Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 17 August 2026

S. Schlesinger
Google
T. Meunier
Cloudflare Inc.

13 February 2026

# Privacy Pass Issuance Protocol for Anonymous Credit Tokens
draft-schlesinger-privacypass-act-01

## Abstract

This document specifies the issuance and redemption protocols for tokens based on the Anonymous Credit Tokens (ACT) protocol.

## About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://SamuelSchlesinger.github.io/draft-act/draft-schlesinger-privacypass-act.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-schlesinger-privacypass-act/.

Discussion of this document takes place on the PRIVACYPASS Privacy Pass mailing list (mailto:privacypass@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/privacy-pass. Subscribe at https://www.ietf.org/mailman/listinfo/privacypass/.

Source for this draft and an issue tracker can be found at https://github.com/SamuelSchlesinger/draft-act.

## Status of This Memo

This Internet-Draft will expire on 17 August 2026.



Table of Contents

## 1. Introduction

[ARCHITECTURE] describes the Privacy Pass architecture, and [ISSUANCE] and [AUTHSCHEME] describe the issuance and redemption protocols for basic Privacy Pass tokens, i.e., those computed using blind RSA signatures as specified in Section 6 of [ISSUANCE] or verifiable oblivious pseudorandom functions as specified in Section 5 of [ISSUANCE]. Further, the [ARC] scheme and its associated integration in [ARCHITECTURE] ([ARC_PP]) extend these approaches to multi-use tokens.

The Anonymous Credit Tokens (ACT) protocol, as specified in [ACT], offers a differentiated approach to rate limiting from [ARC]. In particular, a credential initially holding N credits, along with its subsequent refunded credentials, can be presented up to N times. When a client spends a certain number of credits from a credential, that credential is invalidated and the client receives a new credential with the remaining balance.

This document specifies the issuance and redemption protocols for ACT. Section 2 describes motivation for this new type of token, Section 4 presents an overview of the protocols, and the remainder of the document specifies the protocols themselves.

## 2. Motivation

To demonstrate how ACT is useful, one can use a similar example to the one presented in Section 2 of [ARC_PP]: a client that wishes to keep its IP address private while accessing a service. [ARC_PP] offers the origin to limit the number of requests a client can make to N. This is enforced by each origin getting its own presentation context, and limiting the number of presentations per context to N. This means that, from a single credential, a client can produce N presentations and access the system N times, unlinkably. These presentations can be generated in parallel.

ACT takes a different approach. Consider a credential initially holding N credits. A client redeeming all N credits individually has to spend 1, receive a refunded credential with N-1 credits, spend 1 from that credential, receive a refunded credential with N-2 credits, and so on. Because the client cannot spend from a credential until they receive the refunded credential from their previous spend, a single live session is enforced per initial credential. This provides concurrency control. A client is also able to spend more than 1 credit at once, allowing for more efficient redemption of

multiple credits. Finally, as each new presentation requires obtaining a refunded credential from the previous spend, the origin gains the ability to invalidate a session by declining to issue a refunded credential. This creates the ability to shed harmful future traffic or redirect it in a favorable way.

Example use cases include privacy proxies, privately accessing web APIs such as artificial intelligence models, and zero trust networks that act as forward proxies for their user traffic.

Therefore, ACT provides the following properties:

1. Concurrency control: Preventing multiple simultaneous uses of the same credential, mitigating abuse from credential sharing or replay.

2. Dynamic Revocation: Enabling immediate invalidation of credentials in response to origin policy, without waiting for credential expiry.

3. Per-Session Rate Limiting: Enforcing access policies that adapt to user, device, or risk context, rather than static per-credential limits. This creates incentives for platforms to deploy such methods.

## 3. 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.

This document uses the terms Origin, Client, Issuer, and Token as defined in Section 2 of [ARCHITECTURE]. Moreover, the following additional terms are used throughout this document.

* Issuer Public Key: The public key (from a private-public key pair) used by the Issuer for issuing and verifying Tokens.

* Issuer Private Key: The private key (from a private-public key pair) used by the Issuer for issuing and verifying Tokens.

The following terms are used as defined in [ACT]: Credit, Token, Nullifier, Scalar, Element, and Domain Separator.

Unless otherwise specified, this document encodes protocol messages in TLS notation from Section 3 of [TLS13]. Moreover, all constants are in network byte order.

## 4. Protocol Overview

The issuance and redemption protocols defined in this document are built on the Anonymous Credit Tokens (ACT) protocol. ACT credentials can be thought of as single use credentials, similar to the RSA Blind Signatures protocol. However, by another viewpoint, they might be thought of as stateful, multi-use credentials.

With ACT, Clients receive TokenChallenge inputs from the redemption protocol ([AUTHSCHEME], Section 2.1). If they have a valid ACT credential for the designated Issuer, Clients can use the TokenChallenge to produce a single token for presentation. Otherwise, Clients invoke the issuance protocol to obtain an ACT credential. This interaction is shown below.

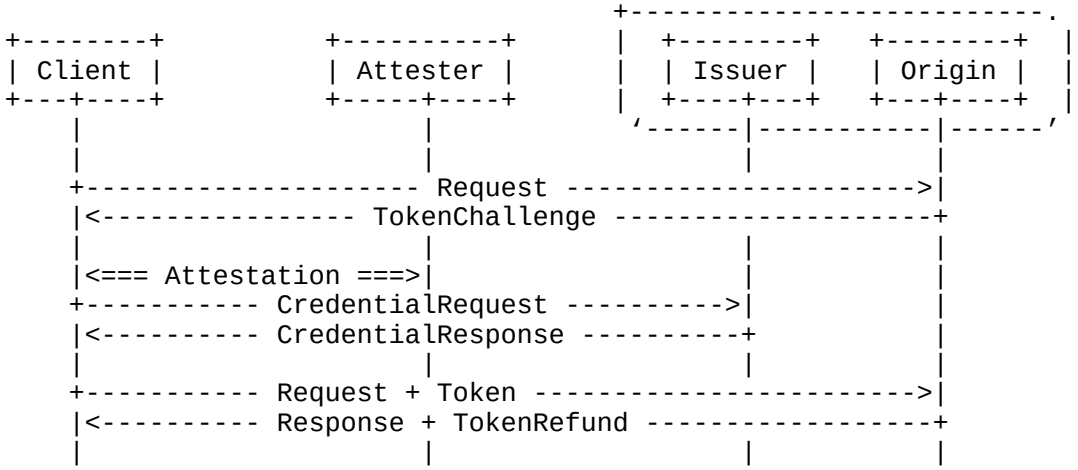



Figure 1: Issuance and Redemption Overview

Similar to the core Privacy Pass protocols, the TokenChallenge can be interactive or non-interactive, and per-origin or cross-origin.

ACT is only compatible with deployment models where the Issuer and Origin are operated by the same entity (see Section 4 of [ARCHITECTURE]), as tokens produced from a credential are not publicly verifiable. The details of attestation are outside the scope of the issuance protocol; see Section 4 of [ARCHITECTURE] for information about how attestation can be implemented in each of the relevant deployment models.

The issuance and redemption protocols in this document are built on [ACT].

## 5. Client State Management

### 5.1. Credential Lifecycle and Distributed Transactions

ACT credentials follow a distributed transaction model where each spend operation creates a two-phase interaction between the client and the origin:

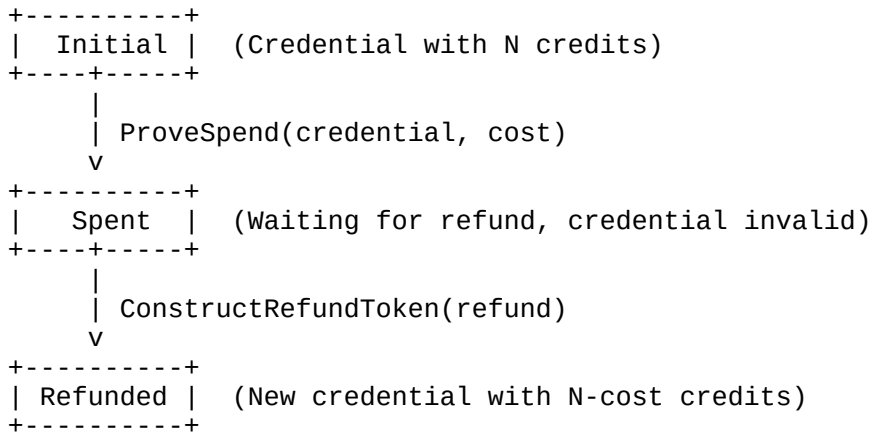
1. *Spend Phase*: The client commits to spending a credential by generating a spend proof containing a nullifier. At this point, the credential MUST be considered invalid and cannot be reused.

2. *Refund Phase*: The origin verifies the spend proof and, if valid, returns a refund that allows the client to construct a new credential with the remaining balance.

This two-phase model is critical for ACT's concurrency control properties. Because a credential becomes invalid immediately upon spend (before the refund is received), a client cannot perform multiple concurrent spend operations with the same credential. This enforces a strict serialization of operations per credential chain.

### 5.2. Client State Transitions

A client managing an ACT credential progresses through the following states:




State transitions:

* *Initial*: Client holds a valid credential with N credits (N > 0). The credential can be spent.

* *Spent*: Client has generated a spend proof and sent it to the origin. The original credential is now invalid and MUST NOT be used again. The client is blocked waiting for a refund response.

* *Refunded*: Client has received a valid refund and constructed a new credential with the remaining balance. If the new balance is greater than 0, the client returns to the Initial state with the new credential. If the balance is 0, the credential is exhausted.

The transition from Spent to Refunded may fail if the origin does not provide a valid refund. Error handling for this case is described in Section 5.5.

### 5.3. Concurrency Control Through Blocking

The key insight enabling ACT's concurrency control is that each credential instance can only be in one state at a time. When a client calls ProveSpend, it must immediately transition the credential to the Spent state, making it unavailable for any other operations. The client cannot perform another spend until it receives a refund and transitions to the Refunded state with a new credential instance.

This blocking behavior prevents credential sharing and concurrent usage:

* If a malicious party attempts to copy a credential and use it elsewhere, only one spend operation can succeed (whichever reaches the origin first). The second spend will be rejected as a double-spend when the origin checks the nullifier.

* Even if a single client attempts to perform concurrent spends with the same credential, the first spend invalidates the credential, preventing the second.

The origin can enforce session behavior by declining to issue a refund, effectively terminating the credential chain and preventing further operations.

### 5.4. Multiple Credential Management

Clients MAY maintain multiple independent credential chains simultaneously. This is useful when credentials are bound to different contexts via the credential_context field (Section 7).

Example: A client might hold:

* Credential A: Bound to issuer1.example, origin1.example, credential_context = "session-2024-01", with 100 credits

* Credential B: Bound to issuer1.example, origin2.example, credential_context = "session-2024-01", with 50 credits

* Credential C: Bound to issuer1.example, origin1.example, credential_context = "session-2024-02", with 75 credits

Each credential chain operates independently with its own state machine. A spend operation on Credential A does not affect Credentials B or C. The client can perform spend operations on different credentials concurrently, but each individual credential chain is strictly serialized through the spend-refund cycle.

When the client receives a TokenChallenge, it determines which credential to use based on the challenge's issuer_name, origin_info, and credential_context fields. If no matching credential exists or all matching credentials are in the Spent state (awaiting refund), the client SHOULD either wait for a refund or request a new credential through the issuance protocol.

## 5.5. Error Handling

### 5.5.1. Delayed or Missing Refunds

After a client sends a spend proof to the origin, the client must wait for the corresponding refund before it can continue using the credential chain. The spent credential instance is immediately invalid and MUST NOT be reused.

Origins MAY retain spend proof state and serve delayed refund requests after service interruptions or failures. This allows credential chains to resume even if the initial refund response was not delivered due to network issues or temporary service outages.

However, origins MAY also implement a timeout after which they delete stored refund state. If a client's refund request arrives after this timeout, the origin will be unable to provide the refund, effectively terminating that credential chain. Clients in this situation will need to request a new credential through the issuance protocol (Section 8) to continue accessing the origin.

## 6. Configuration

ACT Issuers are configured with key material used for issuance and credential verification. Concretely, Issuers run the KeyGen function from [ACT] to produce a private and public key, denoted skI and pkI, respectively.

```
skI, pkI = KeyGen()
```

The Issuer Public Key ID, denoted issuer_key_id, is computed as the SHA-256 hash of the Issuer Public Key, i.e., issuer_key_id = SHA-256(pkI_serialized), where pkI_serialized is the serialized version of pkI as described in Section 4.1 of [ACT].

Protocol messages are encoded using CBOR as specified in Section 4 of [ACT].

### 6.1. Request Context Extension

This Privacy Pass integration binds credentials to application-specific contexts using a request_context parameter. This approach is inspired by the Anonymous Rate-Limited Credentials (ARC) protocol [ARC_PP], which threads a similar context parameter through its cryptographic operations for domain separation.

The CFRG ACT specification [ACT] includes request_context (ctx) as a parameter in its cryptographic functions. Specifically:

1. *IssueRequest*: No changes needed - the client generates a blinded commitment without knowledge of the final context binding.

2. *IssueResponse*: Takes request_context as an input parameter alongside the credit amount c. The issuer determines the appropriate context based on its policy (e.g., derived from TokenChallenge fields) and binds the credential to this context via an additional generator H4. The context scalar is included in the BBS signature as H4 * ctx.

3. *VerifyAndRefund*: The request_context is included in the spend proof and used to verify that the proof is bound to the correct application context. The issuer reconstructs the context from the TokenChallenge and uses it during verification and refund issuance.

4. *Credential Structure*: The Anonymous Credit Token includes the context-bound component within the BBS signature via the H4 generator.

The key insight is that request_context is determined by the issuer (like the credit amount), not by the client. The issuer sets the context during IssueResponse based on the TokenChallenge requirements, and both parties reconstruct it from TokenChallenge fields during spending.

For reference on how request context threading works in practice, see Section 3 of [ARC_PP], which demonstrates the pattern of binding credentials to application-specific contexts through cryptographic commitments.

## 7. Token Challenge Requirements

The ACT protocol uses a modified TokenChallenge structure from the one specified in [AUTHSCHEME]. In particular, the updated TokenChallenge structure is as follows:

```
struct {
    uint16_t token_type = 0xE5AD; /* Type ACT(Ristretto255) */
    opaque issuer_name<1..2^16-1>;
    opaque redemption_context<0..32>;
    opaque origin_info<0..2^16-1>;
    opaque credential_context<0..32>;
} TokenChallenge;
```

Note: The token type value 0xE5AD is provisional pending IANA assignment.

With the exception of credential_context, all fields are exactly as specified in Section 2.1.1 of [AUTHSCHEME]. The credential_context field is defined as follows:

* "credential_context" is a field that is either 0 or 32 bytes, prefixed with a single octet indicating the length (either 0 or 32). If the value is non-empty, it is a 32-byte value generated by the origin that allows the origin to require that clients fetch credentials bound to a specific context. Challenges with credential_context values of invalid lengths MUST be rejected.

Similar to the redemption_context field, the credential_context is used to bind information to the credential. This might be useful, for example, to enforce some expiration on the credential. Origins might do this by constructing credential_context as F(current time window), where F is a pseudorandom function. Semantically, this is equivalent to the Origin asking the Client for a token from a credential that is bound to "current time window."

Origins SHOULD construct credential_context using time-based epochs (e.g., F(current time window) where F is a pseudorandom function) to enforce credential expiration, or per-application binding values to isolate credential usage across different services. The redemption_context SHOULD be set to a fresh random value for each TokenChallenge to ensure token freshness, or left empty when freshness is not required.

In addition to this updated TokenChallenge, the HTTP authentication challenge also SHOULD contain the following additional attribute:

* "cost", which contains a JSON number indicating the amount of credits to spend out of the ACT credential.

Implementation-specific steps: the client should store the Origin-provided input tokenChallenge so that when they receive a new tokenChallenge value, they can check if it has changed and which fields are different. This will inform the client's behavior - for example, if credential_context is being used to enforce an expiration on the credential, then if the credential_context has changed, this can prompt the client to request a new credential.

## 8. Credential Issuance Protocol

Issuers provide an Issuer Private and Public Key, denoted skI and pkI respectively, used to produce credentials as input to the protocol. See Section 6 for how these keys are generated.

Clients provide the following as input to the issuance protocol:

* Issuer Request URL: A URL identifying the location to which issuance requests are sent. This can be a URL derived from the "issuer-request-uri" value in the Issuer's directory resource, or it can be another Client-configured URL. The value of this parameter depends on the Client configuration and deployment model. For example, in the 'Joint Origin and Issuer' deployment model, the Issuer Request URL might correspond to the Client's configured Attester, and the Attester is configured to relay requests to the Issuer.

* Issuer name: An identifier for 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 token_key_id computed as described in Section 6.

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

### 8.1. Client-to-Issuer Request

Given Origin-provided input tokenChallenge and the Issuer Public Key ID issuer_key_id, the Client first creates a credential request message using the IssueRequest function from [ACT] as follows:

```
(clientSecrets, request) = IssueRequest()
```

The Client then creates a TokenRequest structure as follows:

```
struct {
  uint16_t token_type = 0xE5AD; /* Type ACT(Ristretto255) */
  uint8_t truncated_issuer_key_id;
  uint8_t encoded_request[Nrequest];
} TokenRequest;
```

The structure fields are defined as follows:

* "token_type" is a 2-octet integer.

* "truncated_issuer_key_id" is the least significant byte of the issuer_key_id (Section 6) in network byte order (in other words, the last 8 bits of issuer_key_id). This value is truncated so that Issuers cannot use issuer_key_id as a way of uniquely identifying Clients; see Section 10 and referenced information for more details.

* "encoded_request" is the Nrequest-octet request, computed as the serialization of the request value as defined in Section 4.1.1 of [ACT].

The Client then generates an HTTP POST request to send to the Issuer Request URL, with the TokenRequest as the content. The media type for this request is "application/private-credential-request". An example request for the Issuer Request URL "https://issuer.example.net/request" is shown below.

```
POST /request HTTP/1.1
Host: issuer.example.net
Accept: application/private-credential-response
Content-Type: application/private-credential-request
Content-Length: <Length of TokenRequest>

<Bytes containing the TokenRequest>
```

### 8.2. Issuer-to-Client Response

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

* The TokenRequest contains a supported token_type equal to value 0xE5AD.

* The TokenRequest.truncated_token_key_id corresponds to the truncated key ID of an Issuer Public Key, with corresponding secret key skI, owned by the Issuer.

* The TokenRequest.encoded_request is of the correct size (Nrequest).

If any of these conditions is not met, the Issuer MUST return an HTTP 422 (Unprocessable Content) error to the client.

If these conditions are met, the Issuer then tries to deserialize TokenRequest.encoded_request according to Section 4.1.1 of [ACT], yielding request. If this fails, the Issuer MUST return an HTTP 422 (Unprocessable Content) error to the client. Otherwise, if the Issuer is willing to produce a credential for the Client, the Issuer determines both the number of initial credits and the request_context based on its policy (e.g., based on attestation results, payment verification, or TokenChallenge requirements). The request_context binds the credential to the specific application context:

```
request_context = concat(tokenChallenge.issuer_name,
  tokenChallenge.origin_info,
  tokenChallenge.credential_context,
  issuer_key_id)
response = IssueResponse(skI, request, initial_credits, request_context)
```

The Issuer then creates a TokenResponse structured as follows:

```
struct {
   uint8_t encoded_response[Nresponse];
} TokenResponse;
```

The structure fields are defined as follows:

* "encoded_response" is the Nresponse-octet encoded issuance response message, computed as the serialization of response as specified in Section 4.1.2 of [ACT].

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

```
HTTP/1.1 200 OK
Content-Type: application/private-credential-response
Content-Length: <Length of TokenResponse>

<Bytes containing the TokenResponse>
```

### 8.3. Credential Finalization

Upon receipt, the Client handles the response and, if successful, deserializes the content values TokenResponse.encoded_response according to Section 4.1.2 of [ACT] yielding response. If deserialization fails, the Client aborts the protocol. Otherwise, the Client processes the response as follows:

```
credential = VerifyIssuance(pkI, request, response, clientSecrets)
```

The Client then saves the credential structure, associated with the given Issuer Name, to use when producing Token values in response to future token challenges.

## 9. Token Redemption Protocol

The token redemption protocol takes as input TokenChallenge and cost values from [AUTHSCHEME], Section 2.1; the cost is sent as an additional attribute within the HTTP challenge as described in Section 7. Clients use credentials from the issuance protocol in producing tokens bound to the TokenChallenge. The process for producing a token in this way, as well as verifying a resulting token, is described in the following sections.

### 9.1. Token Creation

Given a TokenChallenge value as input, denoted challenge, a cost, denoted cost, and a previously obtained credential that is valid for the Issuer identifier in the challenge, denoted credential, containing at least cost credits, Clients compute a spend request as follows:

```
spend_proof, state = ProveSpend(credential, cost)
```

Each credential instance MUST only ever be used for a single spend request. When the client receives the refunded credential from the server, the client uses that new credential instance for the next spend. If the same credential instance is used more than once, the privacy assumptions of ACT are violated by presenting the same nullifier twice.

The resulting Token value is then constructed as follows:

```
struct {
    uint16_t token_type = 0xE5AD; /* Type ACT(Ristretto255) */
    uint8_t challenge_digest[32];
    uint8_t issuer_key_id[Nid];
    uint8_t encoded_spend_proof[Nspend_proof];
} Token;
```

The structure fields are defined as follows:

* "token_type" is a 2-octet integer, in network byte order, equal to 0xE5AD.

* "challenge_digest" is a 32-octet value containing the hash of the original TokenChallenge, SHA-256(TokenChallenge).

* "issuer_key_id" is a Nid-octet identifier for the Issuer Public Key, computed as defined in Section 6.

* "encoded_spend_proof" is a Nspend_proof-octet encoded spend proof, set to the serialized spend_proof value as specified in Section 4.1.3 of [ACT]. The spend proof contains the nullifier (field 1) and spend amount (field 2), among other cryptographic values.

## 9.2. Token Refund

Upon receiving a Token from the Client, the Origin deserializes the spend_proof according to Section 4.1.3 of [ACT], yielding a SpendProofMsg structure. The Origin then extracts the relevant fields from the spend proof:

```
// Extract fields from SpendProofMsg (see Section 4.1.3 of ACT)
nullifier = spend_proof.k      // Field 1: nullifier (32 bytes)
spend_amount = spend_proof.s   // Field 2: spend amount (32 bytes)
```

The Origin SHOULD verify that the spend_amount matches the requested cost from TokenChallenge to ensure the client is spending the expected amount.

To verify the Token and issue a refund, the Origin constructs the request_context and invokes VerifyAndRefund:

```
request_context = concat(tokenChallenge.issuer_name,
  tokenChallenge.origin_info,
  tokenChallenge.credential_context,
  issuer_key_id)
t = <Origin-configured credits to return to the client>
refund = VerifyAndRefund(skI, spend_proof, t)
```

The parameter t controls how many of the spent credits are returned to the client (0 <= t <= cost). Setting t = 0 consumes the full spend amount; setting t > 0 enables pre-authorization patterns where the origin holds credits temporarily and returns unused ones. The origin determines t based on its own policy (e.g., from the TokenChallenge or application logic).

This function returns the refund serialized according to Section 4.1.4 of [ACT] if the spend proof is valid, and nil otherwise.

As mentioned in Section 2.2.2 of [AUTHSCHEME], Origins MUST implement double-spend prevention that prevents a token with the same nonce from being redeemed twice. With ACT, the Origin MUST check that the nullifier has not previously been seen before calling VerifyAndRefund. It then stores the nullifier for use in future double-spending checks. To reduce the overhead of performing double spend checks, the Origin MAY store and look up the nullifiers corresponding to the associated request_context value.

```
struct {
    uint8_t refund[Nrefund];
} Refund;
```

The Origin sends the refund back to the client encoded as the above Refund struct.

### 9.3. New Credential from Refund

Unlike [ARC], clients must construct a new credential instance based on the Refund response. To do so, clients invoke the ConstructRefundToken function from Section 3.4.4 of [ACT] as follows:

```
credential = ConstructRefundToken(pkI, spend_proof, refund, state)
```

The client then uses this new credential instance for subsequent spend operations.

# 10. Security Considerations

## 10.1. Privacy Properties

Privacy considerations for tokens based on deployment details, such as issuer configuration and issuer selection, are discussed in Section 6.1 of [ARCHITECTURE]. Note that ACT requires a joint Origin and Issuer configuration (where the Issuer and Origin are operated by the same entity) given that tokens produced from credentials are not publicly verifiable.

ACT credentials offer Origin-Client unlinkability, Issuer-Client unlinkability, and redemption context unlinkability, as described in Section 3.3 of [ARCHITECTURE]. The cryptographic security properties of the underlying ACT protocol, including unforgeability and unlinkability guarantees, are analyzed in [ACT].

## 10.2. Double-Spend Prevention

Section 2.2.2 of [AUTHSCHEME] specifies double spending requirements that Origin MUST follow.

For ACT, the double-spend prevention mechanism relies on Origins maintaining state to track nullifiers. As described in Section 9.2, Origins MUST check that a nullifier has not been previously seen before accepting a spend proof. This check is critical for preventing credential reuse attacks.

The nullifier space is large (32 bytes), making random collisions computationally unlikely. However, Origins SHOULD ensure that their nullifier storage is persistent and survives server restarts. If an Origin loses nullifier state (e.g., due to data loss or cache eviction), it becomes vulnerable to replay attacks where an attacker can resubmit previously spent credentials during the credential validity period.

Origins MAY scope nullifier storage by request_context to improve lookup performance and enable efficient storage management. Since nullifiers are only meaningful within the context of a specific credential binding (determined by issuer_name, origin_info, credential_context, and issuer_key_id), Origins can partition nullifier storage accordingly.

When an Origin implements a timeout for refund state (as described in Section 5.5), it MAY also expire the corresponding nullifier after the same timeout period, provided that no future spend attempts with that nullifier are possible.

### 10.3. Concurrency Control and Credential Sharing

The state machine described in Section 5 enforces that each credential instance can only be spent once. This prevents multiple parties from using the same credential concurrently, which is critical for preventing credential sharing attacks.

If an attacker copies a credential and attempts to use it from multiple locations simultaneously, only the first spend operation to reach the Origin will succeed. The Origin's nullifier check will reject all subsequent spends with the same credential as double-spend attempts. This property holds even if the spend operations occur in parallel from different network locations.

Clients MUST implement the state machine correctly and transition credentials to the Spent state immediately upon calling ProveSpend. A client implementation that allows multiple concurrent ProveSpend calls on the same credential would enable double-spend attempts and violate ACT's privacy guarantees by revealing that the same credential is being used multiple times.

### 10.4. Issuer Key Identification

As described in Section 8, the TokenRequest includes only a truncated issuer_key_id (the least significant byte) to prevent Issuers from using the key identifier as a client tracking mechanism.

Issuers MUST prevent truncated key ID collisions among simultaneously active keys. This can be accomplished by generating keys until the truncated key ID does not collide with any existing active key's truncated ID. With a 1-byte (256 value) truncated key space, this is straightforward for reasonable numbers of concurrent keys.

Additionally, Issuers SHOULD limit the total number of simultaneously active keys for privacy reasons, as a large number of active keys can enable partitioning attacks.

### 10.5. Dynamic Revocation and Origin Control

ACT's design allows Origins to terminate credential chains by declining to issue refunds. While this enables the dynamic revocation property described in Section 2, it also gives Origins significant control over client access.

Clients have no cryptographic recourse if an Origin refuses to provide refunds after accepting valid spend proofs. This is an intentional design property that enables Origins to enforce access policies and shed abusive traffic, but it does mean that clients must trust Origins to behave correctly within the protocol.

Clients MAY implement reputation systems or other mechanisms to track Origin behavior and avoid Origins that consistently fail to provide refunds. However, such mechanisms are outside the scope of this specification.

## 11. IANA Considerations

This document updates the "Privacy Pass Token Type" Registry with the following entries.

* Value: 0xE5AD
* Name: ACT (Ristretto255)
* Token Structure: As defined in Section 2.2 of [AUTHSCHEME]
* Token Key Encoding: Serialized as described in Section 6
* TokenChallenge Structure: As defined in Section 2.1 of [AUTHSCHEME]
* Public Verifiability: N
* Public Metadata: N
* Private Metadata: N
* Nk: 0 (not applicable)
* Nid: 32
* Reference: This document
* Notes: None

## 12. Normative References

[ACT] Schlesinger, S. and J. Katz, "Anonymous Credit Tokens", Work in Progress, Internet-Draft, draft-schlesinger-cfrg-act-00, 18 August 2025, <https://datatracker.ietf.org/doc/html/draft-schlesinger-cfrg-act-00>.

[ARC] Yun, C. and C. A. Wood, "Privacy Pass Issuance Protocol for Anonymous Rate-Limited Credentials", Work in Progress, Internet-Draft, draft-yun-privacypass-arc-02, 20 October 2025, <https://datatracker.ietf.org/doc/html/draft-yun-privacypass-arc-02>.

[ARCHITECTURE] Davidson, A., Iyengar, J., and C. A. Wood, "The Privacy Pass Architecture", RFC 9576, DOI 10.17487/RFC9576, June 2024, <https://www.rfc-editor.org/rfc/rfc9576>.

[ARC_PP] Yun, C. and C. A. Wood, "Anonymous Rate-Limited Credentials", Work in Progress, Internet-Draft, draft-yun-cfrg-arc-01, 6 August 2025, <https://datatracker.ietf.org/doc/html/draft-yun-cfrg-arc-01>.

[AUTHSCHEME] Pauly, T., Valdez, S., and C. A. Wood, "The Privacy Pass HTTP Authentication Scheme", RFC 9577, DOI 10.17487/RFC9577, June 2024, <https://www.rfc-editor.org/rfc/rfc9577>.

[ISSUANCE] Celi, S., Davidson, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocols", RFC 9578, DOI 10.17487/RFC9578, June 2024, <https://www.rfc-editor.org/rfc/rfc9578>.

[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/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

## Acknowledgments

The authors would like to thank Cathie Yun, Thibault Meunier, and Chris Wood.

## Authors' Addresses

Samuel Schlesinger
Google
Email: samschlesinger@google.com

Thibault Meunier
Cloudflare Inc.
Email: ot-ietf@thibault.uk