Network Working Group
Internet-Draft
Intended status: Informational
Expires: 10 September 2020

A. Davidson
Cloudflare Portugal
9 March 2020

Privacy Pass: The Protocol
draft-davidson-pp-protocol-00

Abstract

This document specifies the Privacy Pass protocol for privacy-preserving authorization of clients to servers. The privacy requirement is that 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 Internet setting.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

A common problem on the internet is providing an effective mechanism for servers to derive trust from the clients that it interacts with, without hampering the accessibility of honest clients. Typically, this can be done by providing some sort of authorization challenge to the client. A client providing a correct solution to the challenge can be provided with a cookie. This cookie can be presented the next time it interacts with the server. The resurfacing of this cookie allows the server to see that the client passed the authorization check in the past. Consequently, the server can re-authorize the client again immediately, without the need for the client to complete a new challenge.

In scenarios where clients need to identify themselves, the authorization challenge usually take the form of some sort of login procedure. Otherwise, the server may just want to verify that the client demonstrates some particular facet of behavior (such as being human). Such cases may only require a lightweight form of challenge (such as completing a CAPTCHA).

In both cases, if a server issues cookies on successful completion of challenges, then the client can use this cookie to bypass future challenges for the lifetime of the cookie. The downside of this approach is that it provides the server with the ability to link all of the client's interactions that it witnesses. In these situations, the client's effective privacy is dramatically reduced.

The Privacy Pass protocol was initially introduced as a mechanism for authorizing clients that had already been authorized in the past, without compromising their privacy [DGSTV18]. The protocol works by providing client's with privacy-preserving re-authentication tokens for a particular server. The tokens are "privacy-preserving" in the sense that they cannot be linked back to the previous session where they were issued.

The Internet performance company Cloudflare has already implemented server-side support for an initial version of the Privacy Pass

protocol [PPSRV], and client-side implementations also exist [PPEXT]. More recently, a number of applications have been built upon the protocol, or slight variants of it; see: [TRUST], [OpenPrivacy], [PrivateStorage]. The protocol can be instantiated using a cryptographic primitive known as a verifiable oblivious pseudorandom function (VOPRF) for implementing the authorization mechanism. Such VOPRF protocols can be implemented already in prime-order groups, and constructions are currently being drafted in separate standardization processes [I-D.irtf-cfrg-voprf].

The Privacy Pass protocol is split into three stages. The first stage, initialisation, produces the global server configuration that is broadcast to (and stored by) all clients. The "issuance" phase provides the client with unlinkable tokens that can be used to initiate re-authorization with the server in the future. The redemption phase allows the client to redeem a given re-authorization token with the server that it interacted with during the issuance phase. In addition, the protocol must satisfy two cryptographic security requirements known as "unlinkability" and "unforgeability".

This document will lay out the generic description of the protocol, along with a secure implementation based on the VOPRF primitive. It will also describe the structure of protocol messages, and the framework for characterizing possible extensions to the protocol description.

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].

1.1. Layout

* Section 2: Describes the terminology and assumptions adopted throughout this document.

* Section 3: Describes the internal functions and data structures that are used by the Privacy Pass protocol.

* Section 4: Describes the generic protocol structure, based on the API provided in Section 3.

* Section 5: Describes the security requirements of the generic protocol description.

* Section 6: Describes an instantiation of the API in Section 3 based on the VOPRF protocol described in [I-D.irtf-cfrg-voprf].

* Section 7: Describes ciphersuites for use with the Privacy Pass protocol based on the instantiation in Section 6.

* Section 8: Describes the policy for implementing extensions to the Privacy Pass protocol.

2. Preliminaries

2.1. 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 documented here (typically denoted S).

* Client: An entity that seeks authorization from a server that supports interactions in the Privacy Pass protocol (typically denoted C).

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

* Commitment: Alternative name for Server's public key corresponding to the secret key that they hold.

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.

2.2. Basic assumptions

We make only a few minimal assumptions about the environment that the clients and servers that support 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.

* We assume that the client has access to a global directory of the current configurations used by all Privacy Pass servers.

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

3. Privacy Pass functional API

Before describing the protocol itself in Section 4, we describe the underlying functions that are used in substantiating the protocol itself. Instantiating this set of functions, along with meeting the security requirements highlighted in Section 5, provides an instantiation of the wider protocol.

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

3.1. Data structures

The following data structures are used throughout the Privacy Pass protocol and 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].

3.1.1. Ciphersuite

The "Ciphersuite" enum describes the ciphersuite that is used for instantiating the Privacy Pass protocol. The values that we provide here are described further in Section 7.

```
enum {
  p384_hkdf_sha512_sswu_ro(0)
  p521_hkdf_sha512_sswu_ro(1)
  curve448_hkdf_sha512_ell2_ro(2)
  (255)
} Ciphersuite;
```

3.1.2. ServerConfig

The "ServerConfig" struct describes and maintains the underlying configuration that is used by the server.

```
struct {
  opaque id<0..2^16-1>
  Ciphersuite ciphersuite;
  SecretKey key<1..2^32-1>;
  PublicKey pub_key<1..2^32-1>;
  opaque max_evals<0..255>;
} ServerConfig;
```

The "SecretKey" and "PublicKey" types are just wrappers around byte arrays.

```
opaque SecretKey<1..2^32-1>;
opaque PublicKey<1..2^32-1>;
```

3.1.3. ServerUpdate

The "ServerUpdate" struct contains the public information related to the creation of a new "ServerConfig" message. This is sent either directly to clients, or indirectly via an update process.

```
struct {
  opaque id<0..2^16-1>
  Ciphersuite ciphersuite;
  PublicKey pub_key<1..2^32-1>;
  opaque max_evals<0..255>;
} ServerUpdate;
```

3.1.4. ClientConfig

The "ClientConfig" struct describes and maintains the underlying configuration that is used by the client.

```
struct {
  ServerUpdate s;
} ClientConfig;
```

3.1.5. ClientIssuanceInput

The "ClientIssuanceInput" struct describes the data that is generated by the client, that is necessary in sending to and processing issuance data received from the server.

```
struct {
  ClientIssuanceProcessing client_data;
  ClientIssuanceElement msg_data;
} ClientIssuanceInput;
```

The struct contains two internal structs, described below.

```
struct {
  opaque client_data<1..2^32-1>;
  opaque gen_data<1..2^32-1>;
} ClientIssuanceProcessing;
```

```
   struct {
     opaque issue_data<1..2^32-1>;
   } ClientIssuanceElement;
```

3.1.6. IssuanceMessage

The "IssuanceMessage" struct corresponds to the message that the client sends to the server during the issuance phase of the protocol (Section 4.2).

```
   struct {
     ClientIssuanceElement issue_element<1..n>
   } IssuanceMessage;
```

In the above, "issue_element" is a vector of length "n", where "n" is some value that must satisfy "n =< m" for "m = max_evals" that is specified in the "ServerConfig".

3.1.7. IssuanceResponse

The "IssuanceResponse" struct describes the data that returned by the server, derived from the issuance message that is sent by the client.

```
   struct {
     ServerEvaluation evaluation<1..n>;
     ServerProof proof;
   } IssuanceResponse;
```

The value of "n" is determined by the length of the "ClientIssuanceElement" vector in the "IssuanceMessage" struct. The internal data types are described below.

```
   struct {
     opaque data<1..2^32-1>;
   } ServerEvaluation;

   struct {
     opaque data<1..2*(2^32)-1>;
   } ServerProof;
```

3.1.8. RedemptionToken

The "RedemptionToken" struct contains the data required to generate the client message in the redemption phase of the Privacy Pass protocol. This data is generated in the issuance phase of the protocol, after receiving the "IssuanceResponse" message.

```
struct {
  opaque data<1..2^32-1>;
  opaque issued<1..2^32-1>;
} RedemptionToken;
```

3.1.9. RedemptionMessage

The "RedemptionMessage" struct consists of the data that is sent by the client during the redemption phase of the protocol (Section 4.3).

```
struct {
  opaque data<1..2^32-1>;
  opaque tag<1..2^32-1>;
  opaque aux<1..2^16-1>;
} RedemptionMessage;
```

3.1.10. RedemptionResponse

The "RedemptionResponse" struct corresponds a boolean value indicating whether the "RedemptionMessage" sent by the client is valid, along with any associated data.

```
struct {
  boolean success;
  opaque additional_data<1..2^32-1>;
} RedemptionResponse;
```

3.2. 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 provided by specific instantiations or extensions.

3.2.1. PP_Server_Setup

Run by the Privacy Pass server to generate its configuration. The key-pair used in the server configuration are generated fresh on each invocation.

Inputs:

* "id": A unique identifier corresponding to the setting of "ServerConfig.id".

Outputs:

* "cfg": A "ServerConfig" struct (Section 3.1.2).

* "update": A "ServerUpdate" struct.

Throws:

* "ERR_UNSUPPORTED_CONFIG" (Section 3.3)

3.2.2. PP_Client_Setup

Run by the Privacy Pass client to generate its configuration. The input public key "pub_key" in the client configuration MUST correspond to a valid server public key.

Inputs:

* "id": A unique identifier corresponding to the setting of "ServerConfig.id".

* "update": A "ServerUpdate" struct.

Outputs:

* "cfg": A "ClientConfig" struct (Section 3.1.4).

Throws:

* "ERR_UNSUPPORTED_CONFIG" (Section 3.3)

3.2.3. PP_Generate

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

Inputs:

* "cli_cfg": A "ClientConfig" struct.

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

Outputs:

* "issuance_data": A "ClientIssuanceInput" struct.

3.2.4. PP_Issue

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

Inputs:

* "srv_cfg": A "ServerConfig" struct.

* "issuance_message": A "IssuanceMessage" struct.

Outputs:

* "issuance_response": A "IssuanceResponse" struct.

Throws:

* "ERR_MAX_EVALS" (Section 3.3)

3.2.5. PP_Process

Run by the client when processing the server response in the issuance phase of the protocol. The output of this function is an array of "RedemptionToken" objects that are unlinkable from the server's computation in "PP_Issue".

Inputs:

* "cli_cfg": A "ClientConfig" struct.

* "issuance_response": A "IssuanceResponse" struct.

* "processing_data": A "ClientIssuanceProcessing" struct.

Outputs:

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

Throws:

* "ERR_PROOF_VALIDATION" (Section 3.3)

3.2.6. PP_Redeem

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

Inputs:

* "cli_cfg": A "ClientConfig" struct.

* "token": A "RedemptionToken" struct.

* "aux": An "opaque<1..2^32-1>" type corresponding to arbitrary auxiliary data.

Outputs:

* "message": A "RedemptionMessage" struct.

3.2.7. PP_Verify

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

Inputs:

* "srv_cfg": A "ServerConfig" struct.

* "message": A "RedemptionMessage" struct.

Outputs:

* "response": A "RedemptionResponse" struct.

3.3. Error types

* "ERR_UNSUPPORTED_CONFIG": Error occurred when trying to recover configuration with unknown identifier

* "ERR_MAX_EVALS": Client attempted to invoke server issuance with number of inputs that is larger than server-specified max_evals value.

* "ERR_PROOF_VALIDATION": Client unable to verify proof that is part of the server response.

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

4. Generalized protocol overview

In this document, we wan to provide a client (C) with the capability to authenticate itself in a lightweight manner to a server (S). The authorization mechanism should not reveal to the server anything about the client; in addition, the client should not be able to forge valid credentials in situations where it does not possess any. These requirements are covered in Section 5.

In this section, we will give a broad overview of how the Privacy Pass protocol functions in achieving these goals. The generic protocol can be split into three phases: initialisation, issuance and redemption. These three phases are built upon the Privacy Pass API in Section 3. We show later (Section 6) that this API can be implemented using an underlying VOPRF protocol. We provide this extra layer of abstraction to allow building extensions into the Privacy Pass protocol that go beyond what is specified in [I-D.irtf-cfrg-voprf].

4.1. Key initialisation phase

In the initialisation phase, the server generates the configuration that it will use for future instantiations of the protocol. It MUST broadcast the configuration that it generates, along with the public key, so that clients are aware of which configuration to use when interacting with the server.

In situations where the number of clients are small, it could do this by sending the data to the client directly. But in situations where there is a large number of clients, the best way of doing is likely to be via posting this information to a public bulletin board. We assume that the server only has a single configuration in place at any one time. There are privacy restrictions related to this that are described in more detail in the architectural document [draft-davidson-pp-architecture].

We give a diagrammatic representation of the initialisation phase below.

```
  C(cfgs)                                                   S(cfg_id)
  -------------------------------------------------------------------
                              (cfg, update) = PP_Server_Setup(cfg_id)

                             update
                      <--------------------

  c_cfg = PP_Client_Setup(cfg_id,update)
  cfgs.set(update.id,c_cfg)
```

In the following (and as above), we will assume that the server "S" is uniquely identifiable by an internal attribute "id". We assume the same internal attribute exists for the public key "s_cfg.pub_key". This can be obtained, for example, by hashing the contents of the object - either the name or underlying contained bytes - using a collision-resistant hash function, such as SHA256.

Note that the client stores their own configuration in the map "cfgs" for future Privacy Pass interactions with "S".

4.2. Issuance phase

The issuance phase allows the client to construct "RedemptionToken" object resulting from an interaction with a server "S" that it has previously interacted with. We give a diagrammatic overview of the protocol below.

```
  C(cfgs,store,m)                                                S(s_cfg)
  -------------------------------------------------------------------
                              S.id
                      <------------------

  c_cfg = cfgs.get(S.id)
  issue_input = PP_Generate(c_cfg, m)
  msg = issue_input.msg_data
  process = issue_input.client_data

                               msg
                      ------------------->

                                   issue_resp = PP_Issue(s_cfg,c_dat)

                           issue_resp
                      <-------------------

  tokens = PP_Process(c_cfg,issue_resp,process)
  store[S.id].push(tokens)
```

In the diagram above, the client MUST know the supported server configuration before it interacts with the Privacy Pass API. The client input "store" is used for appending redemption tokens that are linked to the server id "S.id".

4.3. Redemption phase

The redemption phase allows the client to reauthenticate to the server, using data that it has received from a previous issuance phase. We lay out the security requirements in Section 5 that establish that the client redemption data is not linkable to any given issuance session.

```
     C(cfgs,store,aux)                                  S(s_cfg,ds_idx)
     -------------------------------------------------------------------
                                S.id
                         <------------------

     c_cfg = cfgs.get(S.id)
     token = store[S.id].pop()
     msg = PP_Redeem(c_cfg,token,aux)

                                msg
                         ------------------->

                                if (ds_idx.includes(data)) {
                                  panic(ERR_DOUBLE_SPEND)
                                }
                                resp = PP_Verify(srv_cfg,data,tag,aux)
                                if (resp.success) {
                                  ds_idx.push(data)
                                }

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

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

```
   ${C.id} .. ${S.id} .. ${current_time()} .. ${requested_resource()}
```

The usage of "current_time()" allows the server to check that the redemption request has happened in an appropriate time window. The function "requested_resource()" is an optional suffix that relates to any specific resources that the client has requested from the server, in order to trigger the authorization request.

4.3.1. Double-spend protection

To protect against clients that attempt to spend a value "data" more than once, the server uses an index, "ds_idx", to collect valid inputs and then check against in future protocols. Since this store needs to only be optimized for storage and querying, a structure such as a Bloom filter suffices. Importantly, the server MUST only eject this storage after a key rotation occurs since all previous client data will be rendered obsolete after such an event.

4.4. Handling errors

It is possible for the API functions from Section 3.2 to return one of the errors indicated in Section 3.3 rather than their expected value. In these cases, we assume that the entire protocol execution panics with the value of the error.

If a panic occurs during the server's operations for one of the documented errors, then the server returns an error response indicating the error that occurred.

5. Security requirements

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 privacy of the client in the redemption phase of the protocol. The notion of unforgeability is to protect against adversarial clients that look to subvert the security of the protocol.

Since these are cryptographic security requirements we discuss them with respect to a polynomial-time algorithm known as the adversary that is looking to subvert the security guarantee. More details on both security requirements can be found in [DGSTV18] and [KLOR20].

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

5.1. Unlinkability

Informally, the "unlinkability" requirement states that it is impossible for an adversarial server to link the client's message in a redemption session, to any previous issuance session that it has encountered.

Formally speaking the security model is the following:

* The adversary runs "PP_Server_Setup" and generates a key-pair "(k, pk)".

* The adversary specifies a number "Q" of issuance phases to initiate, where each phase "i in 1..Q" consists of "m_i" server evaluations.

* The adversary runs "PP_Issue" using the key-pair 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 "1..Q".

* A redemption phase is initiated with a single token with index "i" randomly sampled from "1..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".

5.2. One-more unforgeability

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

The security model takes the following form:

* A server is created that runs "PP_Server_Setup" and broadcasts the "ServerUpdate" message "update".

* The adversary runs "PP_Client_Setup" on "update".

* The adversary specifies a number "Q" of issuance phases to initiate with the server, where each phase "i in 1..Q" consists of "m_i" server evaluations. Let "m = sum(m_i)" where "i in 1..Q".

* The client 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.

5.3. 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 issuer individually. This prevents clients from "replaying" previous requests, and is necessary for achieving the unforgeability requirement.

6. VOPRF instantiation

In this section, we show how to instantiate the functional API in Section 3 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 5, based on the security proofs provided in [DGSTV18] and [KLOR20].

6.1. VOPRF conventions

The VOPRF ciphersuite [I-D.irtf-cfrg-voprf] that is used determines the member functions and prime-order group used by the protocol. We detail a number of specific conventions here that we use for interacting with the specific ciphersuite.

6.1.1. Ciphersuites

Let "F" denote a generic VOPRF API function as detailed in [I-D.irtf-cfrg-voprf] (Section TODO), and let "ciph" denote the ciphersuite that is used for instantiating the VOPRF. In this document, we explicitly write "ciph.F" to show that "F" is explicitly evaluated with respect to "ciph".

In addition, we define the following member functions associated with the ciphersuite.

* "recover_ciphersuite_from_id(id)": Takes a string identifier "id" as input, and outputs a VOPRF ciphersuite. Returns "null" if "id" is not recognized.

* "group()": Returns the prime-order group associated with the ciphersuite.

* "H1()": The function "H1()" defined in [I-D.irtf-cfrg-voprf] (Section TODO). This function allows deterministically mapping arbitrary bytes to a random element of the group. In the elliptic curve setting, this is achieved using the functions defined in [I-D.irtf-cfrg-hash-to-curve].

6.1.2. Prime-order group conventions

We detail a few functions that are required of the prime-order group "GG" used by the VOPRF in [I-D.irtf-cfrg-voprf].

Let "p" be the order of the Galois field "GF(p)" associated with the group "GG". We expose the following functions associated with "GG".

* "GG.generator()": Returns the fixed generator associated with the group "GG".

* "GG.scalar_field()": Provides access to the field "GF(p)".

* "GG.scalar_field().random()": Samples a scalar uniformly at random from GF(p). This can be done by sampling a random sequence of bytes that produce a scalar "r", where "r < p" is satisfied (via rejection-sampling).

We also use the following functions for transitioning between different data types.

* "as_bytes()": For a scalar element of "GG.scalar_field()", or an element of "GG"; the "as_bytes()" functions serializes the element into bytes and returns this array as output.

* "as_scalar()": Interprets a sequence of bytes as a scalar value in "GG.scalar_field()". For an array of byte arrays, we define the function "as_scalars()" to individually deserialize each of the individual byte arrays into a scalar and output a new array containing each scalar value.

* "as_element()": Interprets a sequence of bytes as a group element in "GG". For an array of byte arrays, we define the function "as_elements()" to individually deserialize each of the individual byte arrays into a single group element and output a new array containing each of these elements.

6.2. API instantiation

For the explicit signatures of each of the functions, refer to Section 3.2.

6.2.1. PP_Server_Setup

```
1. ciph = recover_ciphersuite_from_id(id)
2. if ciph == null: panic(ERR_UNSUPPORTED_CONFIG)
3. (k,Y,GG) = ciph.VerifiableSetup()
4. key = k.as_bytes()
5. pub_key = Y.as_bytes()
6. cfg = ServerConfig {
          id: id
          ciphersuite: ciph,
          key: key,
          pub_key: pub_key,
          max_evals: max_evals
      }
7. update = ServerUpdate {
             id: id
             ciphersuite: ciph,
             pub_key: pub_key,
             max_evals: max_evals
         }
8. Output (cfg, update)
```

6.2.2. PP_Client_Setup

```
1. ciph = recover_ciphersuite_from_id(id)
2. if ciph == null: panic(ERR_UNSUPPORTED_CONFIG)
3. cfg = ClientConfig {
         s: update
      }
4. Output cfg
```

6.2.3. PP_Generate

```
   1. ciph = cli_cfg.s.ciphersuite
   2. GG = ciph.group()
   3. c_data = []
   4. i_data = []
   5. g_data = []
   6. for i in 0..m:
          1. c_data[i] = GG.scalar_field().random().as_bytes()
   7. (blinds,group_elems) = ciph.VerifiableBlind(c_data)
   8. for i in 0..m:
          1. i_data[i] = group_elems[i].as_bytes()
          2. g_data[i] = blinds[i].as_bytes()
   9. Output ClientIssuanceInput {
                 ClientIssuanceProcessing {
                     client_data: c_data,
                     gen_data: g_data,
                 },
                 ClientIssuanceElement {
                     msg_data: i_data,
                 }
             }
```

6.2.4. PP_Issue

```
   1. ciph = srv_cfg.ciphersuite
   2. pk = srv_cfg.pub_key.as_element()
   3. GG = ciph.group()
   4. m = msg_data.length
   5. if m > max_evals: panic(ERR_MAX_EVALS)
   6. G = GG.generator()
   7. elts = msg_data.as_elements();
   8. Z,D = ciph.VerifiableEval(key.as_scalar(),G,pk,elts)
   9. evals = []
   10. for i in 0..m:
       1.  eval[i] = ServerEvaluation {
                       data: Z[i].as_bytes();
                     }
   11. proof = ServerProof {
                   data: D.as_bytes()
               }
   12. Output IssuanceResponse {
                   evaluations: eval,
                   proof: proof,
               }
```

6.2.5. PP_Process

```
1. ciph = cli_cfg.s.ciphersuite
2. GG = ciph.group()
3. G = GG.generator()
4. pk = cli_cfg.s.pub_key.as_element()
5. M = i_data.as_elements()
6. Z = evals.as_elements()
7. r = g_data.as_scalars()
8. N = ciph.VerifiableUnblind(G,pk,M,Z,r,proof)
9. if N == "error": panic(ERR_PROOF_VALIDATION)
10. tokens = []
11. for i in 0..m:
        1. issued = N[i].as_bytes()
        2. rt = RedemptionToken { data: c_data[i], issued: issued }
        3. tokens[i] = rt
12. Output tokens
```

6.2.6. PP_Redeem

```
1. ciph = cli_cfg.s.ciphersuite
2. GG = ciph.group()
3. token = store[S.id].pop();
4. data = token.data
5. issued = token.issued.as_element();
6. tag = ciph.VerifiableFinalize(data,issued,aux)
7. Output RedemptionMessage {
              data: data,
              tag: tag,
              aux: aux,
          }
```

6.2.7. PP_Verify

```
1. ciph = srv_cfg.ciphersuite
2. GG = ciph.group()
3. key = srv_cfg.key
4. T = ciph.H1(msg.data)
5. N' = ciph.Eval(key,T)
6. tag' = ciph.Finalize(msg.data,N',msg.aux)
7. Output RedemptionResponse {
              success: (msg.tag == tag')
          }
```

Note: at this stage we use the non-verifiable VOPRF API functions rather than the verifiable equivalents ("Eval" rather than "VerifiableEval"), as we do not need to recompute the proof data that is used for producing verifiable outputs at this stage.

6.3. Security justification

The protocol that we devise in Section 4, coupled with the API instantiation in Section 6.2, are equivalent to the protocol description in [DGSTV18]. In [DGSTV18], it is proven that this protocol satisfies the security requirements of unlinkability (Section 5.1) and unforgeability (Section 5.2).

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.

7. Ciphersuites

The Privacy Pass protocol essentially operates as a wrapper around the instantiation of the VOPRF that is used in Section 6. There is no extra cryptographic machinery used on top of what is established in the VOPRF protocol. Therefore, the ciphersuites that we support are the transitively exposed from the underlying VOPRF functionality, we detail these below. Each of the ciphersuites is detailed in [I-D.irtf-cfrg-voprf].

* VOPRF-P384-HKDF-SHA512-SSWU-RO
  - maximum security parameter: 192 bits
* VOPRF-curve448-HKDF-SHA512-ELL2-RO
  - maximum security parameter: 224 bits
* VOPRF-P521-HKDF-SHA512-SSWU-RO
  - maximum security parameter: 256 bits

When referring to the 'maximum security parameter' size above, we are referring to the _maximum_ effective key length of the ciphersuite, as specified in [NIST]. The reason that this is the maximum length

is because there may be attacks that serve to lower the actual value of the security parameter. See [I-D.irtf-cfrg-voprf] for more details.

Note than any extension to the Privacy Pass protocol that modifies either VOPRF instantiation, or the way that the Privacy Pass API is implemented, MUST specify its own ciphersuites.

8. Extensions framework policy

The intention with providing the Privacy Pass API in Section 3 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 6.

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 5. 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 privacy that the extension may introduce.

9. References

9.1. 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>.

[I-D.irtf-cfrg-hash-to-curve]
Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R., and C. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-05, 2 November 2019, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-hash-to-curve-05.txt>.

[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-02, 4 November 2019, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-voprf-02.txt>.

[NIST] "Keylength - NIST Report on Cryptographic Key Length and Cryptoperiod (2016)", n.d., <https://www.keylength.com/en/4/>.

[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>.

9.2. Informative References

[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>.

[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>.

[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>.

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

Author's Address

Alex Davidson
Cloudflare Portugal
Largo Rafael Bordalo Pinheiro 29
Lisbon
Portugal

Email: alex.davidson92@gmail.com