Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: 16 September 2026

W. Guo
L. Xia
J. Li
Y. Li
Huawei Technologies

15 March 2026

Privacy Pass with Token Binding Extension
draft-guo-privacypass-token-binding-02

Abstract

This document provides a token binding extension for the Privacy Pass protocol. This extension allows a Client to cryptographically bind a Privacy Pass token to its own generated one-time public key during the issuance flow, without exposing the public key to the Issuer. Later, when spending the token during the redemption flow, the Client must use the corresponding private key to generate a binding proof that the Origin needs to additionally verify except the token itself, where the proof generation can optionally support channel binding. This proof constrains the legitimate presenter of the token to be only the Client who holds the private key and further constrains the legitimate usage of the token to be only the Client’s specific channel when channel binding is used in the proof generation, and thus can prevent token export and replay attacks. In addition, the token binding extension does not introduce additional cryptographic primitives, and only uses the primitives currently utilized in the issuance protocol to generate the one-time public-private keypair as well as generate and verify the binding proof.

Status of This Memo

Table of Contents

1. Introduction

Privacy Pass provides an anonymous authorization architecture based on privacy-preserving authentication mechanisms [RFC9576], and it consists of the issuance protocol and the redemption protocol. In the issuance protocol [RFC9578], a Client can interact with a token Issuer to obtain Privacy Pass tokens after completing some authentication successfully, where the tokens are generated by the Issuer using the Issuer Private Key. In the redemption protocol [RFC9577], the Client holding such a token can prove to an Origin that it is authorized by the Issuer to access the Origin’s protected resources, without allowing the Origin (even with the Issuer) to link it with the issuance flow.

However, Privacy Pass tokens pertain to bearer tokens, and any party in possession of Privacy Pass tokens can also access to the protected resources. Therefore, attackers can take advantage of this by exporting Privacy Pass tokens from a user’s machines or application connections, presenting them to the application server (the Origin), and impersonating the legitimate user. The token binding extension for the Privacy Pass protocol is to prevent such attacks by cryptographically binding a Privacy Pass token to the user agent (the Client) generated one-time public key and requiring the Client presenting the token to additionally prove possession of the corresponding private key. It can be used not only for basic tokens [RFC9578] but also for advanced tokens (e.g., batched tokens [I-D.draft-ietf-privacypass-batched-tokens-07], tokens with public metadata [I-D.draft-ietf-privacypass-public-metadata-issuance-02]).

In addition, this extension does not introduce additional cryptographic primitives, and only uses the primitives (such as Schnorr NIZKP in Section 2.2 of [RFC9497] and RSA signature in [RFC8017]) currently utilized in the issuance protocol to generate the one-time public-private keypair as well as generate and verify the binding proof. Specifically, we choose the Schnorr NIZKP to achieve token binding for both privately verifiable tokens and publicly verifiable tokens because of the following reasons: 1) the keypair of the Schnorr NIZKP can be generated from a secure seed, which is not supported by the RSA signature; 2) the keypair generation of the Schnorr NIZKP is much faster than that of the RSA signature. Because the keypair is only used one time, a more lightweight proving is to directly present the one-time private key to the Origin, which is so not a zero-knowledge proof, and its verification procedure needs to compute the corresponding one-time public key and verify the bound token.

At a high level, Privacy Pass with token binding extension proceeds as follows. First, a Client generates a one-time public-private keypair probably within a secure hardware module, such as a Trusted Platform Module (TPM) and a Trusted Execution Environment (TEE). Then, the client concatenates the token input and the public key to obtain a bound token input, blinds the bound token input, and generates a TokenRequest similar to Section 5.1 and Section 6.1 of [RFC9578]. After finalizing the issuance protocol successfully, the client will obtain a Privacy Pass bound token. Lastly, when spending the token, the client needs to use the private key to generate a binding proof, which is used to prove possession of the private key corresponding to the public key included in the bound token input. After receiving the token, the public key and the binding proof, the Origin not only needs to verify the token bound with the public key but also needs to verify the binding proof using the bound public key.

Moreover, if there is a channel (e.g., TLS [RFC5246] [RFC8446], HPKE [RFC9180]) between the Client and the Origin, then a secret value exported from the channel can be used as an additional input in the Schnorr NIZK proof generation to support the property of channel binding, which guarantees that the token and the binding proof for one channel that obtained by an attacker cannot be used in another channel.

As a result, if an attacker attempts to export and replay a Privacy Pass bound token, it also needs to use the Client’s private key that corresponds to the token’s bound public key. But this is hard to do if the private key is specially protected in a secure hardware module.

2. Terminology

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

Unless otherwise specified, this document encodes protocol messages in TLS notation from Section 3 of [RFC8446].

This document uses terms "Origin", "Client", "Issuer", "Issuance", "Redemption", "Token challenge", "Token request", "Token response" and "Token" as defined in [RFC9576], and also uses the terms "Issuer key", "Token redemption" as defined in [RFC9577] as well as the terms "Issuer Public Key", "Issuer Private Key" as defined in [RFC9578].

3. Issuance Protocol with Token Binding Extension

This section describes the issuance protocol with token binding extension, which includes two types of issuance protocol: one for privately verifiable tokens and another for publicly verifiable tokens.

3.1. Token Binding for Privately Verifiable Token Issuance

This section uses notation described in Section 4.4 of [RFC9497], including SerializeElement, DeserializeElement, and DeserializeScalar.

The constants Ne and Ns are defined in Section 4.4 of [RFC9497], and the constant Nk is defined in Section 8.2.1 of [RFC9578].

3.1.1. Client-to-Issuer Request

The Client first generates a context as follows:

```
client_context = SetupVOPRFClient("P384-SHA384", pkI)
```

Here, pkI denotes the Issuer Public Key defined in Section 5 of [RFC9578]. "P384-SHA384" is the identifier corresponding to the OPRF(P-384, SHA-384) ciphersuite defined in [RFC9497]. SetupVOPRFClient is defined in Section 3.2 of [RFC9497].

The Client then generates a random 32-byte nonce and creates a token_input as follows:

```
nonce = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0x8001,
                     nonce,
                     challenge_digest,
                     token_key_id)
```

Here, the challenge is an opaque string, and might be provided by the redemption protocol described in Section 2.1 of [RFC9577]. The 2 bytes value 0x8001 represents the token type of VOPRF(P-384, SHA-384) with Token Binding (P-384, SHA-384), where the latter ciphersuite (P-384, SHA-384) will be used for token binding. The token_key_id is a key identifier of the Issuer Public Key pkI, which is computed as described in Section 5.5 of [RFC9578].

The Client generates an ephemeral public-private keypair (pkE, skE) in the group of elliptic curve P-384, where pkE and skE are of Ne-byte length and Ns-byte length, respectively, and the constants Ne = 49 and Ns = 48 are defined in Section 4.4 of [RFC9497]. A RECOMMENDED method for generating ephemeral keypairs is as follows:

```
seed = random(Ns)
ephemeral_seed = SHA384(concat(seed, nonce))
(skE, pkE) = DeriveKeyPair(ephemeral_seed, "PrivacyPassTokenBinding")
```

Here, the long-term seed is probably generated within a secure hardware module and can be used to protect multiple tokens (or a batch of tokens), and the DeriveKeyPair function is defined in Section 3.2.1 of [RFC9497].

After that, the Client creates a bound_token_input and blinds it as follows:

```
binding_pkE = SerializeElement(pkE)
bound_token_input = concat(token_input, binding_pkE)
blind, blinded_element = client_context.Blind(bound_token_input)
blinded_msg = SerializeElement(blinded_element)
```

Here, the Blind function is defined in Section 3.3.1 and Section 3.3.2 of [RFC9497]. If the Blind function fails, the Client aborts the protocol. The Client stores the bound_token_input, blind and blinded_element values locally for use when finalizing the issuance protocol to produce a token (as described in Section 3.1.3).

The Client creates a TokenRequest structured as follows, where structure fields are defined in Section 5.1 of [RFC9578].

```
struct {
    uint16_t token_type = 0x8001;
    uint8_t truncated_token_key_id;
    uint8_t blinded_msg[Ne];
} TokenRequest;
```

The Client then generates an HTTP POST request to send to the Issuer Request URL, with the TokenRequest as the content. Please refer to Section 5.1 of [RFC9578] for an example request.

3.1.2. Issuer-to-Client Response

This section is the same as Section 5.2 of [RFC9578], except that the token type value is 0x8001.

3.1.3. Finalization

Upon receipt, the Client handles the response and, if successful, deserializes the values TokenResponse.evaluate_msg and TokenResponse.evaluate_proof, and obtains evaluated_element and proof as follows.

```
evaluated_element = DeserializeElement(TokenResponse.evaluate_msg)
c = DeserializeScalar(TokenResponse.evaluate_proof[0])
s = DeserializeScalar(TokenResponse.evaluate_proof[1])
proof = (c, s)
```

If deserialization of either value fails, the Client aborts the protocol. Otherwise, the Client processes the response as follows:

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

Here, the Finalize function is defined in Section 3.3.2 of [RFC9497]. If this succeeds, the Client creates a token as follows:

```
struct {
    uint16_t token_type = 0x8001;
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

If the Finalize function fails, the Client aborts the protocol; otherwise it stores the token and its corresponding one-time public key binding_pkE.

3.2. Token Binding for Publicly Verifiable Token Issuance

This section uses the SerializeElement notation defined in Section 4.3 of [RFC9497].

The constant Nk is defined in Section 8.2.2 of [RFC9578].

3.2.1. Client-to-Issuer Request

The Client first generates a random 32-byte nonce and creates a token_input as follows:

```
nonce = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0x8002,
                     nonce,
                     challenge_digest,
                     token_key_id)
```

Here, the challenge is an opaque string, and might be provided by the redemption protocol described in Section 2.1 of [RFC9577]. The 2 bytes value 0x8002 represents the token type of Blind RSA (2048-bit) with Token Binding (P-256, SHA-256), where the latter ciphersuite (P-256, SHA-256) will be used for token binding. The token_key_id is a key identifier of the Issuer Public Key pkI, which is computed as described in Section 6.5 of [RFC9578].

The Client generates an ephemeral public-private keypair (pkE, skE) in the group of elliptic curve P-256, where pkE and skE are of Ne-byte length and Ns-byte length, respectively, and the constants Ne = 33 and Ns = 32 are defined in Section 4.3 of [RFC9497]. A RECOMMENDED method for generating ephemeral keypairs is as follows:

```
seed = random(Ns)
ephemeral_seed = SHA256(concat(seed, nonce))
(skE, pkE) = DeriveKeyPair(ephemeral_seed, "PrivacyPassTokenBinding")
```

Here, the long-term seed is probably generated within a secure hardware module and can be used to protect multiple tokens (or a batch of tokens), and the DeriveKeyPair function is defined in Section 3.2.1 of [RFC9497].

After that, the Client creates a bound_token_input and blinds it as follows:

```
binding_pkE = SerializeElement(pkE)
bound_token_input = concat(token_input, binding_pkE)
blinded_msg, blind_inv = Blind(pkI, PrepareIdentity(bound_token_input))
```

Here, pkI denotes the Issuer Public Key defined in Section 6 of [RFC9578]. The PrepareIdentity and Blind functions are defined in Section 4.1 and Section 4.2 of [RFC9474], respectively. If the Blind function fails, the Client aborts the protocol. The Client stores the bound_token_input and blind_inv values locally for use when finalizing the issuance protocol to produce a token (as described in Section 3.2.3).

The Client creates a TokenRequest structured as follows, where structure fields are defined in Section 6.1 of [RFC9578].

```
    struct {
        uint16_t token_type = 0x8002;
        uint8_t truncated_token_key_id;
        uint8_t blinded_msg[Nk];
    } TokenRequest;
```

The Client then generates an HTTP POST request to send to the Issuer Request URL, with the TokenRequest as the content. Please refer to Section 6.1 of [RFC9578] for an example request.

3.2.2. Issuer-to-Client Response

This section is the same as Section 6.2 of [RFC9578], except that the token type value is 0x8002.

3.2.3. Finalization

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

```
authenticator =
  Finalize(pkI, PrepareIdentity(bound_token_input), blind_sig, blind_inv)
```

Here, the Finalize function is defined in Section 4.4 of [RFC9474]. If this succeeds, the Client creates a token as follows:

```
    struct {
        uint16_t token_type = 0x8002;
        uint8_t nonce[32];
        uint8_t challenge_digest[32];
        uint8_t token_key_id[32];
        uint8_t authenticator[Nk];
    } Token;
```

If the Finalize function fails, the Client aborts the protocol; otherwise it stores the token and its corresponding one-time public key binding_pkE.

4. Redemption Protocol with Token Binding Extension

This section describes the redemption protocol with token binding extension, which includes two types of redemption protocol: one for privately verifiable tokens and another for publicly verifiable tokens.

4.1. Token Binding for Privately Verifiable Token Redemption

This section uses notation described in Section 4.4 of [RFC9497], including Generator, RandomScalar, HashToScalar, SerializeElement and DeserializeElement, and SerializeScalar and DeserializeScalar.

4.1.1. Token Binding Generation

The Client first creates a proof_input as follows:

```
proof_input = concat(token,
                     channel_binding_type,
                     channel_binding_secret)
```

This document defines three variants for channel binding, and the following one-byte values will be used to distinguish between types:

Type	Value
no_channel_binding	0x00
tls_channel_binding	0x01
hpke_channel_binding	0x02

Table 1: Channel Binding Types

The meaning of the above types and the corresponding channel_binding_secret are described as follows:

* The "no_channel_binding" type means that no channel binding is used in the token binding generation, then the channel_binding_secret is a 0-byte value.

* The "tls_channel_binding" type means that a TLS channel binding is used in the token binding generation, then the channel_binding_secret is a 32-byte value. The computations of this secret are defined in [RFC5705] for TLS 1.2 and in Section 7.5 of [RFC8446] for TLS 1.3, respectively, and the corresponding inputs are defined in Section 2 of [RFC9266].

* The "hpke_channel_binding" type means that a HPKE channel binding is used in the token binding generation, then the channel_binding_secret is a 32-byte value. This secret is generated according to Section 5.3 of [RFC9180], and it is defined as follows, where the Context.Export interface is defined in Section 5.3 of [RFC9180].

```
channel_binding_secret = Context.Export("EXPORTER-Channel-Binding", 32)
```

The Client then generates a binding_proof as follows, which is to prove possession of the private key skE corresponding to the previously bound public key pkE:

```
r = RandomScalar()
R = r * Generator()
serialized_R = SerializeElement(R)
challengeTranscript = I2OSP(len(serialized_R), 2) || serialized_R ||
                      I2OSP(len(proof_input), 2) || proof_input ||
                      "Challenge"
c = HashToScalar(challengeTranscript)
s = r - c * skE
binding_proof = (SerializeScalar(c), SerializeScalar(s))
```

Here, the HashToScalar function is based on SHA-384, and the private key skE MAY be recovered from the long-term seed and the Token.nonce as follows.

```
ephemeral_seed = SHA384(concat(seed, Token.nonce))
(skE, _) = DeriveKeyPair(ephemeral_seed, "PrivacyPassTokenBinding")
```

Here, the DeriveKeyPair function defined in Section 3.2.1 of [RFC9497] is used to derive only skE.

The Client creates a TokenBinding structured as follows:

```
struct {
    uint8_t channel_binding_type;
    uint8_t binding_pkE[Ne];
    uint8_t binding_proof[Ns+Ns];
} TokenBinding;
```

If the "no_channel_binding" type is used, then the token binding can also be generated in another more lightweight way without Schnorr NIZK proof generation and verification. Note that the keypair (skE, pkE) is only used once for a Privacy Pass bound token, so the Client can directly present the one-time private key skE to the Origin to prove possession of this key, and thus it is not a zero-knowledge proof. In this case, the first Ns-byte value of the binding_proof

field is set to SerializeScalar(skE) and the second Ns-byte value of this field is set to all zero bytes, and the binding_pkE field is set to zero-length string since the one-time public key pkE can be recovered from the one-time private key skE.

4.1.2. Sending Token and Token Binding

When used for Client authorization, the "PrivateToken" authentication scheme defines one parameter, "token", which contains the base64url-encoded Token structure. This document defines a new parameter, "token_binding", which contains the base64url-encoded TokenBinding structure. This document follows the default padding behavior described in Section 3.2 of [RFC4648], so the base64url value MUST include padding. The Client presents the Token structure and the TokenBinding structure to the Origin in a new HTTP request using the Authorization header field as follows:

```
Authorization: PrivateToken token="abc...", token_binding="def..."
```

4.1.3. Token and Token Binding Verification

Upon receipt, the Origin needs to verify both the token and the token binding, and if any one of verifications fails, this authorization request will be rejected.

For the token type of VOPRF(P-384, SHA-384) with Token Binding (P-384, SHA-384) (0x8001), verifying a token requires creating a VOPRF context using the Issuer Private Key, evaluating the bound token input, and comparing the result against the token authenticator value.

```
server_context = SetupVOPRFServer("P384-SHA384", skI)
token_authenticator_input =
  concat(Token.token_type,
         Token.nonce,
         Token.challenge_digest,
         Token.token_key_id)
bound_token_authenticator_input =
  concat(token_authenticator_input, TokenBinding.binding_pkE)
bound_token_authenticator =
  server_context.Evaluate(bound_token_authenticator_input)
valid = (bound_token_authenticator == Token.authenticator)
```

Here, the SetupVOPRFServer and Evaluate functions are defined in Section 3.3.1 and Section 3.2 of [RFC9497], respectively.

Verifying a token binding requires checking that TokenBinding.binding_proof is a valid Schnorr NIZK proof using the TokenBinding.binding_pkE, where the HashToScalar function is based on SHA-384.

```
proof_verification_input =
  concat(token,
         TokenBinding.channel_binding_type,
         expected_channel_binding_secret)
pkE = DeserializeElement(TokenBinding.binding_pkE)
c = DeserializeScalar(TokenBinding.binding_proof[0])
s = DeserializeScalar(TokenBinding.binding_proof[1])
R = (s * Generator()) + (c * pkE)
serialized_R = SerializeElement(R)
challengeTranscript =
  I2OSP(len(serialized_R), 2) || serialized_R ||
  I2OSP(len(proof_verification_input), 2) || proof_verification_input ||
  "Challenge"
expectedC = HashToScalar(challengeTranscript)
valid = (expectedC == c)
```

If the "no_channel_binding" type is used and the token binding is generated without using Schnorr NIZKP, then a token binding is implicitly verified in the above token verification, where the binding_pkE is computed as follows.

```
skE = DeserializeScalar(TokenBinding.binding_proof[0])
pkE = skE * Generator()
binding_pkE = SerializeElement(pkE)
```

4.2. Token Binding for Publicly Verifiable Token Redemption

This section uses notation described in Section 4.3 of [RFC9497], including Generator, RandomScalar, HashToScalar, SerializeElement and DeserializeElement, and SerializeScalar and DeserializeScalar.

4.2.1. Token Binding Generation

The Client first creates a proof_input as follows:

```
proof_input = concat(token,
                     channel_binding_type,
                     channel_binding_secret)
```

Here, the channel_binding_type and the channel_binding_secret fields are defined in Section 4.1.1.

The Client then generates a binding_proof as follows, which is to prove possession of the private key skE corresponding to the previously bound public key pkE:

```
r = RandomScalar()
R = r * Generator()
serialized_R = SerializeElement(R)
challengeTranscript = I2OSP(len(serialized_R), 2) || serialized_R ||
                      I2OSP(len(proof_input), 2) || proof_input ||
                      "Challenge"
c = HashToScalar(challengeTranscript)
s = r - c * skE
binding_proof = (SerializeScalar(c), SerializeScalar(s))
```

Here, the HashToScalar function is based on SHA-256, and the private key skE MAY be recovered from the long-term seed and the Token.nonce as follows.

```
ephemeral_seed = SHA256(concat(seed, Token.nonce))
(skE, _) = DeriveKeyPair(ephemeral_seed, "PrivacyPassTokenBinding")
```

Here, the DeriveKeyPair function defined in Section 3.2.1 of [RFC9497] is used to derive only skE.

The Client creates a TokenBinding structured as follows:

```
struct {
    uint8_t channel_binding_type;
    uint8_t binding_pkE[Ne];
    uint8_t binding_proof[Ns+Ns];
} TokenBinding;
```

If the "no_channel_binding" type is used, then the token binding can also be generated by using the same lightweight way as in Section 4.1.1, thus is omitted here.

4.2.2. Sending Token and Token Binding

As an example, the Client presents the Token structure and the TokenBinding structure to the Origin in a new HTTP request using the Authorization header field as follows:

```
Authorization: PrivateToken token="abc...", token_binding="def..."
```

4.2.3. Token and Token Binding Verification

Upon receipt, the Origin needs to verify both the token and the token binding, and if any one of verifications fails, this authorization request will be rejected.

For the token type of Blind RSA (2048-bit) with Token Binding (P-256, SHA-256) (0x8002), verifying a token requires checking that Token.authenticator is a valid RSA signature over the bound token input using the Issuer Public Key.

```
token_authenticator_input =
  concat(Token.token_type,
         Token.nonce,
         Token.challenge_digest,
         Token.token_key_id)
bound_token_authenticator_input =
  concat(token_authenticator_input, TokenBinding.binding_pkE)
valid = RSASSA-PSS-VERIFY(pkI,
                          bound_token_authenticator_input,
                          Token.authenticator)
```

Here, the RSASSA-PSS-VERIFY function is defined in Section 8.1.2 of [RFC8017], using SHA-384 as the hash function, MGF1 with SHA-384 as the Probabilistic Signature Scheme (PSS) mask generation function (MGF), and a 48-byte salt length (sLen).

Verifying a token binding requires checking that TokenBinding.binding_proof is a valid Schnorr NIZK proof using the TokenBinding.binding_pkE, where the HashToScalar function is based on SHA-256.

```
proof_verification_input =
  concat(token,
         TokenBinding.channel_binding_type,
         expected_channel_binding_secret)
pkE = DeserializeElement(TokenBinding.binding_pkE)
c = DeserializeScalar(TokenBinding.binding_proof[0])
s = DeserializeScalar(TokenBinding.binding_proof[1])
R = (s * Generator()) + (c * pkE)
serialized_R = SerializeElement(R)
challengeTranscript =
  I2OSP(len(serialized_R), 2) || serialized_R ||
  I2OSP(len(proof_verification_input), 2) || proof_verification_input ||
  "Challenge"
expectedC = HashToScalar(challengeTranscript)
valid = (expectedC == c)
```

If the "no_channel_binding" type is used and the token binding is generated without using Schnorr NIZKP, then a token binding is implicitly verified in the above token verification, where the binding_pkE is computed as follows.

```
skE = DeserializeScalar(TokenBinding.binding_proof[0])
pkE = skE * Generator()
binding_pkE = SerializeElement(pkE)
```

5. IANA Considerations

This document defines two new token types "0x8001" and "0x8002" with the following contents, and requests that IANA add the two values to the "Privacy Pass Token Types" Registry defined in Section 6.2 of [RFC9577].

Note that Token Binding, Ne, Ns are newly added fields, where the latter two MUST be provided if Token Binding is Y, or not applicable (N/A) otherwise.

5.1. Token Type VOPRF(P-384, SHA-384) with Token Binding (P-384, SHA-384)

Value: 0x8001

Name: VOPRF(P-384, SHA-384) with Token Binding (P-384, SHA-384)

Token Structure: As defined in Section 3.1.

Token Key Encoding: Serialized using SerializeElement (Section 2.1 of [RFC9497]).

TokenChallenge Structure: As defined in Section 2.1 of [RFC9577].

Publicly Verifiable: N

Public Metadata: N

Private Metadata: N

Nk: 48

Nid: 32

Token Binding: Y

Ne: 49

Ns: 48

Change controller: IETF

Reference: This document, Section 3.1

Notes: None

5.2. Token Type Blind RSA (2048-bit) with Token Binding (P-256, SHA-256)

Value: 0x8002

Name: Blind RSA (2048-bit) with Token Binding (P-256, SHA-256)

Token Structure: As defined in Section 3.2.

Token Key Encoding: Serialized as a DER-encoded SubjectPublicKeyInfo (SPKI) object using the RSASSA-PSS OID [RFC5756].

TokenChallenge Structure: As defined in Section 2.1 of [RFC9577].

Publicly Verifiable: Y

Public Metadata: N

Private Metadata: N

Nk: 256

Nid: 32

Token Binding: Y

Ne: 33

Ns: 32

Change controller: IETF

Reference: This document, Section 3.2

Notes: None

6. References

6.1. Normative References

[RFC9576] 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>.

[RFC9578] 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>.

[RFC9577] 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>.

[RFC9497] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups", RFC 9497, DOI 10.17487/RFC9497, December 2023, <https://www.rfc-editor.org/rfc/rfc9497>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/rfc/rfc5246>.

[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/rfc/rfc8446>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

[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>.

[RFC9474] Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", RFC 9474, DOI 10.17487/RFC9474, October 2023, <https://www.rfc-editor.org/rfc/rfc9474>.

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <https://www.rfc-editor.org/rfc/rfc5705>.

[RFC9266] Whited, S., "Channel Bindings for TLS 1.3", RFC 9266, DOI 10.17487/RFC9266, July 2022, <https://www.rfc-editor.org/rfc/rfc9266>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters", RFC 5756, DOI 10.17487/RFC5756, January 2010, <https://www.rfc-editor.org/rfc/rfc5756>.

6.2. Informative References

[I-D.draft-ietf-privacypass-batched-tokens-07] Robert, R., Wood, C. A., and T. Meunier, "Batched Token Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-batched-tokens-07, 25 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-batched-tokens-07>.

[I-D.draft-ietf-privacypass-public-metadata-issuance-02] Hendrickson, S. and C. A. Wood, "Privacy Pass Issuance Protocols with Public Metadata", Work in Progress, Internet-Draft, draft-ietf-privacypass-public-metadata-issuance-02, 27 May 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-public-metadata-issuance-02>.

Authors' Addresses

Wei Guo
Huawei Technologies
Email: guowei90@huawei.com

Liang Xia
Huawei Technologies
Email: frank.xialiang@huawei.com

Ji Li
Huawei Technologies
Email: liji100@huawei.com

Yong Li
Huawei Technologies
Email: Yong.Li1@huawei.com