

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 bytearray "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