Workgroup: Network Working Group
Internet-Draft:
draft-robert-privacypass-batched-tokens-01
Published: 13 March 2023
Intended Status: Informational
Expires: 14 September 2023
Authors: R. Robert
Phoenix R&D
C. A. Wood
Cloudflare

Batched Token Issuance Protocol

Abstract

This document specifies a variant of the Privacy Pass issuance protocol that allows for batched issuance of tokens. This allows clients to request more than one token at a time and for issuers to isse more than one token at a time.

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 September 2023.

Table of Contents

1. Introduction

This document specifies a variant of the Privacy Pass issuance protocol (as defined in [ARCH]) that allows for batched issuance of tokens. This allows clients to request more than one token at a time and for issuers to isse more than one token at a time.

The base Privacy Pass issuance protocol [ISSUANCE] defines stateless anonymous tokens, which can either be publicly verifiable or not. While it is possible to run multiple instances of the issuance protocol in parallel, e.g., over a multiplexed transport such as HTTP/3 [HTTP3], the cost of doing so scales linearly with the number of instances.

This variant builds upon the privately verifiable issuance protocol that uses VOPRF [OPRF], and allows for batched issuance of tokens. This allows clients to request more than one token at a time and for issuers to issue more than one token at a time. In effect, batched issuance performance scales better than linearly.

This issuance protocol registers the batched token type (Section 7.1), to be used with the PrivateToken HTTP authentication scheme defined in [AUTHSCHEME].

2. Motivation

Privately Verifiable Tokens (as defines in [ISSUANCE]) offer a simple way to unlink the issuance from the redemption. The base protocol however only allows for a single token to be issued at a time for every challenge. In some cases, especially where a large number of clients need to fetch a large number of tokens, this may introduce performance bottlenecks. The Batched Token Issuance

Protocol improves upon the basic Privately Verifiable Token issuance protocol in the following key ways:

1. Issuing multiple tokens at once in response to a single TokenChallenge, thereby reducing the size of the proofs required for multiple tokens.

2. Improving server and client issuance efficiency by amortizing the cost of the VOPRF proof generation and verification, respectively.

3. Client-to-Issuer Request

Except where specified otherwise, the client follows the same protocol as described in [ISSUANCE], Section 5.1.

The Client first creates a context as follows:

```
client_context = SetupVOPRFClient("ristretto255-SHA512", pkI)
```

Here, "ristretto255-SHA512" is the identifier corresponding to the OPRF(ristretto255, SHA-512) ciphersuite in [OPRF]. SetupVOPRFClient is defined in [OPRF], Section 3.2.

Nr denotes the number of tokens the clients wants to request. For every token, the Client then creates an issuance request message for a random value nonce with the input challenge and Issuer key identifier as described below:

```
nonce_i = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0xF91A, nonce_i, challenge_digest, key_id)
blind_i, blinded_element_i = client_context.Blind(token_input)
```

The above is repeated for each token to be requested. Importantly, a fresh nonce MUST be sampled each time.

The Client then creates a TokenRequest structured as follows:

```
struct {
    uint8_t blinded_element[Ne];
} BlindedElement;

struct {
   uint16_t token_type = 0xF91A;
   uint8_t token_key_id;
   BlindedElement blinded_elements<0..2^16-1>;
} 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_elements" is a list of Nr serialized elements, each of length Ne bytes and computed as SerializeElement(blinded_element_i), where blinded_element_i is the i-th output sequence of Blind invocations above. Ne is as defined in [OPRF], Section 4.

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

- The TokenRequest contains a supported token_type equal to 0xF91A.

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

- Nr, as determined based on the size of TokenRequest.blinded_elements, is less than or equal to the number of tokens that the issuer can issue in a single batch.

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

4. Issuer-to-Client Response

Except where specified otherwise, the client follows the same protocol as described in [ISSUANCE], Section 5.2.

Upon receipt of a TokenRequest, the Issuer tries to deseralize the i-th element of TokenRequest.blinded_elements using DeserializeElement from Section 2.1 of [OPRF], yielding blinded_element_i of type Element. If this fails for any of the TokenRequest.blinded_elements values, the Issuer MUST return an HTTP 400 error to the client. Otherwise, if the Issuer is willing to produce a token to the Client, the issuer forms a list of Element values, denoted blinded_elements, and computes a blinded response as follows:

```
server_context = SetupVOPRFServer("ristretto255-SHA512", skI, pkI)
evaluated_elements, proof = server_context.BlindEvaluateBatch(skI, blind
```

SetupVOPRFServer is defined in [OPRF], Section 3.2. The issuer uses a list of blinded elements to compute in the proof generation step. The BlindEvaluateBatch function is a batch-oriented version of the

BlindEvaluate function described in [OPRF], Section 3.3.2. The description of BlindEvaluateBatch is below.

```
Input:

  Element blindedElements[Nr]

Output:

  Element evaluatedElements[Nr]
  Proof proof

Parameters:

  Group G
  Scalar skS
  Element pkS

def BlindEvaluateBatch(blindedElements):
  evaluatedElements = []
  for blindedElement in blindedElements:
    evaluatedElements.append(skS * blindedElement)

  proof = GenerateProof(skS, G.Generator(), pkS,
                        blindedElements, evaluatedElements)
  return evaluatedElements, proof
```

The Issuer then creates a TokenResponse structured as follows:

```
struct {
    uint8_t evaluated_element[Ne];
} EvaluatedElement;

struct {
   EvaluatedElement evaluated_elements<0..2^16-1>;
   uint8_t evaluated_proof[Ns + Ns];
} TokenResponse;
```

The structure fields are defined as follows:

* "evaluated_elements" is a list of Nr serialized elements, each of length Ne bytes and computed as SerializeElement(evaluate_element_i), where evaluate_element_i is the i-th output of BlindEvaluate.

* "evaluated_proof" is the (Ns+Ns)-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.

5. Finalization

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

```
authenticator_values = client_context.FinalizeBatch(token_input, blind,
```

The FinalizeBatch function is a batched variant of the Finalize function as defined in [OPRF], Section 3.3.2. FinalizeBatch accepts lists of evaluated elements and blinded elements as input parameters, and is implemented as described below:

```
Input:

  PrivateInput input
  Scalar blind
  Element evaluatedElements[Nr]
  Element blindedElements[Nr]
  Proof proof

Output:

  opaque output[Nh * Nr]

Parameters:

  Group G
  Element pkS

Errors: VerifyError

def FinalizeBatch(input, blind, evaluatedElements, blindedElements, proo
  if VerifyProof(G.Generator(), pkS, blindedElements,
                 evaluatedElements, proof) == false:
    raise VerifyError

  output = nil
  for evaluatedElement in evaluatedElements:
    N = G.ScalarInverse(blind) * evaluatedElement
    unblindedElement = G.SerializeElement(N)
    hashInput = I2OSP(len(input), 2) || input ||
                I2OSP(len(unblindedElement), 2) || unblindedElement ||
                "Finalize"
    output = concat(output, Hash(hashInput))

  return output
```

If this succeeds, the Client then constructs Nr Token values as follows, where authenticator is the i-th Nh-byte length slice of authenticator_values that corresponds to nonce, the i-th nonce that was sampled in Section 3:

```
struct {
    uint16_t token_type = 0xF91A
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nh];
} Token;
```

If the FinalizeBatch function fails, the Client aborts the protocol.

6. Security considerations

Implementors SHOULD be aware of the security considerations described in [OPRF], Section 6.2.3 and implement mitigation mechanisms. Application can mitigate this issue by limiting the number of clients and limiting the number of token requests per client per key.

7. IANA considerations

7.1. Token Type

This document updates the "Token Type" Registry ([AUTHSCHEME]) with the following value:

Value	Name	Publicly Verifiable	Public Metadata	Private Metadata	Nk	Reference
0xF91A	Batched Token VOPRF (ristretto255, SHA-512)	N	N	N	32	This document

Table 1: Token Types

8. References

8.1. Normative References

[ARCH] Davidson, A., Iyengar, J., and C. A. Wood, "The Privacy Pass Architecture", Work in Progress, Internet-Draft, draft-ietf-privacypass-architecture-11, 6 March 2023,

<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-architecture-11>.

[AUTHSCHEME] Pauly, T., Valdez, S., and C. A. Wood, "The Privacy Pass HTTP Authentication Scheme", Work in Progress, Internet-Draft, draft-ietf-privacypass-auth-scheme-09, 6 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-auth-scheme-09>.

[ISSUANCE] Celi, S., Davidson, A., Faz-Hernandez, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-10, 6 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-10>.

[OPRF] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-21, 21 February 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-21>.

8.2. Informative References

[HTTP3] Bishop, M., Ed., "HTTP/3", RFC 9114, DOI 10.17487/RFC9114, June 2022, <https://www.rfc-editor.org/rfc/rfc9114>.

Authors' Addresses

Raphael Robert
Phoenix R&D

Email: ietf@raphaelrobert.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net