a number of different pieces of information.

- \* The ciphersuite that the server is using.
- \* The public keys that are active for the server.

For reasons that we address later in Section 7, the way that the server publishes and maintains this information impacts the effective privacy of the clients. In this section, we describe the main

policies that need to be satisfied for a key management system in a Privacy Pass ecosystem.

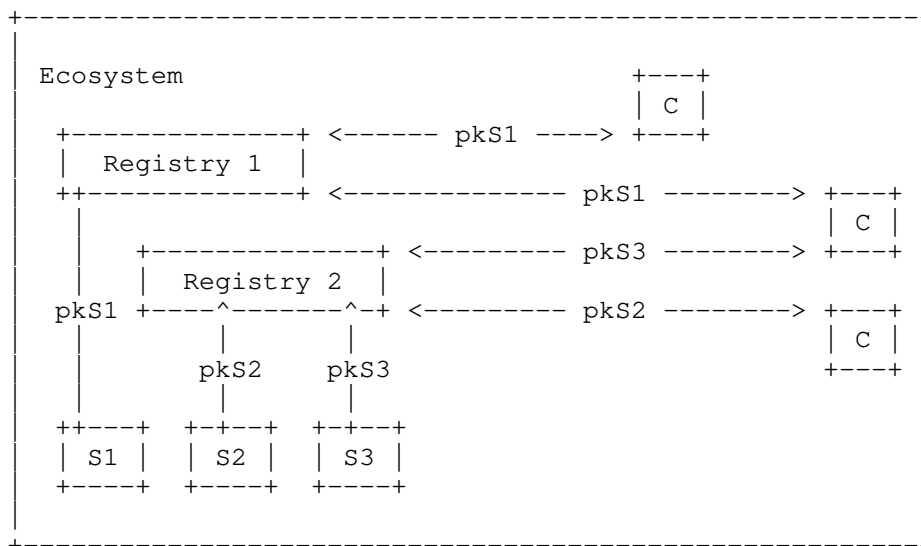
Note that we only specify a set of guidelines and recommendations for operating a public key registry in this document. Actual specification of such a registry and how it operates will be covered elsewhere.

#### 4.1. Public key registries

Server's must provide their public keys to clients along with details about the cryptographic ciphersuite that they are using. In Section 7, we address the importance of providing clients with sources of truth for learning the server's key configuration.

In particular, server key material **MUST** be publicly available in a tamper-proof data structure, which we refer to as a key registry. A registry must be globally consistent. Clients using the same registry should coordinate in some way to ensure they have a consistent view of said registry. This can be done via gossiping or some other mechanism. The exact mechanism for this coordination will be described elsewhere. It is assumed there will be at least one such mechanism.

It is **RECOMMENDED** that each key registry is an append-only data structure, such as a Merkle Tree. The key registry should be operated independently of any server that publishes key material to the registry. This ensures that any client can make better judgements on whether to trust the registry and, transitively, each server.



While there may be multiple key registries for a given ecosystem, a server **MUST** only publish its key material to a single registry. This ensures that the server is keeping a consistent view of its key material.

#### 4.2. Key rotation

Token issuance associates all issued tokens with a particular choice of key. If a server issues tokens with many keys, then this may harm the anonymity of the Client. For example, they would be able to map the Client's access patterns by inspecting which key each token they possess has been issued under.

To prevent against this, servers **MUST** only use one private key for issuing tokens at any given time. Servers may use two or more keys for redemption to allow servers for seamless key rotation.

Key rotations must be limited in frequency for similar reasons. See Section 9 for guidelines on what frequency of key rotations are permitted.

#### 4.3. Ciphersuites

Since a server is only permitted to have a single active issuing key, this implies that only a single ciphersuite is allowed per issuance period. If a server wishes to change their ciphersuite, they **MUST** do so during a key rotation.

## 5. Server running modes

We provide an overview of some of the possible frameworks for configuring the way that servers run in the Privacy Pass ecosystem. In short, servers may be configured to provide symmetric issuance and redemption with clients. While some servers may be configured as proxies that accept Privacy Pass data and send it to another server that actually processes issuance or redemption data. Finally, we also consider instances of the protocol that may permit public verification.

The intention with providing each of these running modes is to cover the different applications that utilize variants of the Privacy Pass protocol. We RECOMMEND that any Privacy Pass server implementation adheres to one of these frameworks.

### 5.1. Single-Verifier

The simplest way of considering the Privacy Pass protocol is in a setting where the same server plays the role of server and verifier, we call this "Single-Verifier" (SV).

Let  $S$  be the server, and  $C$  be the client. When  $S$  wants to issue tokens to  $C$ , they invoke the issuance protocol where  $C$  generates their own inputs, and  $S$  uses their secret key  $sk_S$ . In this setting,  $C$  can only perform token redemption with  $S$ . When a token redemption is required,  $C$  and  $S$  invoke the redemption phase of the protocol, where  $C$  uses an issued token from a previous exchange, and  $S$  uses  $sk_S$  to validate the redemption.

### 5.2. Delegated-Verifier

In this setting, each client  $C$  obtains issued tokens from a server  $S$  via the issuance phase of the protocol. The difference is that  $C$  can prove that they hold a valid authorization with any verifier  $V$ . We still only consider  $S$  to hold their own secret key. We name this mode "Delegated-Verifier" (DV).

When  $C$  interacts with  $V$ ,  $V$  can ask  $C$  to provide proof of authorization to the separate server  $S$ . The first stage of the redemption phase of the protocol is invoked between  $C$  and  $V$ , which sees  $C$  send an unused redemption token to  $V$ . This message is then used in a redemption exchange between  $V$  and  $S$ , where  $V$  plays the role of the Client. Then  $S$  sends the result of the redemption verification to  $V$ , and  $V$  uses this result to determine whether  $C$  has a valid token.

### 5.3. Asynchronous-Verifier

This setting is inspired by recently proposed APIs such as [TrustTokenAPI]. It is similar to the DV configuration, except that the verifiers V no longer interact with the server S. Only C interacts with S, and this is done asynchronously to the authorization request from V. Hence "Asynchronous-Verifier" (AV).

When V invokes a redemption for C, C then invokes a redemption exchange with S in a separate session. If verification is carried out successfully by S, S instead returns a Signed Redemption Record (SRR) that contains the following information:

```
"result": {  
  "timestamp": "2019-10-09-11:06:11",  
  "verifier": "V",  
},  
"signature": sig,
```

The "signature" field carries a signature evaluated over the contents of "result" using a long-term signing key for the server S, of which the corresponding public key is well-known to C and V. This would need to be published alongside other public key data for S. Then C can prove that they hold a valid authorization from S to V by sending the SRR to V. The SRR can be verified by V by verifying the signature, using the well-known public key for S.

Such records can be cached to display again in the future. The server can also add an expiry date to the record to determine when the client must refresh the record.

### 5.4. Public-Verifier

We consider the case where client redemptions can be verified publicly using the server public key. This allows for defining extensions of Privacy Pass that use public-key cryptography to allow public verification.

In this case, the client C obtains a redemption token from S. The redemption token is publicly verifiable in the sense that any entity that knows the public key for S can verify the token. This running mode is known as "Public-Verifier" (PV).

### 5.5. Bounded number of servers

Each of the configurations above can be generalized to settings where a bounded number of servers are allowed, and verifiers can invoke authorization verification for any of the available servers.

As we will discuss later in Section 7, configuring a large number of servers can lead to privacy concerns for the clients in the ecosystem. Therefore, we are careful to ensure that the number of servers is kept strictly bounded. The actual servers can be replaced with different servers as long as the total never exceeds this bound. Moreover, server replacements also have an effect on client anonymity that is similar to when a key rotation occurs. server so replacement should only be permitted at similar intervals.

See Section 7 for more details about maintaining privacy with multiple servers.

## 6. Client-Server trust relationship

It is important, based on the architecture above, that any client can determine whether it would like to interact with a given server in the ecosystem. Note that this decision must be taken before a client issues a valid redemption to the server, since redemptions reveal the anonymity set that the client belongs to.

This judgement can be based on a multitude of factors, associated with the way that a server presents itself in the ecosystem. A non-exhaustive list of server characteristics that a client MAY want to check are the following.

- \* Which key registry a server posts their key updates to.
- \* How frequent key updates are issued, and which ciphersuite they use.
- \* The reason given to initiate the redemption.

To aid client trust decisions, a server can publish a "Privacy Pass policy" that documents the procedures that the server uses to ensure that client privacy is respected. If a server does not publish such a document then the client may choose to use its own judgement, or to reject the server altogether.

It should be noted that the client trust decision can be made apriori by specifying an allow-list of all servers that it accepts tokens from. This means that these checks do not have to be performed online.

## 7. Privacy considerations

In the Privacy Pass protocol [draft-davidson-pp-protocol], redemption tokens intentionally encode very little information beyond which key was used to sign them. The protocol intentionally uses components that provide cryptographic guarantees of this fact. However, even with these guarantees, the way that the ecosystem is constructed can be used to identify clients based on this limited information.

The goal of the Privacy Pass ecosystem is to construct an environment that can easily measure (and maximize) the relative anonymity of any client that is part of it. An inherent feature of being part of this ecosystem is that any client can only remain private relative to the entire space of users using the protocol. Moreover, by owning tokens for a given set of keys, the client's anonymity set shrinks to the total number of clients controlling tokens for the same keys.

In the following, we consider the possible ways that servers and servers can leverage their position to try and reduce the anonymity sets that clients belong to (or, user segregation). For each case, we provide mitigations that the Privacy Pass ecosystem must implement to prevent these actions.

### 7.1. Server key rotation

Techniques to introduce client "segregation" can be used to reduce client anonymity. Such techniques are closely linked to the type of key schedule that is used by the server. When a server rotates their key, any client that invokes the issuance protocol in this key cycle will be part of a group of possible clients owning valid tokens for this key. To mechanize this attack strategy, a server could introduce a key rotation policy that forces clients into small key cycles. Thus, reducing the size of the anonymity set for these clients.

We RECOMMEND that servers should only invoke key rotation for fairly large periods of time such as between 1 and 12 weeks. Key rotations represent a trade-off between client privacy and continued server security. Therefore, it is still important that key rotations occur on a fairly regular cycle to reduce the harmfulness of a server key compromise.

With an active user-base, a week gives a fairly large window for clients to participate in the Privacy Pass protocol and thus enjoy the anonymity guarantees of being part of a larger group. The low ceiling of 12 weeks prevents a key compromise from being too destructive. If a server realizes that a key compromise has occurred then the server should sample a new key, and upload the public key to

the key registry; invoking any revocation procedures that may apply for the old key.

## 7.2. Large numbers of servers

Similarly to the server rotation dynamic that is raised above, if there are a large number of servers then segregation can occur. In the FV, AV and PV running modes (Section 5), a verifier OV can trigger redemptions for any of the available servers. Each redemption token that a client holds essentially corresponds to a bit of information about the client that OV can learn. Therefore, there is an exponential loss in anonymity relative to the number of servers that there are.

For example, if there are 32 servers, then OV learns 32 bits of information about the client. If the distribution of server trust is anything close to a uniform distribution, then this is likely to uniquely identify any client amongst all other Internet users. Assuming a uniform distribution is clearly the worst-case scenario, and unlikely to be accurate, but it provides a stark warning against allowing too many servers at any one time.

In cases where clients can hold tokens for all servers at any given time, a strict bound SHOULD be applied to the active number of servers in the ecosystem. We propose that allowing no more than 4 servers at any one time is highly preferable (leading to a maximum of 64 possible user segregations). However, as highlighted in Section 9, having a very large user base (> 5 million users), could potentially allow for larger values. server replacements should only occur with the same frequency as config rotations as they can lead to similar losses in anonymity if clients still hold redemption tokens for previously active servers.

In addition, we RECOMMEND that trusted registries indicate at all times which servers are deemed to be active. If a client is asked to invoke any Privacy Pass exchange for an server that is not declared active, then the client SHOULD refuse to retrieve the server configuration during the protocol.

### 7.2.1. Allowing larger number of servers

The bounds on the numbers of servers that we proposed above are very restrictive. This is due to the fact that we considered a situation where a client could be issued (and forced to redeem) tokens for any issuing key.

An alternative system is to ensure a robust strategy for ensuring that clients only possess redemption tokens for a similarly small

number of servers at any one time. This prevents a malicious verifier from being able to invoke redemptions for many servers since the client would only be holding redemption tokens for a small set of servers. When a client is issued tokens from a new server and already has tokens from the maximum number of servers, it simply deletes the oldest set of redemption tokens in storage and then stores the newly acquired tokens.

For example, if clients ensure that they only hold redemption tokens for 4 servers, then this increases the potential size of the anonymity sets that the client belongs to. However, this doesn't protect clients completely as it would if only 4 servers were permitted across the whole system. For example, these 4 servers could be different for each client. Therefore, the selection of servers they possess tokens for is still revealing. Understanding this trade-off is important in deciding the effective anonymity of each client in the system.

### 7.3. Partitioning of server key material

If there are multiple key registries, or if a key registry colludes with an server, then it is possible to provide a split-view of an server's key material to different clients. This would involve posting different key material in different locations, or actively modifying the key material at a given location.

Key registries should operate independently of server's in the ecosystem, and within the guidelines stated in Section 4. Any client should follow the recommendations in Section 6 for determining whether an server and its key material should be trusted.

### 7.4. Additional token metadata

In [draft-davidson-pp-protocol], it is permissible to add public and private metadata bits to redemption tokens. The core protocol instantiation that is described does not include additional metadata. However, future instantiations may use this functionality to provide redemption verifiers with additional information about the user.

Note that any arbitrary bits of information can be used to further segment the size of the user's anonymity set. Any server that wanted to track a single user could add a single metadata bit to user tokens. For the tracked user it would set the bit to "1", and "0" otherwise. Adding additional bits provides an exponential increase in tracking granularity similarly to introducing more servers (though with more potential targeting).

For this reason, the amount of metadata used by an server in creating redemption tokens must be taken into account - together with the bits of information that server's may learn about clients otherwise. We discuss this more in Section 9.

#### 7.5. Tracking and identity leakage

Privacy losses may be encountered if too many redemptions are allowed in a short burst. For instance, in the Internet setting, this may allow delegated or asynchronous verifiers to learn more information from the metadata that the client may hold (such as first-party cookies for other domains). Mitigations for this issue are similar to those proposed in Section 7.2 for tackling the problem of having large number of servers.

In AV, cached SRRs and their associated server public keys have a similar tracking potential to first party cookies in the browser setting. These considerations will be covered in a separate document, detailing Privacy Pass protocol integration into the wider web architecture [draft-svaldez-pp-http-api].

#### 7.6. Client incentives for anonymity reduction

Clients may see an incentive in accepting all tokens that are issued by a server, even if the tokens fail later verification checks. This is because tokens effectively represent a form of currency that they can later redeem for some sort of benefit. The verification checks that are put in place are there to ensure that the client does not sacrifice their anonymity. However, a client may judge the "monetary" benefit of owning tokens to be greater than their own privacy.

Firstly, a client behaving in this way would not be compliant with the protocol, as laid out in [draft-davidson-pp-protocol].

Secondly, acting in this way only affects the privacy of the immediate client. There is an exception if a large number of clients colluded to accept bad data, then any client that didn't accept would be part of a smaller anonymity set. However, such a situation would be identical to the situation where the total number of clients in the ecosystem is small. Therefore, the reduction in the size of the anonymity set would be equivalent; see Section 7.2 for more details.

### 8. Security considerations

We present a number of security considerations that prevent malicious clients from abusing the protocol.

### 8.1. Double-spend protection

All issuing server should implement a robust 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. But all servers must perform global double-spend checks to avoid clients from exploiting the possibility of spending tokens more than once against distributed token checking systems. For the same reason, the global data storage must have quick update times. While an update is occurring it may be possible for a malicious client to spend a token more than once.

### 8.2. Token exhaustion

When a client holds tokens for an server, it is possible for any verifier to invoke that client to redeem tokens for that server. This can lead to an attack where a malicious verifier can force a client to spend all of their tokens for a given server. To prevent this from happening, methods should be put into place to prevent many tokens from being redeemed at once.

For example, it may be possible to cache a redemption for the entity that is invoking a token redemption. If the verifier requests more tokens then the client simply returns the cached token that it returned previously. This could also be handled by simply not redeeming any tokens for verification if a redemption had already occurred in a given time window.

In AV, the client instead caches the SRR that it received in the asynchronous redemption exchange with the server. If the same verifier attempts another redemption request, then the client simply returns the cached SRR. The SRRs can be revoked by the server, if need be, by providing an expiry date or by signaling that records from a particular window need to be refreshed.

### 8.3. Avoiding server centralization

[[OPEN ISSUE: explain potential and mitigations for server centralization]]

## 9. Protocol parametrization

We provide a summary of the parameters that we use in the Privacy Pass protocol ecosystem. These parameters are informed by both privacy and security considerations that are highlighted in Section 7 and Section 8, respectively. These parameters are intended as a single reference point for those implementing the protocol.

Firstly, let  $U$  be the total number of users,  $I$  be the total number of servers. We let  $M$  be the total number of metadata bits that are allowed to be added by any given server. Assuming that each user accept tokens from a uniform sampling of all the possible servers, as a worst-case analysis, this segregates users into a total of  $2^I$  buckets. As such, we see an exponential reduction in the size of the anonymity set for any given user. This allows us to specify the privacy constraints of the protocol below, relative to the setting of  $A$ .

parameter	value
Minimum anonymity set size ( $A$ )	5000
Recommended key lifetime ( $L$ )	2 - 24 weeks
Recommended key rotation frequency ( $F$ )	$L/2$
Maximum additional metadata bits ( $M$ )	1
Maximum allowed servers ( $I$ )	$(\log_2(U/A) - 1) / 2$
Maximum active issuance keys	1
Maximum active redemption keys	2
Minimum cryptographic security parameter	128 bits

Table 1

### 9.1. Justification

We make the following assumptions in these parameter choices.

- \* Inferring the identity of a user in a 5000-strong anonymity set is difficult.
- \* After 2 weeks, all clients in a system will have rotated to the new key.

In terms of additional metadata, the only concrete applications of Privacy Pass that use additional metadata require just a single bit. Therefore, we set the ceiling of permitted metadata to 1 bit for now, this may be revisited in future revisions.

The maximum choice of  $I$  is based on the equation  $1/2 * U/2^{(2I)} = A$ . This is derived from the fact that permitting  $I$  servers lead to  $2^I$  segregations of the total user-base  $U$ . Moreover, if we permit  $M = 1$ , then this effectively halves the anonymity set for each server, and thus we incur a factor of  $2I$  in the exponent. By reducing  $I$ , we limit the possibility of performing the attacks mentioned in Section 7.

We must also account for each user holding issued data for more than one possible active keys. While this may also be a vector for monitoring the access patterns of clients, it is likely to unavoidable that clients hold valid issuance data for the previous key epoch. This also means that the server can continue to verify redemption data for a previously used key. This makes the rotation period much smoother for clients.

For privacy reasons, it is recommended that key epochs are chosen that limit clients to holding issuance data for a maximum of two keys. By choosing  $F = L/2$  then the minimum value of  $F$  is a week, since the minimum recommended value of  $L$  is 2 weeks. Therefore, by the initial assumption, then all users should only have access to only two keys at any given time. This reduces the anonymity set by another half at most.

Finally, the minimum security parameter size is related to the cryptographic security offered by the protocol that is run. This parameter corresponds to the number of operations that any adversary has in breaking one of the security guarantees in the Privacy Pass protocol [draft-davidson-pp-protocol].

## 9.2. Example parameterization

Using the specification above, we can give some example parameterizations. For example, the current Privacy Pass browser extension [PPEXT] has nearly 300000 active users (from Chrome and Firefox). As a result,  $\log_2(U/A)$  is approximately 6 and so the maximum value of  $I$  should be 3.

If the value of  $U$  is much bigger (e.g. 5 million) then this would permit  $I = (\log_2(5000000/5000)-1)/2 \approx 4$  servers.

### 9.3. Allowing more servers

Using the recommendations in Section 7.2.1, it is possible to tolerate larger number of servers if clients in the ecosystem ensure that they only store tokens for a small number of them. In particular, if clients limit their storage of redemption tokens to the bound implied by  $I$ , then prevents a malicious verifier from triggering redemptions for all servers in the ecosystem.

## 10. Extension integration policy

The Privacy Pass protocol and ecosystem are both intended to be receptive to extensions that expand the current set of functionality. As specified in [draft-davidson-pp-protocol], all extensions to the Privacy Pass protocol SHOULD be specified as separate documents that modify the content of this document in some way. We provide guidance on the type of modifications that are possible in the following.

Any such extension should also come with a detailed analysis of the privacy impacts of the extension, why these impacts are justified, and guidelines on changes to the parametrization in Section 9. Similarly, extensions MAY also add new server running modes, if applicable, to those that are documented in Section 5.

Any extension to the Privacy Pass protocol must adhere to the guidelines specified in Section 4 for managing server public key data.

## 11. Existing applications

The following is a non-exhaustive list of applications that currently make use of the Privacy Pass protocol, or some variant of the underlying functionality.

### 11.1. Cloudflare challenge pages

Cloudflare uses an implementation of the Privacy Pass protocol for allowing clients that have previously interacted with their Internet challenge protection system to bypass future challenges [PPSRV]. These challenges can be expensive for clients, and there have been cases where bugs in the implementations can severely degrade client accessibility.

Clients must install a browser extension [PPEXT] that acts as the Privacy Pass client in an exchange with Cloudflare's Privacy Pass server, when an initial challenge solution is provided. The client extension stores the issued tokens and presents a valid redemption token when it sees future Cloudflare challenges. If the redemption

token is verified by the server, the client passes through the security mechanism without completing a challenge.

#### 11.2. Trust Token API

The Trust Token API [TrustTokenAPI] has been devised as a generic API for providing Privacy Pass functionality in the browser setting. The API is intended to be implemented directly into browsers so that server's can directly trigger the Privacy Pass workflow.

#### 11.3. Zero-knowledge Access Passes

The PrivateStorage API developed by Least Authority is a solution for uploading and storing end-to-end encrypted data in the cloud. A recent addition to the API [PrivateStorage] allows clients to generate Zero-knowledge Access Passes (ZKAPs) that the client can use to show that it has paid for the storage space that it is using. The ZKAP protocol is based heavily on the Privacy Pass redemption mechanism. The client receives ZKAPs when it pays for storage space, and redeems the passes when it interacts with the PrivateStorage API.

#### 11.4. Basic Attention Tokens

The browser Brave uses Basic Attention Tokens (BATs) to provide the basis for an anonymity-preserving rewards scheme [Brave]. The BATs are essentially Privacy Pass redemption tokens that are provided by a central Brave server when a client performs some action that triggers a reward event (such as watching an advertisement). When the client amasses BATs, it can redeem them with the Brave central server for rewards.

#### 11.5. Token Based Services

Similarly to BATs, a more generic approach for providing anonymous peers to purchase resources from anonymous servers has been proposed [OpenPrivacy]. The protocol is based on a variant of Privacy Pass and is intended to allow clients purchase (or pre-purchase) services such as message hosting, by using Privacy Pass redemption tokens as a form of currency. This is also similar to how ZKAPs are used.

### 12. References

#### 12.1. Normative References

[draft-davidson-pp-protocol]

Davidson, A., "Privacy Pass: The Protocol", n.d.,  
<<https://tools.ietf.org/html/draft-davidson-pp-protocol-00>>.

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

## 12.2. Informative References

- [Brave] "Brave Rewards", n.d., <<https://brave.com/brave-rewards/>>.
- [OpenPrivacy]  
"Token Based Services - Differences from PrivacyPass", n.d., <<https://openprivacy.ca/assets/towards-anonymous-prepaid-services.pdf>>.
- [PPEXT] "Privacy Pass Browser Extension", n.d., <<https://github.com/privacypass/challenge-bypass-extension>>.
- [PPSRV] Sullivan, N., "Cloudflare Supports Privacy Pass", n.d., <<https://blog.cloudflare.com/cloudflare-supports-privacy-pass/>>.
- [PrivateStorage]  
Steininger, L., "The Path from S4 to PrivateStorage", n.d., <<https://medium.com/least-authority/the-path-from-s4-to-privatestorage-ae9d4a10b2ae>>.
- [TrustTokenAPI]  
Google, ., "Getting started with Trust Tokens", n.d., <<https://web.dev/trust-tokens/>>.

## Appendix A. Contributors

- \* Alex Davidson (alex.davidson92@gmail.com)
- \* Christopher Wood (caw@heapingbits.net)

## Author's Address

Alex Davidson  
Cloudflare Portugal  
Largo Rafael Bordalo Pinheiro 29  
Lisbon  
Portugal

Email: alex.davidson92@gmail.com

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;">&gt;</div> </div> </div>	
serverResp = Issue(pkS, skS, req)	
<div style="display: flex; align-items: center; justify-content: center;"> <div style="text-align: right; margin-right: 10px;">issueResp</div> <div style="flex-grow: 1; border-bottom: 1px dashed black; position: relative;"> <div style="position: absolute; left: -5px; top: -5px;">&lt;</div> </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..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<sup>16</sup>-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: 14 January 2021

S. Valdez  
Google LLC  
13 July 2020

Privacy Pass: HTTP API  
draft-svaldez-pp-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 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 . . . . .	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 . . . . .	9
7. Security Considerations . . . . .	11
8. IANA Considerations . . . . .	11
8.1. Well-Known URI . . . . .	11
9. Normative References . . . . .	11
Author's Address . . . . .	12

## 1. Introduction

The Privacy Pass protocol as described in [draft-davidson-pp-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-davidson-pp-protocol] and [draft-davidson-pp-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-davidson-pp-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-davidson-pp-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.

### 6.1. Generic Token Redemption

Inputs: - "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 call the "Redeem" function with "redemption\_token" and additional data of "info" storing the resulting "data" and "tag".
2. 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: - "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: - "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-davidson-pp-architecture].

## 8. IANA Considerations

### 8.1. Well-Known URI

This specification registers a new well-known URI.

URI suffix: "privacy-pass"

Change controller: IETF.

Specification document(s): this specification

## 9. Normative References

[draft-davidson-pp-architecture]

Davidson, A., "Privacy Pass: Architectural Framework", n.d., <<https://tools.ietf.org/html/draft-davidson-pp-architecture-00>>.

[draft-davidson-pp-protocol]

Davidson, A., "Privacy Pass: The Protocol", n.d., <<https://tools.ietf.org/html/draft-davidson-pp-protocol-00>>.

[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>>.

[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>>.

[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](mailto:svaldez@chromium.org)