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.
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 use-cases, the Privacy Pass protocol is used in a number of practical applications. See [DGSTV18], [TrustTokenAPI], [PrivateStorage], [OpenPrivacy], and [Brave] for examples.
3.2. Anonymity and security guarantees

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

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

Section 6 elaborates on these protocol anonymity and security requirements.

3.3. Basic assumptions

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

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

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

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

4. Protocol description

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

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

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

4.2. Client setup

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

4.3. Issuance phase

The issuance phase allows the client to receive "m" anonymous authorization tokens from the server.

\[
\begin{align*}
\text{Client}(pkS, m) & \quad \text{Server}(skS, pkS) \\
\text{cInput} = \text{Generate}(m) & \\
\text{req} = \text{cInput}.req & \\
\text{serverResp} = \text{Issue}(pkS, skS, \text{req}) & \\
\text{issueResp} & \\
\text{tokens} = \text{Process}(pkS, \text{cInput}, \text{issueResp}) & \\
\text{store}[\text{server.id}].push(\text{tokens}) & 
\end{align*}
\]

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)
------------------------------------------------------------

\[ \text{token} = \text{store}[\text{Issue.id}].\text{pop()} \]
\[ \text{req} = \text{Redeem}(\text{token, info}) \]

\[
\begin{align*}
\text{if (dsIdx.includes}(\text{req.data})) & \{ \\
\quad \text{raise ERR DOUBLE_SPEND} \\
\text{\} resp = \text{Verify}(pkS, skS, req) \\
\quad \text{if (resp.success)} & \{ \\
\quad \quad \text{dsIdx.push}(\text{req.data}) \\
\text{\} } \\
\text{\} resp \\
\end{align*}
\]

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:

\[ \text{len(aux)} | | \text{aux} | | \text{len(server.id)} | | \text{server.id} | | \text{current_time()} \]

where "\text{len(x)}" is the length of "\text{x}" in bytes, and "\text{aux}" is arbitrary auxiliary data chosen by the client. The usage of "\text{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 "\text{req.data}" more than once, the server uses an index, "\text{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:
5.2.3. Process

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

Inputs:
* "pkS": An server "PublicKey".
* "input": An "IssuanceInput" struct.
* "resp": An "IssuanceResponse" struct.

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

Throws:
* "ERR_PROOF_VALIDATION" (Section 5.3)

5.2.4. Redeem

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

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

Outputs:
* "req": A "RedemptionRequest" struct.

5.2.5. Verify

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

Inputs:
* "pkS": An server "PublicKey".
* "skS": An server "PrivateKey".
* "req": A "RedemptionRequest" struct.

Outputs:
* "resp": A "RedemptionResponse" struct.

5.3. Error types

* "ERR_PROOF_VALIDATION": Error occurred when a client attempted to verify the proof that is part of the server’s response.
* "ERR_DOUBLE_SPEND": Error occurred when a client has attempted to redeem a token that has already been used for authorization.

6. Security considerations

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

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

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

6.1. Unlinkability

Formally speaking the security model is the following:

* The adversary runs the server setup and generates a keypair "(pkS, skS)".
* The adversary specifies a number "Q" of issuance phases to initiate, where each phase "i in range(Q)" consists of "m_i" Issue evaluations.
* The adversary runs "Issue" using the keypair that it generated on each of the client messages in the issuance phase.
* When the adversary wants, it stops the issuance phase, and a random number "l" is picked from "range(Q)".

* A redemption phase is initiated with a single token with index "i" randomly sampled from "range(m_1)".

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

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

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

6.2. One-more unforgeability

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

The security model roughly takes the following form:

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

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

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

* The adversary succeeds if "m_succ > m".

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

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

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

6.4. Additional token metadata

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

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

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

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

6.5. Maximum number of tokens issued

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

7. VOPRF instantiation

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

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

* OPRF(curve448, SHA-512) (ID = 0x0002);
* OPRF(P-384, SHA-512) (ID = 0x0004);
* OPRF(P-521, SHA-512) (ID = 0x0005).

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

7.2. Protocol contexts

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

7.3. Functionality

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

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

7.3.2. Issue

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

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

7.3.3. Process

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

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

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

7.3.5. Verify

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

7.4. Security justification

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

The unlinkability property follows unconditionally as the view of the adversary in the redemption phase is distributed independently of the issuance phase. The unforgeability property follows from the one-more decryption security of the ElGamal cryptosystem [DGSTV18]. In [KLOR20] it is also proven that this protocol satisfies the stronger notion of unforgeability, where the adversary is granted a verification oracle, under the chosen-target Diffie-Hellman assumption.

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

8. Protocol ciphersuites

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

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

8.1. PP(OPRF2)

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

8.2. PP(OPRF4)

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

8.3. PP(OPRF5)

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

9. Extensions framework policy

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

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

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

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

If an extension requires changing the generic protocol description as described in Section 4, then the change may have to result in changes to the draft specification here also.
Each new extension that modifies the internals of the protocol in either of the two ways MUST re-justify that the extended protocol still satisfies the security requirements in Section 6. Protocol extensions MAY put forward new security guarantees if they are applicable.

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

10. References

10.1. Normative References

[draft-davidson-pp-architecture]

[draft-svaldez-pp-http-api]

[I-D.irtf-cfrg-voprf]


10.2. Informative References


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

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 13 January 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

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

1.1. Terminology

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

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

1.2. Layout

* Section 2: Describes the wrapping of messages within HTTP requests/responses.
* Section 3: Describes how HTTP clients retrieve server configurations and key commitments.

* Section 5: Describes how issuance requests are performed via a HTTP API.

* Section 6: Describes how redemption requests are performed via a HTTP API.

1.3. Requirements

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

2. Privacy Pass HTTP API Wrapping

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

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

* "type" whose value is a String conveying the function that is being performed with this request.

* "body" whose value is a byte sequence containing a Privacy Pass protocol message.

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

3. Server key registry

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

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

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

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

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

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

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

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

The client can then perform a request for the key commitment against either the global registry or the server as described in Section 4. Note that the signature should be verified by the client to ensure that the key material is owned by the server. This requires that the client know the public verification key that is associated with the server.
To avoid user segregation as a result of server configuration/commitment rotation, the log operator SHOULD enforce limits on how many active commitments exist and how quickly the commitments are being rotated. Clients SHOULD reject configurations/commitments that violate their requirements for avoiding user segregation. These considerations are discussed as part of [draft-ietf-privacypass-architecture].

3.2. Server Configuration Retrieval

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

No outputs.

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

```c
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:

```c
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.
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:

```c
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:

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

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

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

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

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

1. When a client wants to request tokens from a server, it should first fetch a key commitment from the server via the process described in Section 4 and keep the result as "commitment".

2. The client should then call the "Generate" function requesting "count" tokens storing the resulting "input" data.

3. The client then makes a POST request to <"server_origin">/.well-known/privacy-pass with a message of type "request-issuance" and a body of:

```plaintext
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:

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

1. The client should then should call the "Process" function with the "public_key", stored "inputs" and resulting "resp", to extract a list of "redemption_tokens".

2. The client should store the "public_key" associated with these tokens and the elements of "redemption_tokens" under storage partitioned by the "server_origin", accessible only via the Privacy Pass API.
6. Privacy Pass Redemption

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

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

6.1. Generic Token Redemption

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

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

1. The client should check whether the "server_id" is present in the "context". If it isn’t and the size of the "context" is beneath the client’s limit, it should be added.

2. The client should call the "Redeem" function with "redemption_token" and additional data of "info" storing the resulting "data" and "tag".

3. The client makes a POST request to <"server_origin">/.well-known/privacy-pass with a message of type "token-redemption" and a body of:

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

1. The server, upon receipt of "request" should call the "Verify" interface with "public_key", "secret_key" and the received "data", "tag", "info" storing the resulting "resp".
2. The server should then respond to the POST request with a message of type "redemption-result" and a signed body of:

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

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

6.2. Direct Redemption

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

1. When a client wants to redeem tokens for a server, it should first fetch a key commitment from the server via the process described in Section 4 and keep the result as "commitment".

2. The client should then look up the storage partition associated with "server_origin" and fetch a "redemption_token" and "public_key".

3. The client should verify that the "public_key" is in the current "commitment". If not, it should discard the token and fail the redemption attempt.

4. As part of the request to "target", the client will include the token as part of the request in the "Sec-Privacy-Pass" header along with whatever other parameters are being passed as part of the request to "target". The header will contain a message of type "token-redemption" with a body of:

```
struct {
    opaque server_id<1..2^16-1> = server_id;
    uint16 ciphersuite = ciphersuite;
    opaque public_key<1..2^16-1> = public_key;
    RedemptionToken token<1..2^16-1> = redemption_token;
    opaque additional_data<1..2^16-1> = additional_data;
}
```
At this point, the "target" can perform a generic redemption as described in Section 6.1 by forwarding the message included in the request to "target".

6.3. Delegated Redemption

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

1. When a client wants to redeem tokens for a server, it should first fetch a key commitment from the server via the process described in Section 4 and keep the result as "commitment".

2. The client should then look up the storage partition associated with "server_origin" and fetch a "redemption_token" and "public_key".

3. The client should verify that the "public_key" is in the current "commitment". If not, it should discard the token and fail the redemption attempt.

4. The client constructs a bytestring "info" made up of the "target", the current "timestamp", and "additional_data":

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

1. The client then performs a token redemption as described in Section 6.1. Storing the resulting "redemption-result" message.

2. As part of the request to "target", the client will include the redemption result as part of the request in the "Sec-Privacy-Pass" header along with whatever other parameters are being passed as part of the request to "target". The header will contain a message of type "signed-redemption-result" with a body of:
struct {
    opaque server_origin<1..2^16-1>;
    opaque target<1..2^16-1>;
    uint64 timestamp;
    opaque additional_data<1..2^16-1> = additional_data;
    opaque signed_redemption<1..2^16-1>;
}

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

7. Security Considerations

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

8. Privacy considerations

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

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

9. IANA Considerations

9.1. Well-Known URI

This specification registers a new well-known URI.

URI suffix: "privacy-pass"

Change controller: IETF.

Specification document(s): this specification

10. Normative References


Author’s Address

Steven Valdez
Google LLC

Email: svaldez@chromium.org
Privacy Pass Issuance Protocol
draft-ietf-privacypass-protocol-06

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 7 January 2023.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document.
1. Introduction

The Privacy Pass protocol provides a privacy-preserving authorization mechanism. In essence, the protocol allows clients to provide cryptographic tokens that prove nothing other than that they have been created by a given server in the past [I-D.ietf-privacypass-architecture].
This document describes the issuance protocol for Privacy Pass. It specifies two variants: one that is privately verifiable based on the oblivious pseudorandom function from [OPRF], and one that is publicly verifiable based on the blind RSA signature scheme [BLINDRSA].

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

2. Terminology

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

The following terms are used throughout this document.

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

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

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

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

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

3. Configuration

Issuers MUST provide two parameters for configuration:

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

2. Issuer Public Key values: an Issuer Public Key for an issuance protocol.
The Issuer parameters can be obtained from an Issuer via a directory object, which is a JSON object whose values are other JSON objects and URLs for the parameters.

| Field Name         | Value                        |
|--------------------+------------------------------|
| issuer-request-uri | Issuer Request URI resource URL as a JSON string |
| token-keys         | List of Issuer Public Key values, each as JSON objects |

Table 1

Each "token-keys" JSON object contains the following fields and corresponding raw values.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>token-type</td>
<td>Integer value of the Token Type, as defined in Section 8.1, as a JSON number</td>
</tr>
<tr>
<td>token-key</td>
<td>The base64url encoding of the public key for use with the issuance protocol, including padding, as a JSON string</td>
</tr>
</tbody>
</table>

Table 2

Issuers MAY advertise multiple token-keys for the same token-type to support key rotation. In this case, Issuers indicate preference for which token key to use based on the order of keys in the list, with preference given to keys earlier in the list.

Altogether, the Issuer’s JSON directory could look like:
4. Token Challenge Requirements

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

5. Issuance Protocol for Privately Verifiable Tokens

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

```
Origin    Client   Issuer
(pki)     (skI, pkI)
                       +------------------------------------\                      
Challenge  ----> TokenRequest          (evaluate)                      |
Token      <----+          <------------ TokenResponse                     
                       \-------------------------------------/                   
```

Issuers provide a Private and Public Key, denoted skI and pkI, respectively, used to produce tokens as input to the protocol. See Section 5.5 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

* Issuer name, identifying the Issuer. This is typically a host name that can be used to construct HTTP requests to the Issuer.
* Issuer Public Key pkI, with a key identifier key_id computed as described in Section 5.5.

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

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

5.1. Client-to-Issuer Request

The Client first creates a context as follows:

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

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

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

```c
nonce = random(32)
context = SHA256(challenge)
token_input = concat(0x0001, nonce, context, key_id)
blind, blinded_element = client_context.Blind(token_input)
```

The Blind function is defined in [OPRF], Section 3.3.2. If the Blind function fails, the Client aborts the protocol. Otherwise, the Client then creates a TokenRequest structured as follows:

```c
struct {
    uint16_t token_type = 0x0001;
    uint8_t token_key_id;
    uint8_t blinded_msg[Ne];
} TokenRequest;
```

The structure fields are defined as follows:

* "token_type" is a 2-octet integer, which matches the type in the challenge.
* "token_key_id" is the least significant byte of the key_id in network byte order (in other words, the last 8 bits of key_id).

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

The values token_input and blinded_element are stored locally and used later as described in Section 5.3. The Client then generates an HTTP POST request to send to the Issuer, with the TokenRequest as the body. The media type for this request is "message/token-request". An example request is shown below.

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

<Bytes containing the TokenRequest>
```

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

* The TokenRequest contains a supported token_type.

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

* The TokenRequest.blinded_request is of the correct size.

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

### 5.2. Issuer-to-Client Response

Upon receipt of a TokenRequest, the Issuer tries to deserialize TokenRequest.blinded_msg using DeserializeElement from Section 2.1 of [OPRF], yielding blinded_element. If this fails, the Issuer MUST return an HTTP 400 error to the client. Otherwise, if the Issuer is willing to produce a token token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:

```plaintext
server_context = SetupVOPRFServer(0x0004, skI, pkI)
evaluate_element, proof = server_context.Evaluate(skI, blinded_element)
```
SetupVOPRFServer is in [OPRF], Section 3.2 and Evaluate is defined in [OPRF], Section 3.3.2. The Issuer then creates a TokenResponse structured as follows:

```c
struct {
    uint8_t evaluate_msg[Nk];
    uint8_t evaluate_proof[Ns+Ns];
} TokenResponse;
```

The structure fields are defined as follows:

* "evaluate_msg" is the N\text{-}octet evaluated message, computed as SerializeElement(evaluate_element).

* "evaluate_proof" is the (Ns+Ns)\text{-}octet serialized proof, which is a pair of Scalar values, computed as concat(SerializeScalar(proof[0]), SerializeScalar(proof[1])), where Ns is as defined in [OPRF], Section 4.

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

:status = 200  
content-type = message/token-response  
content-length = <Length of TokenResponse>

<Bytes containing the TokenResponse>

5.3. Finalization

Upon receipt, the Client handles the response and, if successful, deserializes the body values TokenResponse.evaluate_response and TokenResponse.evaluate_proof, yielding evaluated_element and proof. If deserialization of either value fails, the Client aborts the protocol. Otherwise, the Client processes the response as follows:

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

The Finalize function is defined in [OPRF], Section 3.3.2. If this succeeds, the Client then constructs a Token as follows:
struct {
    uint16_t token_type = 0x0001
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;

Otherwise, the Client aborts the protocol.

5.4. Token Verification

To verify a token, a verifier creates a VOPRF context, evaluates the token contents, and compares the result against the token authenticator value, as follows:

server_context = SetupVOPRFServer(0x0004, skI, pkI)
token_authenticator_input =
    concat(Token.token_type,
           Token.nonce,
           Token.challenge_digest,
           Token.token_key_id)
token_authenticator = server_context.Evaluate(token_authenticator_input)
valid = (token_authenticator == Token.authenticator)

5.5. Issuer Configuration

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

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

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

key_id = SHA256(concat(0x0001, SerializeElement(pkI)))

6. Issuance Protocol for Publicly Verifiable Tokens

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

1. The output tokens are publicly verifiable by anyone with the Issuer public key.
2. The issuance protocol does not admit public or private metadata to bind additional context to tokens.

The first property means that any Origin can select a given Issuer to produce tokens, as long as the Origin has the Issuer public key, without explicit coordination or permission from the Issuer. This is because the Issuer does not learn the Origin that requested the token during the issuance protocol.

Beyond these differences, the publicly verifiable issuance protocol variant is nearly identical to the privately verifiable issuance protocol variant. In particular, Issuers provide a Private and Public Key, denoted $sk_I$ and $pk_I$, respectively, used to produce tokens as input to the protocol. See Section 6.5 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

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

* Issuer Public Key $pk_I$, with a key identifier $key_id$ computed as described in Section 6.5.

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

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

6.1. Client-to-Issuer Request

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

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

The $rsabssa_blind$ function is defined in [BLINDRSA], Section 5.1.1.

The Client then creates a TokenRequest structured as follows:
struct {
    uint16_t token_type = 0x0002
    uint8_t token_key_id;
    uint8_t blinded_msg[Nk];
} TokenRequest;

The structure fields are defined as follows:

* "token_type" is a 2-octet integer, which matches the type in the challenge.

* "token_key_id" is the least significant byte of the key_id in network byte order (in other words, the last 8 bits of key_id).

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

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

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

<Bytes containing the TokenRequest>

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

* The TokenRequest contains a supported token_type.

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

* The TokenRequest.blinded_msg is of the correct size.

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

If the Issuer is willing to produce a token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:

\[
\text{blind}\_\text{sig} = \text{rsabssa}\_\text{blind}\_\text{sign}(skI, \text{TokenRequest.blinded}\_\text{rmsg})
\]

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

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

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

```plaintext
:status = 200
content-type = message/token-response
content-length = <Length of TokenResponse>
<Bytes containing the TokenResponse>
```

6.3. Finalization

Upon receipt, the Client handles the response and, if successful, processes the body as follows:

\[
\text{authenticator} = \text{rsabssa}\_\text{finalize}(pkI, \text{nonce}, \text{blind}\_\text{sig}, \text{blind}\_\text{inv})
\]

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

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

Otherwise, the Client aborts the protocol.
6.4. Token Verification

To verify a token, a verifier checks that Token.authenticator is a valid signature over the remainder of the token input as described below. The function RSA-Verify(msg, pk, sig) is a procedure to verify a signature sig over message msg using the public key pk. Its implementation is not specified in this document.

token_authenticator_input = concat(Token.token_type, Token.nonce, Token.challenge_digest, Token.token_key_id)
valid = RSA-Verify(token_authenticator_input, pkI, Token.authenticator)

6.5. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted skI and pkI, respectively, used to produce tokens. Each key pair SHALL be generated as as specified in FIPS 186-4 [DSS].

The key identifier for a keypair (skI, pkI), denoted key_id, is computed as SHA256(encoded_key), where encoded_key is a DER-encoded SubjectPublicKeyInfo (SPKI) object carrying pkI. The SPKI object MUST use the RSASSA-PSS OID [RFC5756], which specifies the hash algorithm and salt size. The salt size MUST match the output size of the hash function associated with the public key and token type.

7. Security considerations

This document outlines how to instantiate the Issuance protocol based on the VOPRF defined in [OPRF] and blind RSA protocol defined in [BLINDRSA]. All security considerations described in the VOPRF document also apply in the Privacy Pass use-case. Considerations related to broader privacy and security concerns in a multi-Client and multi-Issuer setting are deferred to the Architecture document [I-D.ietf-privacypass-architecture].

8. IANA considerations

8.1. Token Type

This document updates the "Token Type" Registry with the following values.
Table 3: Token Types

8.2. Media Types

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

* TokenRequest: "message/token-request"
* TokenResponse: "message/token-response"

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

8.2.1. "message/token-request" media type

Type name: message

Subtype name: token-request

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 7

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A
8.2.2. "message/token-response" media type

Type name: message
Subtype name: access-token-response
Required parameters: N/A
Optional parameters: None
Encoding considerations: only "8bit" or "binary" is permitted
Security considerations: see Section 7
Interoperability considerations: N/A
Published specification: this specification
Applications that use this media type: N/A
Fragment identifier considerations: N/A
Additional information: Magic number(s): N/A
Deprecated alias names for this type: N/A
File extension(s): N/A
Macintosh file type code(s): N/A
Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

9. References

9.1. Normative References


9.2. Informative References


Appendix A. Acknowledgements

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

Appendix B. Test Vectors

This section includes test vectors for the two basic issuance protocols specified in this document. Appendix B.1 contains test vectors for token issuance protocol 1 (0x0001), and Appendix B.2 contains test vectors for token issuance protocol 2 (0x0002).

B.1. Issuance Protocol 1 - VOPRF(P-384, SHA-384)

The test vector below lists the following values:

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

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

* challenge: A random challenge digest, represented as a hexadecimal string.
nonce: The 32-byte client nonce generated according to Section 5.1, represented as a hexadecimal string.

blind: The blind used when computing the OPRF blinded message, serialized using SerializeScalar from Section 2.1 of [OPRF] and represented as a hexadecimal string.

token_request: The TokenRequest message constructed according to Section 5.1, represented as a hexadecimal string.

token_request: The TokenResponse message constructed according to Section 5.2, represented as a hexadecimal string.

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

skS: 0177781aeced893dccdf80713d318a801e2a0498240fdec650304bbbf0d08d3b5c0cf6cfee457aaa983ee02ff2f283b7a9
pkS: 022c63f79ac59c0ba32d04245f676a2133bd6120c90d67afa05cd6f8614294b7366c2526458300551b79a4911c2590a36
challenge: a5d46383359ef34e3c4a7b8d1b3165778bffc9b70c9e6a60dd14143e4c9c9fbd
nonce: 5d4799f8338ddc50a6685f83b8edc264b2f157015229d12b3384c0f199efe7b8blind: 0322fec505230992256296063d989b59cc03e83184eb6187076d264137622d20248e4e525bdc007b8d1560e0a6f49d9
token_request: 00011a02861fd50d14be873611cfff0131d2c872c79d0260c6763498a2a3f14ca926009c0f247653406e1d52b68d61b7ed2bac9ea
token_response: 038e3625b6a769668a99680e46cf9479f5dc1e86d57164ab3b4a569ddfc4b6bf1c49719f5a5194f1d0518de8444968421ba36e8144aa7902705ff0f3cf045863d69451a27ba210cc45760c2f1a6045134d877b39e8bcbbf920e5de4a3372557debf211765cd369976860b0c39f9082d6a3e03f9e891246240173d2cf3d69a4613b0f841597902922e74c7af2e4639e4
token: 00015d4799f8338d5ca6685f83b8edc264b2f157015229d12b3384c0f199efe7bb7842cfdbe0ed756a680868ef109a280a393e001d2fa56b1be46ecb31fa25e76731a5bd698ea7ab843b887a7ed9b2ffafa704757a43a8fc687939424b29a7554b40fde130ab7a822715909cb73f99a45b640ca1c85180ba9ca1a40bab8b664406a34bcbc63b5e2e5c455cea00001a968f7

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

The test vector below lists the following values:

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

* pkS: The DER-encoded SubjectPublicKeyInfo object carrying the public key corresponding to skS, as described in Section 6.5, represented as a hexadecimal string.
* challenge: A random challenge digest, represented as a hexadecimal string.

* nonce: The 32-byte client nonce generated according to Section 6.1, represented as a hexadecimal string.

* blind: The blind used when computing the blind RSA blinded message, represented as a hexadecimal string.

* salt: The randomly generated 48-byte salt used when encoding the blinded token request message, represented as a hexadecimal string.

* token_request: The TokenRequest message constructed according to Section 6.1, represented as a hexadecimal string.

* token_request: The TokenResponse message constructed according to Section 6.2, represented as a hexadecimal string.

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

skS: 2d2d2d2d2d424547494e20505249566144545204b45592d2d2d2d0a4d49494576514942414441ae42676b716866b6947397730424151456143343246b6776676536a416
74541416c49424151444c4775317261705831736334420a4f67a38717957355379356b6
f6a4130354355b66717444774e38366a424b5a4f76457245526b49314c527876734d646
3327961323333616b4745714c756b440a556a35743561496b3172417643655844644e445
03442325055708513436e696396e6b4926b677257697444494841386139739137586
e6c5079596f784f530a466e5658563835464f314a752b623973336356d586d3451a7513
9455961497138371724450567a50335758712b524e4d636379323269686763624c766d4
2390a6a41353534475666325a6c74785954736f4c346872377a586964e394637466216
5676f75396765b524d846453522b4a3956595a634a73a624c756570480a544f72535
a4d494850b53581d4166414f544a547426dd4430683566672f43473475676a79486
4e51383733414e4b5a55716d3676574574413872514c620a4530742b496c70666174d4
241145367475414c7a4362647a69316a506453584d6b52b4346679666653232b726
648ee726672466502f566344787275690a320731615358a5969625634552b462f24
6d56546c48506714c773178513457657263636443736866cc78ac575536587734
2737663386f364750320a635936ef77704244776362618474b55b530456b62393503
84ca57634753473561556e48a58523769e7834635ac666f4e6e74255165366855787
34d710a623064487864484424d644765677764bf6a4f6a70532f39386d45559735
6422f3661326c7265676766a632f326e4b434b74593737437645716c47460a78a414
261577538364d435a342f513133c762b42656662717449397315a5776a7264556851
835643787293251564d4575169e57684174364d7154340a53425354726f6c5a7a777
2716a65384d50a393175614ed4d458474c63484c4932637358a76374b5314b4267514
47667773505555764139a5a325a58958350a6d49784d5424e445467a56625550754b
b41319759631554d44e63556a71682b7a652f376b33794678b683051463316271365
0654c93047495369414f0a354b4f574d39454b6f2b7841513226214b314d664f5931472
b386a7a4258557042733935e3b53535387395864d366e79647676730424a385a63565

challenge: 3f5af1c30d1f86022458ace836df8af325378054370fe8a3d77e2e6d74d810d
nonce: c0fcbbb243d8f5d4f661dbdefca95879b39aeccb77b7db731b59c09688773125
blind: 0002ca832fffabdd44e2cd54e5e24d74519d297608aec9ab88e26b732adcb382781e7e2657c8b947519bfa6b2ed02ecc383f8cd0ae9627db52a7f7eb7ea168b1e4f635637caca49f8990d5359f8a7dcac1ba58fb6854d32a67621d368cc11291757d4f8ee5
salt: 4daf07bc96a829736ce6386df8af325378054370fe8a3d77e2e6d74d810d

Celi, et al. Expires 7 January 2023
token_response: 6e7d5334765b84ea43b81ae8f41334fdac47b3d3faae52b3c99f42a67d823992ac4fa129a938e139bf052d85804bdaa907f754fda346e6eaae0ccc15a500fb2987b534d0558d832df68b3533f6cb953dabcfffe2f66af336c1128f607f079f1906a2fed919691340e751a8e2173e67456f97e4beb7ad0ee5c65ce82ad477d3e44b3755b1c0f168ab85ece6623f87c5634b036382d6ad4ab870ab975e8ffdd0b95bcf457dc83337ffe85bbc777d4ae5cb4bdc5aeecfc9588cc822cc53ded3da699af86bfa0d3504fe49da8eb5502e444a3bd4c38f9e3cbadd50cba56b4f3f0718a6e5e7d8d6c40762c9b962edc731f6a7ec8641cbf98a0e9c9dfe8bf7f6

token: 00020fcbb243d8f5d4f661dbde29a5879b39aeeccc777b7db731b59c09688773125ad76ab53adcc2a424e62ae3d71b9bf3fc9332122faef07cbb709d0e4da86ca6f7ca572f8982a9ca248a3056186322d93ca147266121ddeb5632c07f1f71cd27080df813634e5d4d16d6f3e80366e56edc1de4ba0d7aed2675c15156d774b311778091bf5f2ee992615f2289459a41c5739dec6d42447744fe07c53c9d090f53263d019255dfc727379132bd6821ad49f1a98db6873319d040c4073d74a8fe1d0806b2a25b4d26246c5bb2f9f277463b0315258906389df894946d82f3b92be773a9f6eb6c1fed9c9f2b6dfbae1ff369f20d0267cdd20f3cbca30f8b0c0e9d9a1a39aa01560614030d5099aa36f085347681ae50f2f3d08b136cd7f97e1a14df1ca19694320c24ccbc7c5d90aecd915af3ac11a3baf56d38c8213e39f6731fa5e701697d0bfbfccfc83b447945b351115a207703702265b2a19df939f308e

Authors’ Addresses

Sofía Celi
Brave Software
Lisbon, Portugal
Email: cherenkov@riseup.net

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

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

Celi, et al. Expires 7 January 2023
Steven Valdez
Google LLC
Email: svaldez@chromium.org

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