The Privacy Pass Protocol

draft-privacy-pass-00

Abstract

This document specifies the Privacy Pass protocol for anonymously authorizing clients with services on the Internet.

Note to Readers

Source for this draft and an issue tracker can be found at https://github.com/grittygrease/draft-privacy-pass [1].

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 May 6, 2020.

Copyright Notice

Copyright (c) 2019 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

[Page 1]
Table of Contents

1. Introduction ................................................. 3
   1.1. Terminology ........................................... 3
   1.2. Preliminaries .......................................... 4
       1.2.1. Elliptic curve points .......................... 4
       1.2.2. Protocol messages ............................ 4
   1.3. Layout ................................................ 4
   1.4. Requirements .......................................... 5
2. Generalized protocol overview ............................... 5
   2.1. Key initialisation phase .............................. 6
   2.2. Issuance phase ........................................ 8
   2.3. Redemption phase ..................................... 9
       2.3.1. Double-spend protection ....................... 10
       2.3.2. Finalization during redemption ............... 11
   2.4. Error types .......................................... 11
3. Key registration ............................................. 11
   3.1. Key rotation .......................................... 12
   3.2. Client retrieval .................................... 13
   3.3. Key revocation ....................................... 14
   3.4. VOPRF ciphersuites .................................. 14
   3.5. ECDSA key material .................................. 15
4. Protocol configurations ..................................... 15
   4.2. Single-Issuer Forwarding-Verifier .................. 16
   4.3. Single-Issue Asynchronous-Verifier ................ 16
   4.4. Bounded-Issuers ..................................... 17
       4.4.1. Fixing the bound .............................. 17
5. Privacy considerations ...................................... 17
   5.1. User segregation .................................... 18
       5.1.1. Key rotation .................................. 18
       5.1.2. Large numbers of issuers ....................... 18
   5.2. Tracking and identity leakage ....................... 20
6. Security considerations ..................................... 20
   6.1. Double-spend protection ................................ 20
   6.2. Key rotation .......................................... 20
   6.3. Token exhaustion ..................................... 21
7. Summary of privacy and security parameters ............... 21
   7.1. Justification ........................................ 22
   7.2. Example parameterization ............................. 23
8. References .................................................. 23
   8.1. Normative References ................................ 23
   8.2. URIs .................................................. 24
Authors’ Addresses ............................................. 24
1. Introduction

In some situations, it may only be necessary to check that a client has been previously authorized by a service; without learning any other information. Such lightweight authorization mechanisms can be useful in quickly assessing the reputation of a client in latency-sensitive communication.

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]. This document seeks to standardize the usage and parametrization of the protocol.

The Internet performance company Cloudflare has already implemented server-side support for an initial version of the Privacy Pass protocol [PPSRV]. This support allows clients to bypass security mechanisms, providing that they have successfully passed these mechanisms previously. There is also a client-side implementation in the form of a browser extension that interacts with the Cloudflare network [PPEXT].

The main security requirement of the Privacy Pass protocol is to ensure that previously authenticated clients do not reveal their identity on reauthorization. The protocol uses a cryptographic primitive known as a verifiable oblivious pseudorandom function (VOPRF) for implementing the authorization mechanism. The VOPRF is implemented using elliptic curves and is currently in a separate standardization process [OPRF]. The protocol is split into three stages. The first two stages, initialisation and evaluation, are essentially equivalent to the VOPRF setup and evaluation phases from [OPRF]. The final stage, redemption, essentially amounts to revealing the client’s secret inputs in the VOPRF protocol. The security (pseudorandomness) of the VOPRF protocol means that the client retains their privacy even after revealing this data.

In this document, we will give a formal specification of the Privacy Pass protocol to be used in settings that require high performance. We will specify the necessary cryptographic operations required by the underlying VOPRF, along with recommendations on how to perform key rotation.

1.1. Terminology

The following terms are used throughout this document.

- PRF: Pseudorandom function
- VOPRF: Verifiable oblivious PRF [OPRF]
o  Server: A service that provides access to a certain resource
   (typically denoted S)

o  Client: An entity that seeks authorization from a server
   (typically denoted C)

o  Key: Server VOPRF key

o  Commitment: Corresponding public key to server’s VOPRF key.

1.2. Preliminaries

Throughout this draft, let D be some object corresponding to an
opaque data type (such as a group element). We write bytes(D) to
denote the encoding of this data type as raw bytes (octet strings).
We assume that such objects can also be interpreted as Buffer
objects, with each internal slot in the buffer set to the value of
the one of the bytes. For two objects x and y, we denote the
concatenation of the bytes of these objects by (bytes(x) ..
bytes(y)). We assume that all bytes are first base64-encoded before
they are sent as part of a protocol message.

We use the notation "[ Ti \]" to indicate an array of objects T1, ...
, TQ where the size of the array is Q, and the size of Q is implicit
from context.

1.2.1. Elliptic curve points

When encoding elliptic curve points into existing data structures or
into protocol messages, we assume that the curve points are first
encoded into bytes. We allow both uncompressed and compressed
encodings, as long as the client and server are aligned on the
encodings that they used. Compressed encodings provide storage and
communication benefits but are slightly more expensive to decode.

1.2.2. Protocol messages

Protocol messages can either be encoded in raw byte format, as
base64-encoded string objects, or as JSON objects where all strings
are represented in base64-encoded format.

1.3. Layout

o  Section 2: A generic overview of the Privacy Pass protocol based
   on VOPRFs.
Section 3: Describes the format of trusted registries that are used for holding public key commitments for each of the Privacy Pass issuers.

Section 4: Details different configurations for using the Privacy Pass protocol.

Section 5: Privacy considerations and recommendations arising from the usage of the Privacy Pass protocol.

Section 6: Additional security considerations to prevent abuse of the protocol from a malicious client.

Section 7: A summary of recommended parameter settings for ensuring privacy and security features of the protocol.

1.4. 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. Generalized protocol overview

In this document, we will be assuming that a client (C) is attempting 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.

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. As we mentioned previously, the first two stages are essentially identical to the setup and evaluation phases of the VOPRF in [OPRF]. The last stage, redemption, corresponds to the client revealing their secret input data during the VOPRF protocol to the server. The server can use this data to confirm that the client has a valid VOPRF output, without being able to link the data to any individual issuance phase.

Throughout this document, we adhere to the recommendations laid out in [OPRF] in integrating the VOPRF protocol into our wider workflow. Where necessary, we lay out exactly which part of the VOPRF API we use. We stress that the generalized protocol only includes steps and messages that contain cryptographic data.
We decide against defining abstract interfaces for enclosing Privacy Pass data and functionality. Instead, we describe the Privacy Pass protocol in the same group setting that is used in [OPRF].

2.1. Key initialisation phase

In the initialisation phase, essentially we run the VOPRF setup phase in that the server runs VOPRF_Setup(l) where l is the required bit-length of the prime used in establishing the order of the group GG. This outputs the tuple \((k,Y,p)\) where: \(p = p(l)\) is the prime order of \(GG = GG(l)\); \(k\) is a uniformly sampled element from \(GF(p)\); and \(Y = kG\) for some fixed generator of \(GG\).

However, the server must first come to an agreement on what group instantiation to support. This involves choosing an instantiation with the required security level implied by the choice of \(l\). The server has a list of supported group params (GROUP_PARAMS) and chooses an identifier, \(id\), associated with the preferred group configuration, and also outputs the implied length of \(l\). It creates a Privacy Pass key object denoted by \(ppKey\) that has fields "private", "public" and "group". It sets \(ppKey.private = \text{bytes}(k)\), \(ppKey.public = \text{bytes}(Y)\) and \(ppKey.group = id\).

The server creates a JSON object of the form below.

```
{
    "Y": pp_key.public,
    "expiry": <expiry_date>,
    "sig": <signature>
}
```

The field "expiry" corresponds to an expiry date for the newly sampled key. We recommend that each key has a lifetime of between 1 month and 6 months. The field "sig" holds an ASN1-encoded ECDSA signature evaluated over the contents of "Y" and "expiry". The ECDSA parameters should be equivalent to the group instantiation used for the OPRF, and the signing key (ecdsaSK) should be long-term with a corresponding publicly available verification key (ecdsaVK). We summarize the creation of this object using the algorithm PP_key_init(), which we define below.
function PP_key_init(k, Y, id) {
    var ppKey = {};
    ppKey.private = k;
    ppKey.public = Y;
    ppKey.group = id;
    var today = new Date();
    var expiry = today.setMonth(today.getMonth() + n);
    var obj = {
        Y: ppKey.public,
        expiry: expiry,
        sig: ECDSA.sign(ecdsaSK, ppKey.public .. bytes(expiry)),
    }
    return [ppKey, obj]
}

Note that the variable n above should correspond to the number of months ahead that the expiry date should correspond to.

We give a diagrammatic representation of the initialisation phase below.

\[
\begin{align*}
C(ecdsaVK) & \quad S(ecdsaSK) \\
\hline
l = GROUP_PARAMS[id] & \qquad (k,Y,p) = VOPRF_Setup(l) \\
[ppKey,obj] = PP_key_init(k,Y,id) \\
obj & \quad \leftarrow \\
\end{align*}
\]

public = key.Y
if (!ECDSA.verify(ecdsaVK, public .. bytes(obj.expiry)) {
    panic(KEY_VERIFICATION_ERROR)
} else if (!(new Date() > obj.expiry)) {
    panic(KEY_VERIFICATION_ERROR)
}
store(obj.id, obj.public) push(key)

The variable obj essentially corresponds to a cryptographic commitment to the server’s VOPRF key. We abstract all signing and verification of ECDSA signatures into the ECDSA.sign and ECDSA.verify functionality \cite{DSS}.

In the initialisation phase above, we require that the server contacts each viable client. In Section 3 we discuss the possibility of uploading public key material to a trusted registry that client’s access when communicating with the server.
2.2. Issuance phase

The issuance phase allows the client to receive VOPRF evaluations from the server. The issuance phase essentially corresponds to a VOPRF evaluation phase \[\text{OPRF}\]. In essence, the client generates a valid VOPRF input \(x\) (a sequence of bytes from some unpredictable distribution), and runs the VOPRF evaluation phase with the server. The client receives an output \(y\) of the form:

\[
y = \text{VOPRF\_Finalize}(x, N, \text{aux})
\]

where \(N\) is a group element, and \(\text{aux}\) is auxiliary data that is generated by the client. More specifically, \(N\) is an unblinded group element equal to \(k \cdot H_1(x)\) where \(H_1\) is a random oracle that outputs elements in \(GG\). The client stores \((x, y)\) as recommended in \[\text{OPRF}\]. We give a diagrammatic overview of the protocol below.
In the diagram above, the client knows the VOPRF ciphersuite supported by the server when it retrieves in the first step. It uses this information to correctly perform group operations before sending the first message.

2.3. Redemption phase

The redemption phase allows the client to reauthenticate to the server, using data that it has received from a previous issuance phase. By the security of the VOPRF, even revealing the original input x that is used in the issuance phase does not affect the privacy of the client.
C()                                     S(ppKey)
----------------------------------------------------------------------
ciph1 = retrieve(S.id, "ciphersuite")
a = pop()
while (a != undefined) {
    (ciph2,x,y,aux) = a
    if (ciph1 != ciph2) {
        // ciphersuites do not match
        a = pop()
        continue
    }
}
if (a == undefined) {
    // no valid data to redeem
    return
}

(x,y,aux)-------------------->

if (store.includes(x)) {
    panic(DOUBLE_SPEND_ERROR)
}
T = H1(x)
N' = OPRF_Eval(ppKey.private, T)
y' = OPRF_Finalize(x,N',aux)
resp = (y' == y)
    ? "success"
    : "failure"
store.push(x)

resp<---------------------

output resp

Note that the server uses the API provided by OPRF_Eval and
OPRF_Finalize, rather than the corresponding VOPRF functions. This
is because the VOPRF functions also compute zero-knowledge proof data
that we do not require at this stage of the protocol.

2.3.1. Double-spend protection

To protect against clients that attempt to spend a value x more than
once, the server uses an index, store, 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.

2.3.2. Finalization during redemption

The last step if the issuance phase for the client is to run VOPRF_Finalize and store the output. In some applications, it may be necessary to link the output of VOPRF_Finalize to the actual redemption that is occurring. This can be done by tailoring the auxiliary data "aux" to something specific.

In order to do this, it is necessary to store only (ciph, x, N) in the issuance phase of the protocol. Then during the redemption phase, generate auxiliary data "aux" and compute VOPRF_Finalize(x,N,aux) after retrieving the triplet above.

2.4. Error types

- **KEY_VERIFICATION_ERROR**: Error occurred when verifying signature and expiry date for a server public key
- **CLIENT_VERIFICATION_ERROR**: Error verifying issuance response from server.
- **DOUBLE_SPEND_ERROR**: Indicates that a client has attempted to redeem a token that has already been used for authorization

3. Key registration

Rather than sending the result of the key initialisation procedure directly to each client, it is preferable to upload the object obj to a trusted, tamper-proof, history-preserving registry. By trusted, we mean from the perspective of clients that use the Privacy Pass protocol. Any new keys uploaded to the registry should be appended to the list. Any keys that have expired can optionally be labelled as so, but should never be removed. A trusted registry may hold key commitments for multiple Privacy Pass service providers (servers).

Clients can either choose to:

- poll the trusted registry and import new keys, rejecting any that throw errors;
- retrieve the commitments for the server at the time at which they are used, throwing errors if no valid commitment is available.

To prevent unauthorized modification of the trusted registry, server’s should be required to identify and authenticate themselves
before they can append data to their configuration. Moreover, only parts of the registry that correspond to the servers configuration can be modifiable.

The registry that we describe could be fulfilled by Key Transparency [keytrans] or other similar architectures.

3.1. Key rotation

Whenever a server seeks to rotate their key, they must append their key to the trusted registry. We recommend that the trusted registry is arranged as a JSON blob with a member for each JSON provider. Each provider appends new keys by creating a new sub-member corresponding to an incremented version label along with their new commitment object.

Concretely, we recommend that the trusted registry is a JSON file of the form below.

```
{
  "server_1": {
    "ciphersuite": ..., 
    "1.0": {
      "Y": ..., 
      "expiry": ..., 
      "sig": ..., 
    },
    "1.1": {
      "Y": ..., 
      "expiry": ..., 
      "sig": ..., 
    },
  },
  "server_2": {
    "ciphersuite": ..., 
    "1.0": {
      "Y": ..., 
      "expiry": ..., 
      "sig": ..., 
    },
  },
  ...
}
```

In this structure, "server_1" and "server_2" are separate service providers. The sub-member "ciphersuite" corresponds to the choice of VOPRF ciphersuite made by the server. The sub-members "1.0", "1.1" of "server_1" correspond to the versions of commitments available to
the client. Increasing version numbers should correspond to newer keys. Each commitment should be a valid encoding of a point corresponding to the group in the VOPRF ciphersuite specified in "ciphersuite".

If "server_2" wants to upload a new commitment with version tag "1.1", it runs the key initialisation procedure from above and adds a new sub-member "1.1" with the value set to the value of the output obj. The "server_2" member should now take the form below.

```
{
  ...
  "server_2": {
    "ciphersuite": ...,
    "1.0": {
      "Y": ...,
      "expiry": ...,
      "sig": ...
    },
    "1.1": {
      "Y": ...,
      "expiry": ...
    },
    ..., 
  } ...
}
```

### 3.2. Client retrieval

We define a function "retrieve(server_id, version_id)" which retrieves the commitment with version label equal to version_id, for the provider denoted by the string server_id. For example, `retrieve("server_1","1.1")` will retrieve the member labelled with "1.1" above.

We implicitly assume that this function performs the following verification checks:

```
if (!ECDSA.verify(ecdsaVK, obj.Y .. bytes(obj.expiry)) {
  return "error"
} else if (!(new Date() < obj.expiry)) {
  return "error"
}
```

If "error" is not returned, then it instead returns the entire object. We also abuse notation and also use "ciph =
retrieve(server_id, "ciphersuite")" to refer to retrieving the ciphersuite for the server configuration.

### 3.3. Key revocation

If a server must revoke a key, then it uses a separate member with label "revoke" corresponding to an array of revoke versions associated with key commitments. In the above example, if "server_2" needs to revoke the key with version "1.0", then it appends a new "revoke" member with the array "[ "1.0" ]". Any future revocations can simply be appended to this array. For an example, see below.

```json
{
  ...
  "server_2": {
    "ciphersuite": ...,
    "1.0": {
      "Y": ...,
      "expiry": ...,
      "sig": ...,
    },
    "1.1": {
      "Y": ...,
      "expiry": ...,
      "sig": ...,
    },
    "revoked": [ "1.0" ],
  },
  ...
}
```

Client’s are required to check the "revoked" member for new additions when they poll the trusted registry for new key data.

### 3.4. VOPRF ciphersuites

We strongly RECOMMEND that a server uses only one VOPRF ciphersuite at any one time. Should a server choose to change some aspect of the ciphersuite (e.g., the group instantiation or other cryptographic functionality) we further RECOMMEND that the server create a new identifying label (e.g. "server_1_${ciphersuite_id}" where ciphersuite_id corresponds to the identifier of the VOPRF ciphersuite. Then "server_1" revokes all keys for the previous ciphersuite and then only offers commitments for the current label.

An alternative arrangement would be to add a new layer of members between server identifiers and key versions in the JSON struct,
corresponding to "ciphersuite_id". Then the client may choose commitments from the appropriate group identifying member.

We strongly recommend that service providers only operate with one group instantiation at any one time. If a server uses two VOPRF ciphersuites at any one time then this may become an avenue for segregating the user-base. User segregation can lead to privacy concerns relating to the utility of the obliviousness of the VOPRF protocol (as raised in [OPRF]). We discuss this more in Section 5.

3.5. ECDSA key material

For clients must also know the verification (ecdsaVK) for each service provider that they support. This enables the client to verify that the commitment is properly formed before it uses it. We do not provide any specific recommendations on how the client has access to this key, beyond that the verification key should be accessible separately from the trusted registry.

While the number of service providers associated with Privacy Pass is low, the client can simply hardcode the verification keys directly for each provider that they support. This may be cumbersome if a provider wants to rotate their signing key, but since these keys should be comparatively long-term (relative to the VOPRF key schedule), then this should not be too much of an issue.

4. Protocol configurations

We provide an overview of some of the possible ways of configuring the Privacy Pass protocol situation, such that it can be used as a lightweight trust attestation mechanism for clients.


The simplest way of considering the Privacy Pass protocol is in a setting where the same server plays the role of issuer and verifier, we call this "Single-Issuer Single-Verifier" (SISV). In SISV, we consider a server S that publishes commitments for their secret key k, that a client C has access to.

When S wants to issue tokens to C, they invoke the issuance protocol where C generates their own inputs and S uses their secret key k. In this setting, C can only perform token redemption with S. When a token redemption is required, C and S invoke the redemption phase of the protocol, where C uses an issued token from a previous exchange, and S uses k as their input again.
In SISV, C proves that S has attested to the honesty of C at some point in the past (without revealing exactly when). S can use this information to inform its own decision-making about C without having to recompute the trust attestation task again.

4.2. Single-Issuer Forwarding-Verifier

In this setting, each client C obtains issued tokens from a server S via the issuance phase of the protocol. The difference is that clients can prove that S has attested to their honesty in the past with any verifier V. We still only consider S to hold their own secret key.

When C interacts with V, V can ask C to provide proof that the separate issuer S has attested to their trust. The first stage of the redemption phase of the protocol is invoked between C and V, which sees C send the unused token \((x,y,\text{aux})\) to V. This message is then used in a redemption exchange between V and S, where V plays the role of the client. Then S sends the result of the redemption exchange to V, and V uses this result to determine whether C has the correct trust attestation.

This configuration is known as "Single-Issuer Forwarding-Verifier" or SIFV to refer to the verifier V who uses the output of the redemption phase for their own decision-making.

4.3. Single-Issue Asynchronous-Verifier

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

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

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

The "signature" field carries a signature evaluated over the contents of "result" using a long-term signing key for the issuer S, of which the corresponding public key is well-known to C and V. Then C can
prove that their trust attestation from S to V by sending the SRR to V. The SRR can be verified by V by verifying the signature using the well-known public key for S.

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

4.4. Bounded-Issuers

Each of the configurations above can be generalized to settings where a bounded number of issuers are allowed, and verifiers can invoke trust attestations for any of the available issuers. Subsequently, this leads to three new configurations known as BISV, BIFV, BIAV.

As we will discuss later in Section 5.1.2, configuring a large number of issuers can lead to privacy concerns for the clients in the ecosystem. Therefore, we are careful to ensure that the number of issuers is kept strictly bounded by a fixed small number $M$. The actual issuers can be replaced with different issuers as long as the total never exceeds $M$. Moreover, issuer replacements also have an effect on client privacy that is similar to when a key rotation occurs, so replacement should only be permitted at similar intervals.

See Section 5.1.2 for more details about safe choices of $M$.

4.4.1. Fixing the bound

Configuring any number of issuers greater than 1 effectively reduces privacy by an extra bit. As a result, we see an exponential decrease in privacy in the number of issuers that are currently active. Therefore the value of $M$ should be kept very low (we recommend no higher than 4).

5. Privacy considerations

We intentionally encode no special information into redemption tokens to prevent a vendor from learning anything about the client. We also have cryptographic guarantees via the VOPRF construction that a vendor can learn nothing about a client beyond which issuers trust it. Still there are ways that malicious servers can try and learn identifying information about clients that it interacts with.

We discuss a number of privacy considerations made in [OPRF] that are relevant to the Privacy Pass protocol use-case, along with additional considerations arising from the specific ways of using the Privacy Pass protocol in Section 4.
5.1. User segregation

An inherent features of using cryptographic primitives like VOPRFs is that any client can only remain private relative to the entire space of users using the protocol. In principle, we would hope that the server can link any client redemption to any specific issuance invocation with a probability that is equivalent to guessing. However, in practice, the server can increase this probability using a number of techniques that can segregate the user space into smaller sets.

5.1.1. Key rotation

As introduced in [OPRF], such techniques to introduce segregation are closely linked to the type of key schedule used by the server. When a server rotates their key, any client that invokes the issuance protocol shortly afterwards will be part of a small number of possible clients that can redeem. To mechanize this attack strategy, a server could introduce a fast key rotation policy which would force clients into small key windows. This would mean that client privacy would only have utility with respect to the smaller group of users that have Trust Tokens for a particular key window.

In the [OPRF] draft it is recommended that great care is taken over key rotations, in particular server’s should only invoke key rotation for fairly large periods of time such as between 1 and 12 months. Key rotations represent a trade-off between client privacy and continued server security. Therefore, it is still important that key rotations occur on a fairly regular cycle to reduce the harmfulness of a server key compromise.

Trusted registries for holding Privacy Pass key commitments can be useful in policing the key schedule that a server uses. Each key must have a corresponding commitment in this registry so that clients can verify issuance responses from servers. Clients may choose to inspect the history of the registry before first accepting redemption tokens from the server. If a server has updated the registry with many unexpired keys, or in very quick intervals a client can choose to reject the tokens.

TODO: Can client’s flag bad server practices?

5.1.2. Large numbers of issuers

Similarly to the key rotation issue raised above, if there are a large number of issuers, similar user segregation can occur. In the BISV, BIFV, BIAV configurations of using the Privacy Pass protocol (Section 4), a verifier OV can trigger redemptions for any of the
available issuers. Each redemption token that a client holds essentially corresponds to a bit of information about the client that OV can learn. Therefore, there is an exponential loss in privacy relative to the number of issuers that there are.

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

As we noted in Section 4.4, a strict bound should be applied to the active number of issuers that are allowed at one time. We propose that allowing no more than 6 issuers at any one time is highly preferable (leading to a maximum of 64 possible user segregations). Issuer replacements should only occur with the same frequency as key rotations as they can lead to similar losses in privacy if users still hold redemption tokens for previously active issuers.

In addition, we recommend that trusted registries indicate at all times which issuers are deemed to be active. If a client is asked to invoke any Privacy Pass exchange for an issuer that is not declared active, then the client should refuse to participate in the protocol.

5.1.2.1. Selected trusted registries

One recommendation is that only a fixed number (TODO: how many?) of issuers are sanctioned to provide redemption tokens at any one time. This could be enforced by the trusted registry that is being used. Client’s can then choose which registries to trust and only accept redemption tokens from issuers accepted into those registries.

5.1.2.2. Maximum number of issuers inferred by client

A second recommendation is that clients only store redemption tokens for a fixed number of issuers at any one time. This would prevent a malicious verifier from being able to invoke redemptions for many issuers since the client would only be holding redemption tokens for a small set of issuers. When a client is issued tokens from a new issuer and already has tokens from the maximum number of issuers, it simply deletes the oldest set of redemption tokens in storage and then stores the newly acquired tokens.
5.2. Tracking and identity leakage

While redemption tokens themselves encode no information about the client redeeming them, there may be problems if we allow too many redemptions on a single page. For instance, the first-party cookie for user U on domain A can be encoded in the trust token information channel and decoded on domain B, allowing domain B to learn the user's domain A cookie until either first-party cookie is cleared. Mitigations for this issue are similar to those proposed in Section 5.1.2 for tackling the problem of having large number of issuers.

In SIAV, cached SRRs and their associated issuer public keys have a similar tracking potential to first party cookies in the browser setting. Therefore these should be clearable by the client using standard deletion methods.

6. Security considerations

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

6.1. Double-spend protection

All issuing server should implement a robust storage-query mechanism for checking that tokens sent by clients have not been spent before. Such tokens only need to be checked for each issuer individually. But all issuers must perform global double-spend checks to avoid clients from exploiting the possibility of spending tokens more than once against distributed token checking systems. For the same reason, the global data storage must have quick update times. While an update is occurring it may be possible for a malicious client to spend a token more than once.

6.2. Key rotation

We highlighted previously that short key-cycles can be used to reduce client privacy. However, regular key rotations are still recommended to maintain good server key hygiene. The key material that we consider to be important are:

- the VOPRF key;
- the signing key used to sign commitment information;
- the signing key used to sign SRRs in the SIAV configuration.
In summary, our recommendations are that VOPRF keys are rotated from anywhere between a month and a single year. With an active user-base, a month gives a fairly large window for clients to participate in the Privacy Pass protocol and thus enjoy the privacy guarantees of being part of a larger group. The low ceiling of a year prevents a key compromise from being too destructive. If a server realizes that a key compromise has occurred then the server should revoke the previous key in the trusted registry and specify a new key to be used.

For the two signing keys, these should both be well-known keys associated with the issuer (TODO: where should they be stored?). Issuers may choose to use the same key for both signing purposes. The rotation schedules for these keys can be much longer, if necessary.

6.3. Token exhaustion

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

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

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

7. Summary of privacy and security parameters

We provide a summary of the parameters that we use in the Privacy Pass protocol. These parameters are informed by both privacy and security considerations that are highlighted in Section 5 and Section 6, respectively. These parameters are intended as a single reference point for implementers when implementing the protocol.
Firstly, let $U$ be the total number of users, $I$ be the total number of issuers. Assuming that each user accept tokens from a uniform sampling of all the possible issuers, as a worst-case analysis, this segregates users into a total of $2^I$ buckets. As such, we see an exponential reduction in the size of the anonymity set for any given user. This allows us to specify the privacy constraints of the protocol below, relative to the setting of $A$.

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum anonymity set size ($A$)</td>
<td>5000</td>
</tr>
<tr>
<td>Recommended key lifetime ($L$)</td>
<td>1 - 6 months</td>
</tr>
<tr>
<td>Recommended key rotation frequency ($F$)</td>
<td>$L/2$</td>
</tr>
<tr>
<td>Maximum allowed issuers ($I$)</td>
<td>$\log_2(U/A)-1$</td>
</tr>
<tr>
<td>Maximum active issuance keys</td>
<td>1</td>
</tr>
<tr>
<td>Maximum active redemption keys</td>
<td>2</td>
</tr>
<tr>
<td>Minimum security parameter</td>
<td>196 bits</td>
</tr>
</tbody>
</table>

7.1. Justification

We make the following assumptions in these parameter choices.

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

The maximum choice of $I$ is based on the equation $1/2 * U/2^I = A$. This is because $I$ issuers lead to $2^I$ segregations of the total user-base $U$. By reducing $I$ we limit the possibility of performing the attacks mentioned in Section 5.1.

We must also account for each user holding issued data for more than one possible active keys. While this may also be a vector for monitoring the access patterns of clients, it is likely to unavoidable that clients hold valid issuance data for the previous key epoch. This also means that the server can continue to verify redemption data for a previously used key. This makes the rotation period much smoother for clients.
For privacy reasons, it is recommended that key epochs are chosen that limit clients to holding issuance data for a maximum of two keys. By choosing $F = L/2$ then the minimum value of $F$ is $1/2$ a month, since the minimum recommended value of $L$ is 1 month. Therefore, by the initial assumption, then all users should only have access to only two keys at any given time. This reduces the anonymity set by another half at most.

Finally, the minimum security parameter size is related to the cryptographic security offered by the group instantiation that is chosen. For example, if we use an elliptic curve over a 256-bit prime field, then the actual group instantiation offers 128 bits of security (or a security parameter of size 128 bits). However, as noted in [OPRF], OPRF protocols reduce the effective security of the group by $\log_2(M)$ where $M$ is the number of queries. As such, we choose the minimum size of the security parameter to be 196 bits, so that it is difficult for a malicious client to exploit this.

7.2. Example parameterization

Using the specification above, we can give some example parameterizations. For example, the current Privacy Pass browser extension [PPEXT] has over 150,000 active users (from Chrome and Firefox). Then $\log_2(U/A)$ is approximately 5 and so the maximum value of $I$ should be 4.

If the value of $U$ is much bigger (e.g. 5 million) then this would permit $I = \log_2(5000000/5000)-1 = 8$ issuers.

8. References

8.1. Normative References


8.2. URIs


Authors’ Addresses

Alex Davidson
Cloudflare Portugal
Largo Rafael Bordalo Pinheiro 29
Lisbon
Portugal

Email: alex.davidson92@gmail.com

Nick Sullivan
Cloudflare
101 Townsend Street
San Francisco
United States of America

Email: nick@cloudflare.com