Workgroup: Network Working Group
Internet-Draft:
draft-ietf-privacypass-auth-scheme-15
Published: 23 October 2023
Intended Status: Standards Track
Expires: 25 April 2024
Authors: T. Pauly Apple Inc.
S. Valdez Google LLC
C. A. Wood Cloudflare

The Privacy Pass HTTP Authentication Scheme

Abstract

This document defines an HTTP authentication scheme for Privacy Pass, a privacy-preserving authentication mechanism used for authorization. The authentication scheme in this document can be used by clients to redeem Privacy Pass tokens with an origin. It can also be used by origins to challenge clients to present Privacy Pass tokens.

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 25 April 2024.

Table of Contents

1. Introduction
  1.1. Terminology
2. HTTP Authentication Scheme
  2.1. Token Challenge
    2.1.1. Token Challenge Structure
    2.1.2. Sending Token Challenges
    2.1.3. Processing Token Challenges
    2.1.4. Token Caching
  2.2. Token Redemption
    2.2.1. Token Structure
    2.2.2. Sending Tokens
    2.2.3. Token Verification
3. Client Behavior
  3.1. Choosing to Redeem Tokens
  3.2. Choosing Between Multiple Challenges
4. Origin Behavior
  4.1. Greasing
5. Security Considerations
  5.1. Randomness Requirements
  5.2. Replay Attacks
  5.3. Reflection Attacks
  5.4. Token Exhaustion Attacks
  5.5. Timing Correlation Attacks
  5.6. Cross-Context Linkability Attacks
6. IANA Considerations
  6.1. Authentication Scheme
  6.2. Token Type Registry
    6.2.1. Reserved Values
7. References
  7.1. Normative References
  7.2. Informative References
Appendix A. Test Vectors
  A.1. Challenge and Redemption Structure Test Vectors
  A.2. HTTP Header Test Vectors
Authors' Addresses

1. Introduction

Privacy Pass tokens are unlinkable authenticators that can be used to anonymously authorize a client (see [ARCHITECTURE]). Tokens are generated by token issuers, on the basis of authentication, attestation, or some previous action such as solving a CAPTCHA. A client possessing such a token is able to prove that it was able to

get a token issued, without allowing the relying party redeeming the client's token (the origin) to link it with the issuance flow.

Different types of authenticators, using different token issuance protocols, can be used as Privacy Pass tokens.

This document defines a common HTTP authentication scheme ([RFC9110], Section 11), PrivateToken, that allows clients to redeem various kinds of Privacy Pass tokens.

Clients and relying parties (origins) interact using this scheme to perform the token challenge and token redemption flow. In particular, origins challenge clients for a token with an HTTP Authentication challenge (using the WWW-Authenticate response header field). Clients can then react to that challenge by issuing a new request with a corresponding token (using the Authorization request header field). Clients generate tokens that match the origin's token challenge by running the token issuance protocol [ISSUANCE]. The act of presenting a token in an Authorization request header field is referred to as token redemption. This interaction between client and origin is shown below.

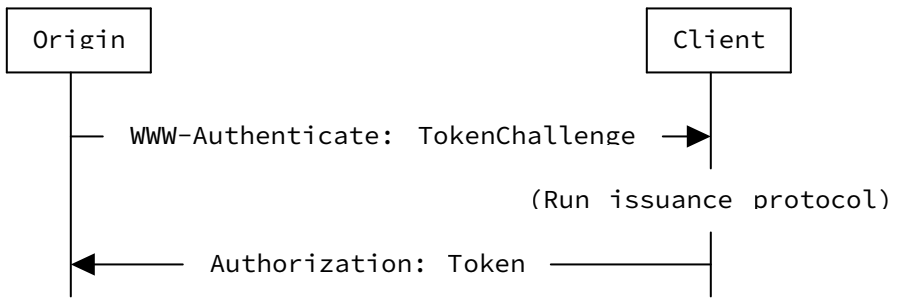

Figure 1: Challenge and redemption protocol flow

In addition to working with different token issuance protocols, this scheme optionally supports use of tokens that are associated with origin-chosen contexts and specific origin names. Relying parties that request and redeem tokens can choose a specific kind of token, as appropriate for its use case. These options allow for different deployment models to prevent double-spending, and allow for both interactive (online challenges) and non-interactive (pre-fetched) tokens.

1.1. 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 [TLS13], Section 3.

This document uses the terms "Client", "Origin", "Issuer", "Issuance Protocol", and "Token" as defined in [ARCHITECTURE]. It additionally uses the following terms in more specific ways:

- Issuer key: Keying material that can be used with an issuance protocol to create a signed token.

- Token challenge: A request for tokens sent from an origin to a client, using the "WWW-Authenticate" HTTP header field. This challenge identifies a specific token issuer and issuance protocol. Token challenges optionally include one or both of: a redemption context (see Section 2.1.1.2), and a list of associated origins. These optional values are then be bound to the token that is issued.

- Token redemption: An action by which a client presents a token to an origin in an HTTP request, using the "Authorization" HTTP header field.

2. HTTP Authentication Scheme

Token redemption is performed using HTTP Authentication ([RFC9110], Section 11), with the scheme "PrivateToken". Origins challenge clients to present a token from a specific issuer (Section 2.1). Once a client has received a token from that issuer, or already has a valid token available, it presents the token to the origin (Section 2.2). The process of presenting a token as authentication to an origin is also referred to as "spending" a token.

In order to prevent linkability across different transactions, clients will often present a particular "PrivateToken" only once. Origins can link multiple transactions to the same client if that client spends the same token value more than once. As such, origins ought to expect at most one unique token value, carried in one request, for each challenge.

The rest of this section describes the token challenge and redemption interactions in more detail.

2.1. Token Challenge

Origins send a token challenge to clients in an "WWW-Authenticate" header field with the "PrivateToken" scheme. This authentication

scheme has two mandatory parameters: one containing a token challenge and another containing the token-key used for producing (and verifying) a corresponding token.

Origins that support the "PrivateToken" authentication scheme need to handle the following tasks in constructing the WWW-Authenticate header field:

1. Select which issuer to use, and configure the issuer name and token-key to include in WWW-Authenticate token challenges. The issuer name is included in the token challenge, and the issuer token-key is used to populate the WWW-Authenticate header parameter.

2. Determine a redemption context construction to include in the token challenge, as discussed in Section 2.1.1.2.

3. Select the origin information to include in the token challenge. This can be empty to allow fully cross-origin tokens, a single origin name that matches the origin itself for per-origin tokens, or a list of origin names containing the origin itself. See Section 3.4 of [ARCHITECTURE] for more information about the difference between cross-origin and per-origin tokens.

Once these decisions are made, origins construct the WWW-Authenticate header by first constructing the token challenge as described in Section 2.1.1. Origins send challenges as described in Section 2.1.2, and clients process them as described in Section 2.1.3 and Section 2.1.4.

2.1.1. Token Challenge Structure

This document defines the default challenge structure that can be used across token types, although future token types MAY extend or modify the structure of the challenge; see Section 6.2 for the registry information which establishes and defines the relationship between "token_type" and the contents of the TokenChallenge message.

All token challenges MUST begin with a 2-octet integer that defines the token type, in network byte order. This type indicates the issuance protocol used to generate the token and determines the structure and semantics of the rest of the structure. Values are registered in an IANA registry, Section 6.2. Client MUST ignore challenges with token types they do not support.

Even when a given token type uses the default challenge structure, the requirements on the presence or interpretation of the fields can differ across token types. For example, some token types might

require that "origin_info" is non-empty, while others allow it to be empty.

The default TokenChallenge message has the following structure:

```
struct {
    uint16_t token_type;
    opaque issuer_name<1..2^16-1>;
    opaque redemption_context<0..32>;
    opaque origin_info<0..2^16-1>;
} TokenChallenge;
```

The structure fields are defined as follows:

* "token_type" is a 2-octet integer, in network byte order, as described above.

* "issuer_name" is an ASCII string that identifies the issuer using the format of a server name defined in Section 2.1.1.1. This name identifies the issuer that is allowed to issue tokens that can be redeemed by this origin. The field that stores this string in the challenge is prefixed with a 2-octet integer indicating the length, in network byte order.

* "redemption_context" is a field that is either 0 or 32 bytes, prefixed with a single octet indicating the length (either 0 or 32). If value is non-empty, it is a 32-byte value generated by the origin that allows the origin to require that clients fetch tokens bound to a specific context, as opposed to reusing tokens that were fetched for other contexts. See Section 2.1.1.2 for example contexts that might be useful in practice. Challenges with redemption_context values of invalid lengths MUST be ignored.

* "origin_info" is an ASCII string that is either empty, or contains one or more origin names that allow a token to be scoped to a specific set of origins. Each origin name uses the format of a server name defined in Section 2.1.1.1. The string is prefixed with a 2-octet integer indicating the length, in network byte order. If empty, any non-origin-specific token can be redeemed. If the string contains multiple origin names, they are delimited with commas "," without any whitespace. If this field is not empty, the Origin MUST include its own name as one of the names in the list.

If "origin_info" contains multiple origin names, this means the challenge is valid for any of the origins in the list, including the origin which issued the challenge (which must always be present in the list if it is non-empty; see Section 2.1.3). This can be useful in settings where clients pre-fetch and cache tokens for a

particular challenge -- including the "origin_info" field -- and then later redeem these tokens with one of the origins in the list. See Section 2.1.4 for more discussion about token caching.

2.1.1.1. Server Name Encoding

Server names contained in a token challenge are ASCII strings that contain a hostname and optional port, where the port is implied to be "443" if missing. The names use the format of the authority portion of a URI as defined in Section 3.2 of [URI]. The names MUST NOT include a "userinfo" portion of an authority. For example, a valid server name might be "issuer.example.com" or "issuer.example.com:8443", but not "issuer@example.com".

2.1.1.2. Redemption Context Construction

The TokenChallenge redemption context allows the origin to determine the context in which a given token can be redeemed. This value can be a unique per-request nonce, constructed from 32 freshly generated random bytes. It can also represent state or properties of the client session. Some example properties and methods for constructing the corresponding context are below. This list is not exhaustive.

- Context bound to a given time window: Construct redemption context as F(current time window), where F is a pseudorandom function.

- Context bound to a client network: Construct redemption context as F(client ASN), where F is a pseudorandom function.

- Context bound to a given time window and client network: Construct redemption context as F(current time window, client ASN), where F is a pseudorandom function.

Preventing double spending on tokens requires the origin to keep state associated with the redemption context. An empty redemption context is not bound to any property of the client request, so state to prevent double spending needs to be stored and shared across all origin servers that can accept tokens until token-key expiration or rotation. For a non-empty redemption context, the double spend state only needs to be stored across the set of origin servers that will accept tokens with that redemption context.

Origins that share redemption contexts, i.e., by using the same redemption context, choosing the same issuer, and providing the same origin_info field in the TokenChallenge, must necessarily share state required to enforce double spend prevention. Origins should consider the operational complexity of this shared state before choosing to share redemption contexts. Failure to successfully synchronize this state and use it for double spend prevention can

allow Clients to redeem tokens to one Origin that were issued after an interaction with another Origin that shares the context.

2.1.2. Sending Token Challenges

When used in an authentication challenge, the "PrivateToken" scheme uses the following parameters:

* "challenge", which contains a base64url-encoded [RFC4648] TokenChallenge value. This document follows the default padding behavior described in Section 3.2 of [RFC4648], so the base64url value MUST include padding. As an Authentication Parameter (auth-param from [RFC9110], Section 11.2), the value can be either a token or a quoted-string, and might be required to be a quoted-string if the base64url string includes "=" characters. This parameter is required for all challenges.

* "token-key", which contains a base64url encoding of the public key for use with the issuance protocol indicated by the challenge. See [ISSUANCE] for more information about how this public key is used by the issuance protocols in that specification. The encoding of the public key is determined by the token type; see Section 6.2. As with "challenge", the base64url value MUST include padding. As an Authentication Parameter (auth-param from [RFC9110], Section 11.2), the value can be either a token or a quoted-string, and might be required to be a quoted-string if the base64url string includes "=" characters. This parameter MAY be omitted in deployments where clients are able to retrieve the issuer key using an out-of-band mechanism.

* "max-age", an optional parameter that consists of the number of seconds for which the challenge will be accepted by the origin.

The header field MAY also include the standard "realm" parameter, if desired. Issuance protocols MAY define other parameters, some of which might be required. Clients MUST ignore parameters in challenges that are not defined for the issuance protocol corresponding to the token type in the challenge.

As an example, the WWW-Authenticate header field could look like this:

```
WWW-Authenticate:
  PrivateToken challenge="abc...", token-key="123..."
```

2.1.2.1. Sending Multiple Token Challenges

It is possible for the WWW-Authenticate header field to include multiple challenges ([RFC9110], Section 11.6.1). This allows the

origin to indicate support for different token types, issuers, or to include multiple redemption contexts. For example, the WWW-Authenticate header field could look like this:

```
WWW-Authenticate:
  PrivateToken challenge="abc...", token-key="123...",
  PrivateToken challenge="def...", token-key="234..."
```

Origins should only include challenges for different types of issuance protocols with functionally equivalent properties. For instance, both issuance protocols in [ISSUANCE] have the same functional properties, albeit with different mechanisms for verifying the resulting tokens during redemption. Since clients are free to choose which challenge they want to consume when presented with options, mixing multiple challenges with different functional properties for one use case is nonsensical. If the origin has a preference for one challenge over another (for example, if one uses a token type that is faster to verify), it can sort it to be first in the list of challenges as a hint to the client.

2.1.3. Processing Token Challenges

Upon receipt of a challenge, a client validates the TokenChallenge structure before taking any action, such as fetching a new token or redeeming a token in a new request. Validation requirements are as follows:

- The token_type is recognized and supported by the client;
- The TokenChallenge structure is well-formed; and
- If the origin_info field is non-empty, the name of the origin that issued the authentication challenge is included in the list of origin names. Comparison of the origin name that issued the authentication challenge against elements in the origin_info list is done via case-insensitive equality checks.

If validation fails, the client MUST NOT fetch or redeem a token based on the challenge. Clients MAY have further restrictions and requirements around validating when a challenge is considered acceptable or valid. For example, clients can choose to ignore challenges that list origin names for which the current connection is not authoritative (according to the TLS certificate).

Caching and pre-fetching of tokens is discussed in Section 2.1.4.

2.1.4. Token Caching

Clients can generate multiple tokens from a single TokenChallenge, and cache them for future use. This improves privacy by separating

the time of token issuance from the time of token redemption, and also allows clients to avoid any overhead of receiving new tokens via the issuance protocol.

Cached tokens can only be redeemed when they match all of the fields in the TokenChallenge: token_type, issuer_name, redemption_context, and origin_info. Clients ought to store cached tokens based on all of these fields, to avoid trying to redeem a token that does not match. Note that each token has a unique client nonce, which is sent in token redemption (Section 2.2).

If a client fetches a batch of multiple tokens for future use that are bound to a specific redemption context (the redemption_context in the TokenChallenge was not empty), clients SHOULD discard these tokens upon flushing state such as HTTP cookies [COOKIES], or if there is a network change and the client does not have any origin-specific state like HTTP cookies. Using these tokens in a context that otherwise would not be linkable to the original context could allow the origin to recognize a client.

2.2. Token Redemption

The output of the issuance protocol is a token that corresponds to the origin's challenge (see Section 2.1).

2.2.1. Token Structure

A token is a structure that begins with a two-octet field that indicates a token type, which MUST match the token_type in the TokenChallenge structure. This value determines the structure and semantics of the rest of token structure.

This document defines the default token structure that can be used across token types, although future token types MAY extend or modify the structure of the token; see Section 6.2 for the registry information which establishes and defines the relationship between "token_type" and the contents of the Token structure.

The default Token message has the following structure:

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

The structure fields are defined as follows:

- "token_type" is a 2-octet integer, in network byte order, as described above.

- "nonce" is a 32-octet value containing a client-generated random nonce.

- "challenge_digest" is a 32-octet value containing the hash of the original TokenChallenge, SHA-256(TokenChallenge), where SHA-256 is as defined in [SHS]. Changing the hash function to something other than SHA-256 would require defining a new token type and token structure (since the contents of challenge_digest would be computed differently), which can be done in a future specification.

- "token_key_id" is a Nid-octet identifier for the token authentication key. The value of this field is defined by the token_type and corresponding issuance protocol.

- "authenticator" is a Nk-octet authenticator that is cryptographically bound to the preceding fields in the token; see Section 2.2.3 for more information about how this field is used in verifying a token. The token_type and corresponding issuance protocol determine the value of the authenticator field and how it is computed. The value of constant Nk depends on token_type, as defined in Section 6.2.

The authenticator value in the Token structure is computed over the token_type, nonce, challenge_digest, and token_key_id fields. A token is considered a valid if token verification using succeeds; see Section 2.2.3 for details about verifying the token and its authenticator value.

2.2.2. Sending Tokens

When used for client authorization, the "PrivateToken" authentication scheme defines one parameter, "token", which contains the base64url-encoded Token struct. As with the challenge parameters (Section 2.1), the base64url value MUST include padding. As an Authentication Parameter (auth-param from [RFC9110], Section 11.2), the value can be either a token or a quoted-string, and might be required to be a quoted-string if the base64url string includes "=" characters. All unknown or unsupported parameters to "PrivateToken" authentication credentials MUST be ignored.

Clients present this Token structure to origins in a new HTTP request using the Authorization header field as follows:

```
Authorization: PrivateToken token="abc..."
```

For context-bound tokens, origins store or reconstruct the contexts of previous TokenChallenge structures in order to validate the token. A TokenChallenge can be bound to a specific TLS session with a client, but origins can also accept tokens for valid challenges in new sessions. Origins SHOULD implement some form of double-spend prevention that prevents a token with the same nonce from being redeemed twice. Double-spend prevention ensures that clients cannot replay tokens for previous challenges. See Section 5.2 for more information about replay attacks. For context-bound tokens, this double-spend prevention can require no state or minimal state, since the context can be used to verify token uniqueness.

2.2.3. Token Verification

A token consists of some input cryptographically bound to an authenticator value, such as a digital signature. Verifying a token consists of checking that the authenticator value is correct.

The authenticator value is as computed when running and finalizing the issuance protocol corresponding to the token type with the following value as the input:

```
struct {
    uint16_t token_type;
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[Nid];
} AuthenticatorInput;
```

The value of these fields are as described in Section 2.2. The cryptographic verification check depends on the token type; see Section 5.4 of [ISSUANCE] and Section 6.4 of [ISSUANCE] for verification instructions for the issuance protocols described in [ISSUANCE]. As such, the security properties of the token, e.g., the probability that one can forge an authenticator value without invoking the issuance protocol, depend on the cryptographic algorithm used by the issuance protocol as determined by the token type.

3. Client Behavior

When a client receives one or more token challenges in response to a request, the client has a set of choices to make:

- Whether or not to redeem a token via a new request to the origin.
- Whether to redeem a previously issued and cached token, or redeem a token freshly issued from the issuance protocol.

* If multiple challenges were sent, which challenge to use for redeeming a token on a subsequent request.

The approach to these choices depends on the use case of the application, as well as the deployment model (see Section 4 of [ARCHITECTURE] for discussion of the different deployment models).

3.1. Choosing to Redeem Tokens

Some applications of tokens might require clients to always present a token as authentication in order to successfully make requests. For example, a restricted service that wants to only allow access to valid users, but do so without learning specific user credential information, could use tokens that are based on attesting user credentials. In these kinds of use cases, clients will need to always redeem a token in order to successfully make a request.

Many other use cases for Privacy Pass tokens involve open services that must work with any client, including those that either cannot redeem tokens, or can only sometimes redeem tokens. For example, a service can use tokens as a way to reduce the incidence of presenting CAPTCHAs to users. In such use cases, services will regularly encounter clients that cannot redeem a token or choose not to. In order to mitigate the risk of these services relying on always receiving tokens, clients that are capable of redeeming tokens can ignore token challenges (and instead behave as if they were a client that either doesn't support redeeming tokens or is unable to generate a new token, by not sending a new request that contains a token to redeem) with some non-trivial probability. See Section 5.1 of [ARCHITECTURE] for further considerations on avoiding discriminatory behavior across clients when using Privacy Pass tokens.

Clients might also choose to not redeem tokens in subsequent requests when the token challenges indicate erroneous or malicious behavior on the part of the challenging origin. For example, if a client's ability to generate tokens via an attester and issuer is limited to a certain rate, a malicious origin could send an excessive number of token challenges with unique redemption contexts in order to cause the client to exhaust its ability to generate new tokens, or to overwhelm issuance servers. The limits here will vary based on the specific deployment, but clients SHOULD have some implementation-specific policy to minimize the number of tokens that can be retrieved by origins.

3.2. Choosing Between Multiple Challenges

A single response from an origin can include multiple token challenges. For example, a set of challenges could include different

token types and issuers, to allow clients to choose a preferred issuer or type.

If clients choose to respond, clients should satisfy exactly one of the challenges presented. The choice of which challenge to use for redeeming tokens is up to client policy. This can involve which token types are supported or preferred, which issuers are supported or preferred, or whether or not the client is able to use cached tokens based on the redemption context or origin information in the challenge. See Section 2.1.4 for more discussion on token caching. Regardless of how the choice is made, it SHOULD be done in a consistent manner to ensure that the choice does not reveal information about the specific client; see Section 6.2 of [ARCHITECTURE] for more details on the privacy implications of issuance consistency.

4. Origin Behavior

Origins choose what token challenges to send to clients, which will vary depending on the use case and deployment model. The origin chooses which token types, issuers, redemption contexts, and origin info to include in challenges. If an origin sends multiple challenges, each challenge SHOULD be equivalent in terms of acceptability for token redemption, since clients are free to choose to generate tokens based on any of the challenges.

Origins ought to consider the time involved in token issuance. Particularly, a challenge that includes a unique redemption context will prevent a client from using cached tokens, and thus can add more delay before the client is able to redeem a token.

Origins SHOULD minimize the number of challenges sent to a particular client context (referred to as the "redemption context" in Section 3.3 of [ARCHITECTURE]), to avoid overwhelming clients and issuers with token requests that might cause clients to hit rate limits.

4.1. Greasing

In order to prevent clients becoming incompatible with new token challenges, origins SHOULD include random token types, from the Reserved list of "greased" types (defined in Section 6.2), with some non-trivial probability.

Additionally, for deployments where tokens are not required (such as when tokens are used as a way to avoiding showing CAPTCHAs), origins SHOULD randomly
choose to not challenge clients for tokens with some non-trivial probability. This helps origins ensure that their behavior for

handling clients that cannot redeem tokens is maintained and exercised consistently.

5. Security Considerations

This section contains security considerations for the PrivateToken authentication scheme described in this document.

5.1. Randomness Requirements

All random values in the challenge and token MUST be generated using a cryptographically secure source of randomness ([RFC4086]).

5.2. Replay Attacks

Applications SHOULD constrain tokens to a single origin unless the use case can accommodate replay attacks. Replaying tokens is not necessarily a security or privacy problem. As an example, it is reasonable for clients to replay tokens in contexts where token redemption does not induce side effects and in which client requests are already linkable. One possible setting where this applies is where tokens are sent as part of 0-RTT data.

If successful token redemption produces side effects, origins SHOULD implement an anti-replay mechanism to mitigate the harm of such replays. See [TLS13], Section 8 and [RFC9001], Section 9.2 for details about anti-replay mechanisms, as well as [RFC8470], Section 3 for discussion about safety considerations for 0-RTT HTTP data.

5.3. Reflection Attacks

The security properties of token challenges vary depending on whether the challenge contains a redemption context or not, as well as whether the challenge is per-origin or not. For example, cross-origin tokens with empty contexts can be reflected from one party by another, as shown below.

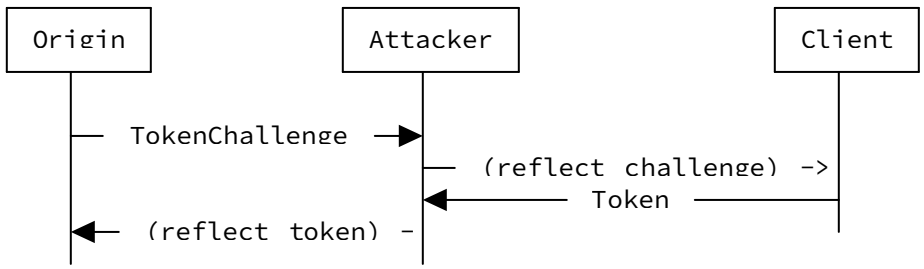

Figure 2: Replay attack example

5.4. Token Exhaustion Attacks

When a Client holds cross-origin tokens with empty contexts, it is possible for any Origin in the cross-origin set to deplete that Client set of tokens. To prevent this from happening, tokens can be scoped to single Origins (with non-empty origin_info) such that they can only be redeemed for a single Origin. Alternatively, if tokens are cross-Origin, Clients can use alternate methods to prevent many tokens from being redeemed at once. For example, if the Origin requests an excess of tokens, the Client could choose to not present any tokens for verification if a redemption had already occurred in a given time window.

Token challenges that include non-empty origin_info bind tokens to one or more specific origins. As described in Section 2.1, clients only accept such challenges from origin names listed in the origin_info string. Even if multiple origins are listed, a token can only be redeemed for an origin if the challenge has a match for the origin_info. For example, if "a.example.com" issues a challenge with an origin_info string of "a.example.com,b.example.com", a client could redeem a token fetched for this challenge if and only if "b.example.com" also included an origin_info string of "a.example.com,b.example.com". On the other hand, if "b.example.com" had an origin_info string of "b.example.com" or "b.example.com,a.example.com" or "a.example.com,b.example.com,c.example.com", the string would not match and the client would need to use a different token.

5.5. Timing Correlation Attacks

Context-bound token challenges require clients to obtain matching tokens when challenged, rather than presenting a token that was obtained from a different context in the past. This can make it more likely that issuance and redemption events will occur at approximately the same time. For example, if a client is challenged for a token with a unique context at time T1 and then subsequently obtains a token at time T2, a colluding issuer and origin can link this to the same client if T2 is unique to the client. This linkability is less feasible as the number of issuance events at time T2 increases. Depending on the "max-age" token challenge parameter, clients MAY try to add delay to the time between being challenged and redeeming a token to make this sort of linkability more difficult. For more discussion on correlation risks between token issuance and redemption, see Section 6.3 of [ARCHITECTURE].

5.6. Cross-Context Linkability Attacks

As discussed in Section 2.1, clients SHOULD discard any context-bound tokens upon flushing cookies or changing networks, to prevent an origin using the redemption context state as a cookie to recognize clients.

6. IANA Considerations

6.1. Authentication Scheme

This document registers the "PrivateToken" authentication scheme in the "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry" defined in [RFC9110], Section 16.4.

Authentication Scheme Name: PrivateToken

Pointer to specification text: Section 2 of this document

6.2. Token Type Registry

IANA is requested to create a new "Privacy Pass Token Type" registry in a new "Privacy Pass Parameters" page to list identifiers for issuance protocols defined for use with the Privacy Pass token authentication scheme. These identifiers are two-byte values, so the maximum possible value is 0xFFFF = 65535.

New registrations need to list the following attributes:

Value: The two-byte identifier for the algorithm
Name: Name of the issuance protocol
Token Structure: The contents of the Token structure in Section 2.2
Token Key Encoding: The encoding of the "token-key" parameter in Section 2.2
TokenChallenge Structure: The contents of the TokenChallenge structure in Section 2.1
Public Verifiability: A Y/N value indicating if the output tokens have the public verifiability property; see Section 3.5 of [ARCHITECTURE] for more details about this property.
Public Metadata: A Y/N value indicating if the output tokens can contain public metadata; see Section 3.5 of [ARCHITECTURE] for more details about this property.
Private Metadata: A Y/N value indicating if the output tokens can contain private metadata; see Section 3.5 of [ARCHITECTURE] for more details about this property.
Nk: The length in bytes of an output authenticator
Nid: The length of the token key identifier
Reference: Where this algorithm is defined
Notes: Any notes associated with the entry

New entries in this registry are subject to the Specification Required registration policy ([RFC8126], Section 4.6). Designated experts need to ensure that the token type is defined to be used for both token issuance and redemption. Additionally, the experts can reject registrations on the basis that they do not meet the security and privacy requirements for issuance protocols defined in Section 3.2 of [ARCHITECTURE].

[ISSUANCE] defines entries for this registry.

6.2.1. Reserved Values

This document defines several Reserved values, which can be used by clients and servers to send "greased" values in token challenges and redemptions to ensure that implementations remain able to handle unknown token types gracefully (this technique is inspired by [RFC8701]). Implementations SHOULD select reserved values at random when including them in greased messages. Servers can include these in TokenChallenge structures, either as the only challenge when no real token type is desired, or as one challenge in a list of challenges that include real values. Clients can include these in Token structures when they are not able to present a real token. The contents of the Token structure SHOULD be filled with random bytes when using greased values.

The initial contents for this registry consists of multiple reserved values, with the following attributes, which are repeated for each registration:

Value: 0x0000, 0x02AA, 0x1132, 0x2E96, 0x3CD3, 0x4473, 0x5A63, 0x6D32, 0x7F3F, 0x8D07, 0x916B, 0xA6A4, 0xBEAB, 0xC3F3, 0xDA42, 0xE944, 0xF057
Name: RESERVED
Token Structure: Random bytes
Token Key Encoding: Random bytes
TokenChallenge Structure: Random bytes
Publicly Verifiable: N/A
Public Metadata: N/A
Private Metadata: N/A
Nk: N/A
Nid: N/A
Reference: This document
Notes: None

7. References

7.1. Normative References

[ARCHITECTURE] Davidson, A., Iyengar, J., and C. A. Wood, "The Privacy Pass Architecture", Work in Progress, Internet-

Draft, draft-ietf-privacypass-architecture-16, 25 September 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-architecture-16>.

[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>.

[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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[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>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[SHS] Dang, Q. H. and National Institute of Standards and Technology, "Secure Hash Standard", DOI 10.6028/nist.fips.180-4, July 2015, <https://doi.org/10.6028/nist.fips.180-4>.

[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>.

[URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

7.2. Informative References

[COOKIES] Bingler, S., West, M., and J. Wilander, "Cookies: HTTP State Management Mechanism", Work in Progress, Internet-Draft, draft-ietf-httpbis-rfc6265bis-12, 10 May 2023,

<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-12>.

[ISSUANCE] Celi, S., Davidson, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-16, 3 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-16>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/rfc/rfc4086>.

[RFC8470] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <https://www.rfc-editor.org/rfc/rfc8470>.

[RFC8701] Benjamin, D., "Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility", RFC 8701, DOI 10.17487/RFC8701, January 2020, <https://www.rfc-editor.org/rfc/rfc8701>.

[RFC9001] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <https://www.rfc-editor.org/rfc/rfc9001>.

Appendix A. Test Vectors

This section includes test vectors for the HTTP authentication scheme specified in this document. It consists of the following types of test vectors:

1. Test vectors for the challenge and redemption protocols. Implementations can use these test vectors for verifying code that builds and encodes TokenChallenge structures, as well as code that produces a well-formed Token bound to a TokenChallenge.

2. Test vectors for the HTTP headers used for authentication. Implementations can use these test vectors for validating whether they parse HTTP authentication headers correctly to produce TokenChallenge structures and the other associated parameters, such as the token-key and max-age values.

A.1. Challenge and Redemption Structure Test Vectors

This section includes test vectors for the challenge and redemption functionalities described in Section 2.1 and Section 2.2. Each test vector lists the following values:

- *token_type: The type of token issuance protocol, a value from Section 6.2. For these test vectors, token_type is 0x0002, corresponding to the issuance protocol in [ISSUANCE].

- *issuer_name: The name of the issuer in the TokenChallenge structure, represented as a hexadecimal string.

- *redemption_context: The redemption context in the TokenChallenge structure, represented as a hexadecimal string.

- *origin_info: The origin info in the TokenChallenge structure, represented as a hexadecimal string.

- *nonce: The nonce in the Token structure, represented as a hexadecimal string.

- *token_key: The public token-key, encoded based on the corresponding token type, represented as a hexadecimal string.

- *token_authenticator_input: The values in the Token structure used to compute the Token authenticator value, represented as a hexadecimal string.

Test vectors are provided for each of the following TokenChallenge configurations:

1. TokenChallenge with a single origin and non-empty redemption context

2. TokenChallenge with a single origin and empty redemption context

3. TokenChallenge with an empty origin and redemption context

4. TokenChallenge with an empty origin and non-empty redemption context

5. TokenChallenge with a multiple origins and non-empty redemption context

6. TokenChallenge for greasing

These test vectors are below.

```
// Test vector 1:
//   token_type(0002), issuer_name(issuer.example),
//   origin_info(origin.example), redemption_context(non-empty)
token_type: 0002
issuer_name: 6973737565722e6578616d706c65
redemption_context:
476ac2c935f458e9b2d7af32dacfbd22dd6023ef5887a789f1abe004e79bb5bb
origin_info: 6f726967696e2e6578616d706c65
nonce:
e01978182c469e5e026d66558ee186568614f235e41ef7e2378e6f202688abab
token_key_id:
ca572f8982a9ca248a3056186322d93ca147266121ddeb5632c07f1f71cd2708
token_authenticator_input: 0002e01978182c469e5e026d66558ee1865686
14f235e41ef7e2378e6f202688abab8e1d5518ec82964255526efd8f9db88205a
8ddd3ffb1db298fcc3ad36c42388fca572f8982a9ca248a3056186322d93ca147
266121ddeb5632c07f1f71cd2708

// Test vector 2:
//   token_type(0002), issuer_name(issuer.example),
//   origin_info(origin.example), redemption_context(empty)
token_type: 0002
issuer_name: 6973737565722e6578616d706c65
redemption_context:
origin_info: 6f726967696e2e6578616d706c65
nonce:
e01978182c469e5e026d66558ee186568614f235e41ef7e2378e6f202688abab
token_key_id:
ca572f8982a9ca248a3056186322d93ca147266121ddeb5632c07f1f71cd2708
token_authenticator_input: 0002e01978182c469e5e026d66558ee1865686
14f235e41ef7e2378e6f202688abab11e15c91a7c2ad02abd66645802373db1d8
23bea80f08d452541fb2b62b5898bca572f8982a9ca248a3056186322d93ca147
266121ddeb5632c07f1f71cd2708

// Test vector 3:
//   token_type(0002), issuer_name(issuer.example),
//   origin_info(), redemption_context(empty)
token_type: 0002
issuer_name: 6973737565722e6578616d706c65
redemption_context:
origin_info:
nonce:
e01978182c469e5e026d66558ee186568614f235e41ef7e2378e6f202688abab
token_key_id:
ca572f8982a9ca248a3056186322d93ca147266121ddeb5632c07f1f71cd2708
token_authenticator_input: 0002e01978182c469e5e026d66558ee1865686
14f235e41ef7e2378e6f202688ababb741ec1b6fd05f1e95f8982906aec161289
6d9ca97d53eef94ad3c9fe023f7a4ca572f8982a9ca248a3056186322d93ca147
266121ddeb5632c07f1f71cd2708
```

```
// Test vector 4:
//   token_type(0002), issuer_name(issuer.example),
//   origin_info(), redemption_context(non-empty)
token_type: 0002
issuer_name: 6973737565722e6578616d706c65
redemption_context:
476ac2c935f458e9b2d7af32dacfbd22dd6023ef5887a789f1abe004e79bb5bb
origin_info:
nonce:
e01978182c469e5e026d66558ee186568614f235e41ef7e2378e6f202688abab
token_key_id:
ca572f8982a9ca248a3056186322d93ca147266121ddeb5632c07f1f71cd2708
token_authenticator_input: 0002e01978182c469e5e026d66558ee1865686
14f235e41ef7e2378e6f202688ababb85fb5bc06edeb0e8e8bdb5b3bea8c4fa40
837c82e8bcaf5882c81e14817ea18ca572f8982a9ca248a3056186322d93ca147
266121ddeb5632c07f1f71cd2708

// Test vector 5:
//   token_type(0002), issuer_name(issuer.example),
//   origin_info(foo.example,bar.example),
//   redemption_context(non-empty)
token_type: 0002
issuer_name: 6973737565722e6578616d706c65
redemption_context:
476ac2c935f458e9b2d7af32dacfbd22dd6023ef5887a789f1abe004e79bb5bb
origin_info: 666f6f2e6578616d706c652c6261722e6578616d706c65
nonce:
e01978182c469e5e026d66558ee186568614f235e41ef7e2378e6f202688abab
token_key_id:
ca572f8982a9ca248a3056186322d93ca147266121ddeb5632c07f1f71cd2708
token_authenticator_input: 0002e01978182c469e5e026d66558ee1865686
14f235e41ef7e2378e6f202688ababa2a775866b6ae0f98944910c8f48728d8a2
735b9157762ddbf803f70e2e8ba3eca572f8982a9ca248a3056186322d93ca147
266121ddeb5632c07f1f71cd2708

// Test vector 6:
//   token_type(0000), structure(random_bytes)
token_type: 0000
token_authenticator_input: 000058405ad31e286e874cb42d0ef9d50461ae
703bb71a21178beb429c43c0effe587456d856f0f2bdfc216ef93d5c225e2a93e
84cb686e63919788087f7ab1054aa817f09dcb919a0ed6f90fe887e8b08cd1eee
44d5be8d813eda9f2656db61c932db8d73f8690604ded0120157923bbd19d5549
e639e4de07530aee1d370f5187b678685715bd878dde24346751eb532a87b71ea
40bbe5a13218658e303c648eb03817453690bfcbe8255081bf27ff0891cd02ee2
483e48a2c494bdef696f943fa992a65303292c25d0d3f62da86a70d0b020f0ff5
b90d0ff0f6abdb097d321fde04f3a1994e63bcd35a88c21236c7dc67600482223
f54b25e39a250439f27ecb5ae9eb8ed548a3ec1f1d6f510d08281929c8fe08834
2959e35ea9b3b6f6a96fc1a8edba4ed297f4cf02d0e4482b79a11f671745d7b7d
```

b120eddd8a4c2b6501bbc895b2160b8071615d9c1b18f32e056bfee29deac6a7d
6cf7b522a5badd63b9cb

A.2. HTTP Header Test Vectors

This section includes test vectors the contents of the HTTP authentication headers. Each test vector consists of one or more challenges that comprise a WWW-Authenticate header, as defined in {(choosing-between-multiple-challenges}}. For each challenge, the token-type, token-key, max-age, and token-challenge parameters are listed. Each challenge also includes an unknown (not specified) parameter that implementations are meant to ignore.

The parameters for each challenge are indexed by their position in the WWW-Authentication challenge list. For example, token-key-0 denotes the token-key parameter for the first challenge in the list, whereas token-key-1 denotes the token-key for the second challenge in the list.

The resulting wire-encoded WWW-Authentication header based on this list of challenges is then listed at the end. Line folding is only used to fit the document formatting constraints and not supported in actual requests.

The last challenge on this list includes Basic authentication, a grease challenge, and a valid challenge for token type 0x0001. Correct client implementations will ignore the Basic and grease challenges.

```
token-type-0: 0x0002
token-key-0: 30820152303d06092a864886f70d01010a3030a00d300b060960864
8016503040202a11a301806092a864886f70d010108300b060960864801650304020
2a2030201300382010f003082010a0282010100cb1aed6b6a95f5b1ce013a4cfcab2
5b94b2e64a23034e4250a7eab43c0df3a8c12993af12b111908d4b471bec31d4b6c9
ad9cdda90612a2ee903523e6de5a224d6b02f09e5c374d0cfe01d8f529c500a78a2f
67908fa682b5a2b430c81eaf1af72d7b5e794fc98a3139276879757ce453b526ef9b
f6ceb99979b8423b90f4461a22af37aab0cf5733f7597abe44d31c732db68a181c6c
bbe607d8c0e52e0655fd9996dc584eca0be87afbcd78a337d17b1dba9e828bbd81e2
91317144e7ff89f55619709b096cbb9ea474cead264c2073fe49740c01f00e109106
066983d21e5f83f086e2e823c879cd43cef700d2a352a9babd612d03cad02db134b7
e225a5f0203010001
max-age-0: 10
token-challenge-0: 0002000e6973737565722e6578616d706c65208a3e83a33d9
8005d2f30bef419fa6bf4cd5c6005e36b1285bbb4ccd40fa4b383000e6f726967696
e2e6578616d706c65

WWW-Authenticate: PrivateToken challenge="AAIADmlzc3Vlci5leGFtcGxlII
o-g6M9mABdLzC-9Bn6a_TNXGAF42sShbu0zNQPpLODAA5vcmlnaW4uZXhhbXBsZQ==",
 token-key="MIIBUjA9BgkqhkiG9w0BAQowMKANMAsGCWCGSAFlAwQCAqEaMBgGCSqG
SIb3DQEBCDALBglghkgBZQMEAgKiAwIBMAOCAQ8AMIIBCgKCAQEAyxrta2qV9bHOATpM
_KsluUsuZKIwNOQlCn6rQ8DfOowSmTrxKxEZCNS0cb7DHUtsmtnN2pBhKi7pA1I-beWi
JNawLwnlw3TQz-Adj1KcUAp4ovZ5CPpoK1orQwyB6vGvcte155T8mKMTknaHl1fORTtS
bvm_bOuZl5uEI7kPRGGiKvN6qwz1cz91l6vkTTHHMttooYHGy75gfYwOUuBlX9mZbcWE
7KC-h6-814ozfRex26noKLvYHikTFxROf_ifVWGXCbCWy7nqR0zq0mTCBz_kl0DAHwDh
CRBgZpg9IeX4PwhuLoI8h5zUPO9wDSo1Kpur1hLQPK0C2xNLfiJaXwIDAQAB",unknow
nChallengeAttribute="ignore-me", max-age="10"

token-type-0: 0x0002
token-key-0: 30820152303d06092a864886f70d01010a3030a00d300b060960864
8016503040202a11a301806092a864886f70d010108300b060960864801650304020
2a2030201300382010f003082010a0282010100cb1aed6b6a95f5b1ce013a4cfcab2
5b94b2e64a23034e4250a7eab43c0df3a8c12993af12b111908d4b471bec31d4b6c9
ad9cdda90612a2ee903523e6de5a224d6b02f09e5c374d0cfe01d8f529c500a78a2f
67908fa682b5a2b430c81eaf1af72d7b5e794fc98a3139276879757ce453b526ef9b
f6ceb99979b8423b90f4461a22af37aab0cf5733f7597abe44d31c732db68a181c6c
bbe607d8c0e52e0655fd9996dc584eca0be87afbcd78a337d17b1dba9e828bbd81e2
91317144e7ff89f55619709b096cbb9ea474cead264c2073fe49740c01f00e109106
066983d21e5f83f086e2e823c879cd43cef700d2a352a9babd612d03cad02db134b7
e225a5f0203010001
max-age-0: 10
token-challenge-0: 0002000e6973737565722e6578616d706c65208a3e83a33d9
8005d2f30bef419fa6bf4cd5c6005e36b1285bbb4ccd40fa4b383000e6f726967696
e2e6578616d706c65
token-type-1: 0x0001
token-key-1: ebb1fed338310361c08d0c7576969671296e05e99a17d7926dfc28a
53fabd489fac0f82bca86249a668f3a5bfab374c9
max-age-1: 10
token-challenge-1: 0001000e6973737565722e6578616d706c65208a3e83a33d9
```

```
8005d2f30bef419fa6bf4cd5c6005e36b1285bbb4ccd40fa4b383000e6f726967696
e2e6578616d706c65

WWW-Authenticate: PrivateToken challenge="AAIADmlzc3Vlci5leGFtcGxlII
o-g6M9mABdLzC-9Bn6a_TNXGAF42sShbu0zNQPpLODAA5vcmlnaW4uZXhhbXBsZQ==",
 token-key="MIIBUjA9BgkqhkiG9w0BAQowMKANMAsGCWCGSAFlAwQCAqEaMBgGCSqG
SIb3DQEBCDALBglghkgBZQMEAgKiAwIBMAOCAQ8AMIIBCgKCAQEAyxrta2qV9bHOATpM
_KsluUsuZKIwNOQlCn6rQ8DfOowSmTrxKxEZCNS0cb7DHUtsmtnN2pBhKi7pA1I-beWi
JNawLwnlw3TQz-Adj1KcUAp4ovZ5CPpoK1orQwyB6vGvcte155T8mKMTknaHl1fORTtS
bvm_bOuZl5uEI7kPRGGiKvN6qwz1cz91l6vkTTHHMttooYHGy75gfYwOUuBlX9mZbcWE
7KC-h6-814ozfRex26noKLvYHikTFxROf_ifVWGXCbCWy7nqR0zq0mTCBz_kl0DAHwDh
CRBgZpg9IeX4PwhuLoI8h5zUPO9wDSo1Kpur1hLQPK0C2xNLfiJaXwIDAQAB",unknow
nChallengeAttribute="ignore-me", max-age="10", PrivateToken challeng
e="AAEADmlzc3Vlci5leGFtcGxlIIo-g6M9mABdLzC-9Bn6a_TNXGAF42sShbu0zNQPp
LODAA5vcmlnaW4uZXhhbXBsZQ==", token-key="67H-0zgxA2HAjQx1dpaWcSluBem
aF9eSbfwopT-r1In6wPgryoYkmmaPOlv6s3TJ",unknownChallengeAttribute="ig
nore-me", max-age="10"

token-type-0: 0x0000
token-key-0: 856de3c710b892e7cca1ae5eb121af42ca8e779137a11224228c9b9
9b0729bf84d5057d030000309b8f0d06ccffa17561f9eacd4c312e985a6bc60ffbea
0610264dcb1726255313da81d665692686a1d8644f1516bf612cea009e6dff6d9a9a
959fb538e1b5b2343c092992942382bdde22d5b324b1e4618ed21d7806286c2ce
token-challenge-0: 0000acc3b25795c636fd9dd8b12982394abba8777d35978e8
77fc8848892a217233045ac25a3d55c07c54efe6372973fee0073e77fc61bf19ab88
0f20edf5d627502
token-type-1: 0x0001
token-key-1: ebb1fed338310361c08d0c7576969671296e05e99a17d7926dfc28a
53fabd489fac0f82bca86249a668f3a5bfab374c9
max-age-1: 10
token-challenge-1: 0001000e6973737565722e6578616d706c65208a3e83a33d9
8005d2f30bef419fa6bf4cd5c6005e36b1285bbb4ccd40fa4b383000e6f726967696
e2e6578616d706c65

WWW-Authenticate: Basic realm="grease", PrivateToken challenge="AACs
w7JXlcY2_Z3YsSmCOUq7qHd9NZeOh3_IhIiSohcjMEWsJaPVXAfFTv5jcpc_7gBz53_G
G_GauIDyDt9dYnUC",token-key="hW3jxxC4kufMoa5esSGvQsqOd5E3oRIkIoybmbB
ym_hNUFfQMAADCbjw0GzP-hdWH56s1MMS6YWmvGD_vqBhAmTcsXJiVTE9qB1mVpJoah2
GRPFRa_YSzqAJ5t_22ampWftTjhtbI0PAkpkpQjgr3eItWzJLHkYY7SHXgGKGws4=",
PrivateToken challenge="AAEADmlzc3Vlci5leGFtcGxlIIo-g6M9mABdLzC-9Bn6
a_TNXGAF42sShbu0zNQPpLODAA5vcmlnaW4uZXhhbXBsZQ==", token-key="67H-0z
gxA2HAjQx1dpaWcSluBemaF9eSbfwopT-r1In6wPgryoYkmmaPOlv6s3TJ",unknownC
hallengeAttribute="ignore-me", max-age="10"
```

Authors' Addresses

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014,
United States of America

Email: tpauly@apple.com

Steven Valdez
Google LLC

Email: svaldez@chromium.org

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net