Stream: Internet Engineering Task Force (IETF)
RFC: 9449
Category: Standards Track
Published: September 2023
ISSN: 2070-1721
Authors: D. Fett, *Authlete*; B. Campbell, *Ping Identity*; J. Bradley, *Yubico*; T. Lodderstedt, *Tuconic*; M. Jones, *Self-Issued Consulting*; D. Waite, *Ping Identity*

RFC 9449
OAuth 2.0 Demonstrating Proof of Possession (DPoP)

Abstract

This document describes a mechanism for sender-constraining OAuth 2.0 tokens via a proof-of-possession mechanism on the application level. This mechanism allows for the detection of replay attacks with access and refresh tokens.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9449.

Copyright Notice

Table of Contents

1. Introduction

Demonstrating Proof of Possession (DPoP) is an application-level mechanism for sender-constraining OAuth [RFC6749] access and refresh tokens. It enables a client to prove the possession of a public/private key pair by including a `DPoP` header in an HTTP request. The value of the header is a JSON Web Token (JWT) [RFC7519] that enables the authorization server to bind issued tokens to the public part of a client's key pair. Recipients of such tokens are then able to verify the binding of the token to the key pair that the client has demonstrated that it holds via the `DPoP` header, thereby providing some assurance that the client presenting the token also possesses the private key. In other words, the legitimate presenter of the token is constrained to be the sender that holds and proves possession of the private part of the key pair.

The mechanism specified herein can be used in cases where other methods of sender-constraining tokens that utilize elements of the underlying secure transport layer, such as [RFC8705] or [TOKEN-BINDING], are not available or desirable. For example, due to a sub-par user experience of TLS client authentication in user agents and a lack of support for HTTP token binding, neither mechanism can be used if an OAuth client is an application that is dynamically downloaded and executed in a web browser (sometimes referred to as a "single-page application"). Additionally, applications that are installed and run directly on a user's device are well positioned to benefit from DPoP-bound tokens that guard against the misuse of tokens by a compromised or malicious resource. Such applications often have dedicated protected storage for cryptographic keys.

DPoP can be used to sender-constrain access tokens regardless of the client authentication method employed, but DPoP itself is not used for client authentication. DPoP can also be used to sender-constrain refresh tokens issued to public clients (those without authentication credentials associated with the `client_id`).

1.1. Conventions and 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 specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234].

This specification uses the terms "access token", "refresh token", "authorization server", "resource server", "authorization endpoint", "authorization request", "authorization response", "token endpoint", "grant type", "access token request", "access token response", "client", "public client", and "confidential client" defined by "The OAuth 2.0 Authorization Framework" [RFC6749].

The terms "request", "response", "header field", and "target URI" are imported from [RFC9110].

The terms "JOSE" and "JOSE Header" are imported from [RFC7515].

This document contains non-normative examples of partial and complete HTTP messages. Some examples use a single trailing backslash to indicate line wrapping for long values, as per [RFC8792]. The character and leading spaces on wrapped lines are not part of the value.

2. Objectives

The primary aim of DPoP is to prevent unauthorized or illegitimate parties from using leaked or stolen access tokens, by binding a token to a public key upon issuance and requiring that the client proves possession of the corresponding private key when using the token. This constrains the legitimate sender of the token to only the party with access to the private key and gives the server receiving the token added assurances that the sender is legitimately authorized to use it.

Access tokens that are sender-constrained via DPoP thus stand in contrast to the typical bearer token, which can be used by any party in possession of such a token. Although protections generally exist to prevent unintended disclosure of bearer tokens, unforeseen vectors for leakage have occurred due to vulnerabilities and implementation issues in other layers in the protocol or software stack (see, e.g., Compression Ratio Info-leak Made Easy (CRIME) [CRIME], Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH) [BREACH], Heartbleed [Heartbleed], and the Cloudflare parser bug [Cloudbleed]). There have also been numerous published token theft attacks on OAuth implementations themselves ([GitHub.Tokens] is just one high-profile example). DPoP provides a general defense in depth against the impact of unanticipated token leakage. DPoP is not, however, a substitute for a secure transport and **MUST** always be used in conjunction with HTTPS.

The very nature of the typical OAuth protocol interaction necessitates that the client discloses the access token to the protected resources that it accesses. The attacker model in [SECURITY-TOPICS] describes cases where a protected resource might be counterfeit, malicious, or compromised and plays received tokens against other protected resources to gain unauthorized access. Audience-restricted access tokens (e.g., using the JWT [RFC7519] `aud` claim) can prevent such misuse. However, doing so in practice has proven to be prohibitively cumbersome for many deployments (despite extensions such as [RFC8707]). Sender-constraining access tokens is a more robust and straightforward mechanism to prevent such token replay at a different endpoint, and DPoP is an accessible application-layer means of doing so.

Due to the potential for cross-site scripting (XSS), browser-based OAuth clients bring to bear added considerations with respect to protecting tokens. The most straightforward XSS-based attack is for an attacker to exfiltrate a token and use it themselves completely independent of the legitimate client. A stolen access token is used for protected resource access, and a stolen refresh token is used for obtaining new access tokens. If the private key is non-extractable (as is possible with [W3C.WebCryptoAPI]), DPoP renders exfiltrated tokens alone unusable.

XSS vulnerabilities also allow an attacker to execute code in the context of the browser-based client application and maliciously use a token indirectly through the client. That execution context has access to utilize the signing key; thus, it can produce DPoP proofs to use in

conjunction with the token. At this application layer, there is most likely no feasible defense against this threat except generally preventing XSS; therefore, it is considered out of scope for DPoP.

Malicious XSS code executed in the context of the browser-based client application is also in a position to create DPoP proofs with timestamp values in the future and exfiltrate them in conjunction with a token. These stolen artifacts can later be used independent of the client application to access protected resources. To prevent this, servers can optionally require clients to include a server-chosen value into the proof that cannot be predicted by an attacker (nonce). In the absence of the optional nonce, the impact of pre-computed DPoP proofs is limited somewhat by the proof being bound to an access token on protected resource access. Because a proof covering an access token that does not yet exist cannot feasibly be created, access tokens obtained with an exfiltrated refresh token and pre-computed proofs will be unusable.

Additional security considerations are discussed in Section 11.

3. Concept

The main data structure introduced by this specification is a DPoP proof JWT that is sent as a header in an HTTP request, as described in detail below. A client uses a DPoP proof JWT to prove the possession of a private key corresponding to a certain public key.

Roughly speaking, a DPoP proof is a signature over:

- some data of the HTTP request to which it is attached,
- a timestamp,
- a unique identifier,
- an optional server-provided nonce, and
- a hash of the associated access token when an access token is present within the request.

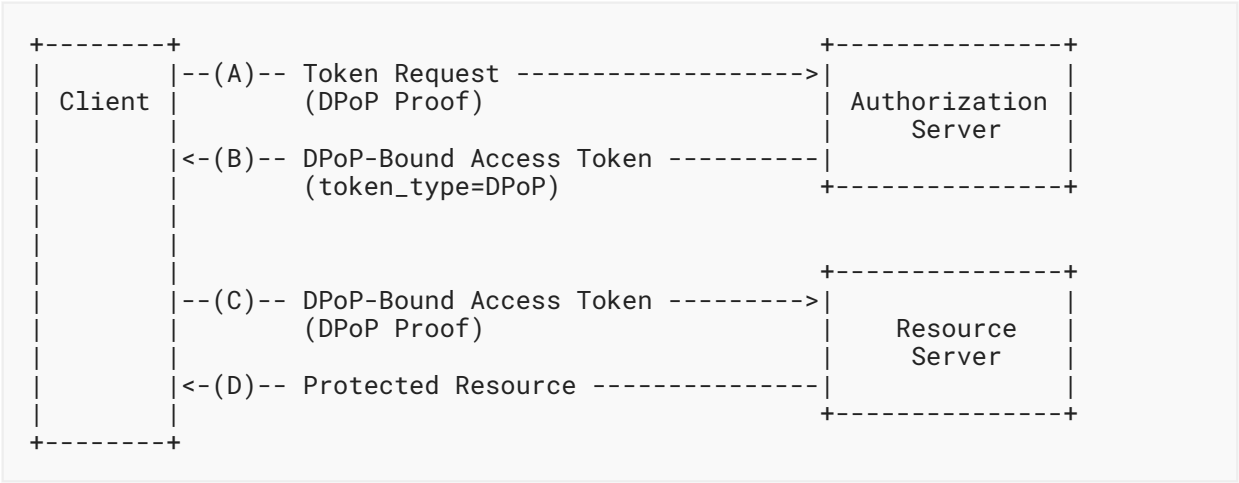

Figure 1: Basic DPoP Flow

The basic steps of an OAuth flow with DPoP (without the optional nonce) are shown in Figure 1.

A. In the token request, the client sends an authorization grant (e.g., an authorization code, refresh token, etc.) to the authorization server in order to obtain an access token (and potentially a refresh token). The client attaches a DPoP proof to the request in an HTTP header.
B. The authorization server binds (sender-constrains) the access token to the public key claimed by the client in the DPoP proof; that is, the access token cannot be used without proving possession of the respective private key. If a refresh token is issued to a public client, it is also bound to the public key of the DPoP proof.
C. To use the access token, the client has to prove possession of the private key by, again, adding a header to the request that carries a DPoP proof for that request. The resource server needs to receive information about the public key to which the access token is bound. This information may be encoded directly into the access token (for JWT-structured access tokens) or provided via token introspection endpoint (not shown). The resource server verifies that the public key to which the access token is bound matches the public key of the DPoP proof. It also verifies that the access token hash in the DPoP proof matches the access token presented in the request.
D. The resource server refuses to serve the request if the signature check fails or if the data in the DPoP proof is wrong, e.g., the target URI does not match the URI claim in the DPoP proof JWT. The access token itself, of course, must also be valid in all other respects.

The DPoP mechanism presented herein is not a client authentication method. In fact, a primary use case of DPoP is for public clients (e.g., single-page applications and applications on a user's device) that do not use client authentication. Nonetheless, DPoP is designed to be compatible with `private_key_jwt` and all other client authentication methods.

DPoP does not directly ensure message integrity, but it relies on the TLS layer for that purpose. See Section 11 for details.

4. DPoP Proof JWTs

DPoP introduces the concept of a DPoP proof, which is a JWT created by the client and sent with an HTTP request using the `DPoP` header field. Each HTTP request requires a unique DPoP proof.

A valid DPoP proof demonstrates to the server that the client holds the private key that was used to sign the DPoP proof JWT. This enables authorization servers to bind issued tokens to the corresponding public key (as described in Section 5) and enables resource servers to verify the key-binding of tokens that it receives (see Section 7.1), which prevents said tokens from being used by any entity that does not have access to the private key.

The DPoP proof demonstrates possession of a key and, by itself, is not an authentication or access control mechanism. When presented in conjunction with a key-bound access token as described in Section 7.1, the DPoP proof provides additional assurance about the legitimacy of the client to present the access token. However, a valid DPoP proof JWT is not sufficient alone to make access control decisions.

4.1. The DPoP HTTP Header

A DPoP proof is included in an HTTP request using the following request header field.

`DPoP`: A JWT that adheres to the structure and syntax of Section 4.2.

Figure 2 shows an example DPoP HTTP header field. The example uses "\" line wrapping per [RFC8792].

```
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik\
 VDIiwieCI6Imw4dEZyaHgtMzR0VjNoUklDUkRZOXpDa0RscEJoRjQyVVFVZldWQVdCR\
 nMiLCJ5IjoiOVZFNGpmX09rX282NHpiVFRsY3VOSmFqSG10NnY5VERWclUwQ2R2R1JE\
 QSIsImNydiI6IlAtMjU2In19.eyJqdGkiOiItQndDM0VTYzZhY2MybFRjIiwiaHRtIj\
 oiUE9TVCIsImh0dSI6Imh0dHBzOi8vc2VydmVyLmV4YW1wbGUuY29tL3Rva2VuIiwia\
 WF0IjoxNTYyMjYyNjE2fQ.2-GxA6T8lP4vfrg8v-FdWP0A0zdrj8igiMLvqRMUvwnQg\
 4PtFLbdLXiOSsX0x7NVY-FNyJK70nfbV37xRZT3Lg
```

Figure 2: Example `DPoP` Header

Note that per [RFC9110], header field names are case insensitive; thus, `DPoP`, `DPOP`, `dpop`, etc., are all valid and equivalent header field names. However, case is significant in the header field value.

The `DPoP` HTTP header field value uses the token68 syntax defined in Section 11.2 of [RFC9110] and is repeated below in Figure 3 for ease of reference.

```
DPoP        = token68
token68     = 1*( ALPHA / DIGIT /
                  "-" / "." / "_" / "~" / "+" / "/" ) *"="
```

Figure 3: DPoP Header Field ABNF

4.2. DPoP Proof JWT Syntax

A DPoP proof is a JWT [RFC7519] that is signed (using JSON Web Signature (JWS) [RFC7515]) with a private key chosen by the client (see below). The JOSE Header of a DPoP JWT **MUST** contain at least the following parameters:

`typ`: A field with the value `dpop+jwt`, which explicitly types the DPoP proof JWT as recommended in Section 3.11 of [RFC8725].

`alg`: An identifier for a JWS asymmetric digital signature algorithm from [IANA.JOSE.ALGS]. It **MUST NOT** be `none` or an identifier for a symmetric algorithm (Message Authentication Code (MAC)).

`jwk`: Represents the public key chosen by the client in JSON Web Key (JWK) [RFC7517] format as defined in Section 4.1.3 of [RFC7515]. It **MUST NOT** contain a private key.

The payload of a DPoP proof **MUST** contain at least the following claims:

`jti`: Unique identifier for the DPoP proof JWT. The value **MUST** be assigned such that there is a negligible probability that the same value will be assigned to any other DPoP proof used in the same context during the time window of validity. Such uniqueness can be accomplished by encoding (base64url or any other suitable encoding) at least 96 bits of pseudorandom data or by using a version 4 Universally Unique Identifier (UUID) string according to [RFC4122]. The `jti` can be used by the server for replay detection and prevention; see Section 11.1.

`htm`: The value of the HTTP method (Section 9.1 of [RFC9110]) of the request to which the JWT is attached.

`htu`: The HTTP target URI (Section 7.1 of [RFC9110]) of the request to which the JWT is attached, without query and fragment parts.

`iat`: Creation timestamp of the JWT (Section 4.1.6 of [RFC7519]).

When the DPoP proof is used in conjunction with the presentation of an access token in protected resource access (see Section 7), the DPoP proof **MUST** also contain the following claim:

`ath`: Hash of the access token. The value **MUST** be the result of a base64url encoding (as defined in Section 2 of [RFC7515]) the SHA-256 [SHS] hash of the ASCII encoding of the associated access token's value.

When the authentication server or resource server provides a `DPoP-Nonce` HTTP header in a response (see Sections 8 and 9), the DPoP proof **MUST** also contain the following claim:

`nonce`: A recent nonce provided via the `DPoP-Nonce` HTTP header.

A DPoP proof **MAY** contain other JOSE Header Parameters or claims as defined by extension, profile, or deployment-specific requirements.

Figure 4 is a conceptual example showing the decoded content of the DPoP proof in Figure 2. The JSON of the JWT header and payload are shown, but the signature part is omitted. As usual, line breaks and extra spaces are included for formatting and readability.

```
{
  "typ":"dpop+jwt",
  "alg":"ES256",
  "jwk": {
    "kty":"EC",
    "x":"l8tFrhx-34tV3hRICRDY9zCkDlpBhF42UQUfWVAWBFs",
    "y":"9VE4jf_Ok_o64zbTTlcuNJajHmt6v9TDVrU0CdvGRDA",
    "crv":"P-256"
  }
}
.
{
  "jti":"-BwC3ESc6acc2lTc",
  "htm":"POST",
  "htu":"https://server.example.com/token",
  "iat":1562262616
}
```

Figure 4: Example JWT Content of a DPoP Proof

Of the HTTP request, only the HTTP method and URI are included in the DPoP JWT; therefore, only these two message parts are covered by the DPoP proof. The idea is to sign just enough of the HTTP data to provide reasonable proof of possession with respect to the HTTP request. This design approach of using only a minimal subset of the HTTP header data is to avoid the substantial difficulties inherent in attempting to normalize HTTP messages. Nonetheless, DPoP proofs can be extended to contain other information of the HTTP request (see also Section 11.7).

4.3. Checking DPoP Proofs

To validate a DPoP proof, the receiving server **MUST** ensure the following:

1. There is not more than one `DPoP` HTTP request header field.
2. The DPoP HTTP request header field value is a single and well-formed JWT.
3. All required claims per Section 4.2 are contained in the JWT.
4. The `typ` JOSE Header Parameter has the value `dpop+jwt`.

5. The `alg` JOSE Header Parameter indicates a registered asymmetric digital signature algorithm [IANA.JOSE.ALGS], is not `none`, is supported by the application, and is acceptable per local policy.
6. The JWT signature verifies with the public key contained in the `jwk` JOSE Header Parameter.
7. The `jwk` JOSE Header Parameter does not contain a private key.
8. The `htm` claim matches the HTTP method of the current request.
9. The `htu` claim matches the HTTP URI value for the HTTP request in which the JWT was received, ignoring any query and fragment parts.
10. If the server provided a nonce value to the client, the `nonce` claim matches the server-provided nonce value.
11. The creation time of the JWT, as determined by either the `iat` claim or a server managed timestamp via the `nonce` claim, is within an acceptable window (see Section 11.1).
12. If presented to a protected resource in conjunction with an access token,
    - ensure that the value of the `ath` claim equals the hash of that access token, and
    - confirm that the public key to which the access token is bound matches the public key from the DPoP proof.

To reduce the likelihood of false negatives, servers **SHOULD** employ syntax-based normalization (Section 6.2.2 of [RFC3986]) and scheme-based normalization (Section 6.2.3 of [RFC3986]) before comparing the `htu` claim.

These checks may be performed in any order.

5. DPoP Access Token Request

To request an access token that is bound to a public key using DPoP, the client **MUST** provide a valid DPoP proof JWT in a `DPoP` header when making an access token request to the authorization server's token endpoint. This is applicable for all access token requests regardless of grant type (e.g., the common `authorization_code` and `refresh_token` grant types and extension grants such as the JWT authorization grant [RFC7523]). The HTTP request shown in Figure 5 illustrates such an access token request using an authorization code grant with a DPoP proof JWT in the `DPoP` header. Figure 5 uses "\" line wrapping per [RFC8792].

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik\
 VDIiwieCI6Imw4dEZyaHgtMzR0VjNoUklDUkRZOXpDa0RscEJoRjQyVVFVZldWQVdCR\
 nMiLCJ5IjoiOVZFNGpmX09rX282NHpiVFRsY3VOSmFqSG10NnY5VERWclUwQ2R2R1JE\
 QSIsImNydiI6IlAtMjU2In19.eyJqdGkiOiItQndDM0VTYzZhY2MybFRjIiwiaHRtIj\
 oiUE9TVCIsImh0dSI6Imh0dHBzOi8vc2VydmVyLmV4YW1wbGUuY29tL3Rva2VuIiwia\
 WF0IjoxNTYyMjYyNjE2fQ.2-GxA6T8lP4vfrg8v-FdWP0A0zdrj8igiMLvqRMUvwnQg\
 4PtFLbdLXiOSsX0x7NVY-FNyJK70nfbV37xRZT3Lg

grant_type=authorization_code\
&client_id=s6BhdRkqt\
&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb\
&code_verifier=bEaL42izcC-o-xBk0K2vuJ6U-y1p9r_wW2dFWIWgjz-
```

Figure 5: Token Request for a DPoP Sender-Constrained Token Using an Authorization Code

The `DPoP` HTTP header field **MUST** contain a valid DPoP proof JWT. If the DPoP proof is invalid, the authorization server issues an error response per Section 5.2 of [RFC6749] with `invalid_dpop_proof` as the value of the `error` parameter.

To sender-constrain the access token after checking the validity of the DPoP proof, the authorization server associates the issued access token with the public key from the DPoP proof, which can be accomplished as described in Section 6. A `token_type` of `DPoP` **MUST** be included in the access token response to signal to the client that the access token was bound to its DPoP key and can be used as described in Section 7.1. The example response shown in Figure 6 illustrates such a response.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
 "access_token": "Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU",
 "token_type": "DPoP",
 "expires_in": 2677,
 "refresh_token": "Q..Zkm29lexi8VnWg2zPW1x-tgGad0Ibc3s3EwM_Ni4-g"
}
```

Figure 6: Access Token Response

The example response in Figure 6 includes a refresh token that the client can use to obtain a new access token when the previous one expires. Refreshing an access token is a token request using the `refresh_token` grant type made to the authorization server's token endpoint. As with all access token requests, the client makes it a DPoP request by including a DPoP proof, as shown in Figure 7. Figure 7 uses "\" line wrapping per [RFC8792].

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik\
 VDIiwieCI6Imw4dEZyaHgtMzR0VjNoUklDUkRZOXpDa0RscEJoRjQyVVFVZldWQVdCR\
 nMiLCJ5IjoiOVZFNGpmX09rX282NHpiVFRsY3VOSmFqSG10NnY5VERWclUwQ2R2R1JE\
 QSIsImNydiI6IlAtMjU2In19.eyJqdGkiOiItQndDM0VTYzZhY2MybFRjIiwiaHRtIj\
 oiUE9TVCIsImh0dSI6Imh0dHBzOi8vc2VydmVyLmV4YW1wbGUuY29tL3Rva2VuIiwia\
 WF0IjoxNTYyMjY1Mjk2fQ.pAqut2IRDm_De6PR93SYmGBPXpwrAk90e8cP2hjiaG5Qs\
 GSuKDYW7_X620BxqhvYC8ynrrvZLTk41mSRroapUA

grant_type=refresh_token\
&client_id=s6BhdRkqt\
&refresh_token=Q..Zkm29lexi8VnWg2zPW1x-tgGad0Ibc3s3EwM_Ni4-g
```

Figure 7: Token Request for a DPoP-Bound Token Using a Refresh Token

When an authorization server supporting DPoP issues a refresh token to a public client that presents a valid DPoP proof at the token endpoint, the refresh token **MUST** be bound to the respective public key. The binding **MUST** be validated when the refresh token is later presented to get new access tokens. As a result, such a client **MUST** present a DPoP proof for the same key that was used to obtain the refresh token each time that refresh token is used to obtain a new access token. The implementation details of the binding of the refresh token are at the discretion of the authorization server. Since the authorization server both produces and validates its refresh tokens, there is no interoperability consideration in the specific details of the binding.

An authorization server **MAY** elect to issue access tokens that are not DPoP bound, which is signaled to the client with a value of `Bearer` in the `token_type` parameter of the access token response per [RFC6750]. For a public client that is also issued a refresh token, this has the effect of DPoP-binding the refresh token alone, which can improve the security posture even when protected resources are not updated to support DPoP.

If the access token response contains a different `token_type` value than `DPoP`, the access token protection provided by DPoP is not given. The client **MUST** discard the response in this case if this protection is deemed important for the security of the application; otherwise, the client may continue as in a regular OAuth interaction.

Refresh tokens issued to confidential clients (those having established authentication credentials with the authorization server) are not bound to the DPoP proof public key because they are already sender-constrained with a different existing mechanism. The OAuth 2.0 Authorization Framework [RFC6749] already requires that an authorization server bind refresh tokens to the client to which they were issued and that confidential clients authenticate to the authorization server when presenting a refresh token. As a result, such refresh tokens are sender-constrained by way of the client identifier and the associated authentication requirement. This existing sender-constraining mechanism is more flexible (e.g., it allows credential rotation for the client without invalidating refresh tokens) than binding directly to a particular public key.

5.1. Authorization Server Metadata

This document introduces the following authorization server metadata [RFC8414] parameter to signal support for DPoP in general and the specific JWS `alg` values the authorization server supports for DPoP proof JWTs.

`dpop_signing_alg_values_supported`: A JSON array containing a list of the JWS `alg` values (from the [IANA.JOSE.ALGS] registry) supported by the authorization server for DPoP proof JWTs.

5.2. Client Registration Metadata

The Dynamic Client Registration Protocol [RFC7591] defines an API for dynamically registering OAuth 2.0 client metadata with authorization servers. The metadata defined by [RFC7591], and registered extensions to it, also imply a general data model for clients that is useful for authorization server implementations even when the Dynamic Client Registration Protocol isn't in play. Such implementations will typically have some sort of user interface available for managing client configuration.

This document introduces the following client registration metadata [RFC7591] parameter to indicate that the client always uses DPoP when requesting tokens from the authorization server.

`dpop_bound_access_tokens`: A boolean value specifying whether the client always uses DPoP for token requests. If omitted, the default value is `false`.

If the value is `true`, the authorization server **MUST** reject token requests from the client that do not contain the DPoP header.

6. Public Key Confirmation

Resource servers **MUST** be able to reliably identify whether an access token is DPoP-bound and ascertain sufficient information to verify the binding to the public key of the DPoP proof (see Section 7.1). Such a binding is accomplished by associating the public key with the token in a way that can be accessed by the protected resource, such as embedding the JWK hash in the issued access token directly, using the syntax described in Section 6.1, or through token introspection as described in Section 6.2. Other methods of associating a public key with an access token are possible per an agreement by the authorization server and the protected resource; however, they are beyond the scope of this specification.

Resource servers supporting DPoP **MUST** ensure that the public key from the DPoP proof matches the one bound to the access token.

6.1. JWK Thumbprint Confirmation Method

When access tokens are represented as JWTs [RFC7519], the public key information is represented using the `jkt` confirmation method member defined herein. To convey the hash of a public key in a JWT, this specification introduces the following JWT Confirmation Method [RFC7800] member for use under the `cnf` claim.

`jkt`: JWK SHA-256 Thumbprint confirmation method. The value of the `jkt` member **MUST** be the base64url encoding (as defined in [RFC7515]) of the JWK SHA-256 Thumbprint (according to [RFC7638]) of the DPoP public key (in JWK format) to which the access token is bound.

The following example JWT in Figure 8 with a decoded JWT payload shown in Figure 9 contains a `cnf` claim with the `jkt` JWK Thumbprint confirmation method member. The `jkt` value in these examples is the hash of the public key from the DPoP proofs in the examples shown in Section 5. The example uses "\" line wrapping per [RFC8792].

```
eyJhbGciOiJFUzI1NiIsImtpZCI6IkJlQUxrYiJ9.eyJzdWIiOiJzb21lb25lQGV4YW1\
wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20iLCJuYmYiOjE\
1NjIyNjI2MTEsImV4cCI6MTU2MjI2NjIxNiwiY25mIjp7ImprdCI6IjBaY09DT1JaTll\
5LURXcHFxMzBqWnlKR0hUTjBkMkhnbEJWM3VpZ3VBNEkifX0.3Tyo8VTcn6u_PboUmAO\
YUY1kfAavomW_YwYMkmRNizLJoQzWy2fCo79Zi5yObpIzjWb5xW4OGld7ESZrh0fsrA
```

Figure 8: JWT Containing a JWK SHA-256 Thumbprint Confirmation

```
{
  "sub":"someone@example.com",
  "iss":"https://server.example.com",
  "nbf":1562262611,
  "exp":1562266216,
  "cnf":
  {
    "jkt":"0ZcOCORZNYy-DWpqq30jZyJGHTN0d2HglBV3uiguA4I"
  }
}
```

Figure 9: JWT Claims Set with a JWK SHA-256 Thumbprint Confirmation

6.2. JWK Thumbprint Confirmation Method in Token Introspection

"OAuth 2.0 Token Introspection" [RFC7662] defines a method for a protected resource to query an authorization server about the active state of an access token. The protected resource also determines metainformation about the token.

For a DPoP-bound access token, the hash of the public key to which the token is bound is conveyed to the protected resource as metainformation in a token introspection response. The hash is conveyed using the same `cnf` content with `jkt` member structure as the JWK Thumbprint

confirmation method, described in Section 6.1, as a top-level member of the introspection response JSON. Note that the resource server does not send a DPoP proof with the introspection request, and the authorization server does not validate an access token's DPoP binding at the introspection endpoint. Rather, the resource server uses the data of the introspection response to validate the access token binding itself locally.

If the `token_type` member is included in the introspection response, it **MUST** contain the value `DPoP`.

The example introspection request in Figure 10 and corresponding response in Figure 11 illustrate an introspection exchange for the example DPoP-bound access token that was issued in Figure 6.

```
POST /as/introspect.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic cnM6cnM6TWt1LTZnX2xDektJZHo0ZnNON2tZY3lhK1Rp

token=Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU
```

Figure 10: Example Introspection Request

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "active": true,
  "sub": "someone@example.com",
  "iss": "https://server.example.com",
  "nbf": 1562262611,
  "exp": 1562266216,
  "cnf":
  {
    "jkt": "0ZcOCORZNYy-DWpqq30jZyJGHTN0d2HglBV3uiguA4I"
  }
}
```

Figure 11: Example Introspection Response for a DPoP-Bound Access Token

7. Protected Resource Access

Requests to DPoP-protected resources **MUST** include both a DPoP proof as per Section 4 and the access token as described in Section 7.1. The DPoP proof **MUST** include the `ath` claim with a valid hash of the associated access token.

Binding the token value to the proof in this way prevents a proof to be used with multiple different access token values across different requests. For example, if a client holds tokens bound to two different resource owners, AT1 and AT2, and uses the same key when talking to the

authorization server, it's possible that these tokens could be swapped. Without the `ath` field to bind it, a captured signature applied to AT1 could be replayed with AT2 instead, changing the rights and access of the intended request. This same substitution prevention remains for rotated access tokens within the same combination of client and resource owner -- a rotated token value would require the calculation of a new proof. This binding additionally ensures that a proof intended for use with the access token is not usable without an access token, or vice-versa.

The resource server is required to calculate the hash of the token value presented and verify that it is the same as the hash value in the `ath` field as described in Section 4.3. Since the `ath` field value is covered by the DPoP proof's signature, its inclusion binds the access token value to the holder of the key used to generate the signature.

Note that the `ath` field alone does not prevent replay of the DPoP proof or provide binding to the request in which the proof is presented, and it is still important to check the time window of the proof as well as the included message parameters, such as `htm` and `htu`.

7.1. The DPoP Authentication Scheme

A DPoP-bound access token is sent using the `Authorization` request header field per Section 11.6.2 of [RFC9110] with an authentication scheme of `DPoP`. The syntax of the `Authorization` header field for the `DPoP` scheme uses the token68 syntax defined in Section 11.2 of [RFC9110] for credentials and is repeated below for ease of reference. The ABNF notation syntax for DPoP authentication scheme credentials is as follows:

```
token68    = 1*( ALPHA / DIGIT /
                 "-" / "." / "_" / "~" / "+" / "/" ) *"="

credentials = "DPoP" 1*SP token68
```

Figure 12: DPoP Authentication Scheme ABNF

For such an access token, a resource server **MUST** check that a DPoP proof was also received in the `DPoP` header field of the HTTP request, check the DPoP proof according to the rules in Section 4.3, and check that the public key of the DPoP proof matches the public key to which the access token is bound per Section 6.

The resource server **MUST NOT** grant access to the resource unless all checks are successful.

Figure 13 shows an example request to a protected resource with a DPoP-bound access token in the `Authorization` header and the DPoP proof in the `DPoP` header. The example uses "\" line wrapping per [RFC8792]. Figure 14 shows the decoded content of that DPoP proof. The JSON of the JWT header and payload are shown, but the signature part is omitted. As usual, line breaks and indentation are included for formatting and readability.

```
GET /protectedresource HTTP/1.1
Host: resource.example.org
Authorization: DPoP Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik\
 VDIiwieCI6Imw4dEZyaHgtMzR0VjNoUklDUkRZOXpDa0RscEJoRjQyVVFVZ1dWQVdCR\
 nMiLCJ5IjoiOVZFNGpmX09rX282NHpiVFRsY3VOSmFqSG10NnY5VERWclUwQ2R2R1JE\
 QSIsImNydiI6IlAtMjU2In19.eyJqdGkiOiJlMWozVl9iS2ljOC1MQUVCIiwiaHRtIj\
 oiR0VUIiwiaHR1IjoiaHR0cHM6Ly9yZXNvdXJjZS5leGFtcGxlLm9yZy9wcm90ZWN0Z\
 WRyZXNvdXJjZSIsImlhdCI6MTU2MjI2MjYxOCwiYXRoIjoiZlVIeU8ycjJaM0RaNTNF\
 c05yV0JiMHhXWG9hTnk1OUlpS0NBcWtzbVFFbyJ9.2oW9RP35yRqzhrtNP86L-Ey71E\
 OptxRimPPToA1plemAgR6pxHF8y6-yqyVnmcw6Fy1dqd-jfxSYoMxhAJpLjA
```

Figure 13: DPoP-Protected Resource Request

```
{
  "typ":"dpop+jwt",
  "alg":"ES256",
  "jwk": {
    "kty":"EC",
    "x":"l8tFrhx-34tV3hRICRDY9zCkDlpBhF42UQUfWVAWBFs",
    "y":"9VE4jf_Ok_o64zbTTlcuNJajHmt6v9TDVrU0CdvGRDA",
    "crv":"P-256"
  }
}
.
{
  "jti":"e1j3V_bKic8-LAEB",
  "htm":"GET",
  "htu":"https://resource.example.org/protectedresource",
  "iat":1562262618,
  "ath":"fUHyO2r2Z3DZ53EsNrWBb0xWXoaNy59IiKCAqksmQEo"
}
```

Figure 14: Decoded Content of the DPoP Proof JWT in Figure 13

Upon receipt of a request to a protected resource within the protection space requiring DPoP authentication, the server can respond with a challenge to the client to provide DPoP authentication information if the request does not include valid credentials or does not contain an access token sufficient for access. Such a challenge is made using the 401 (Unauthorized) response status code ([RFC9110], Section 15.5.2) and the `WWW-Authenticate` header field ([RFC9110], Section 11.6.1). The server **MAY** include the `WWW-Authenticate` header in response to other conditions as well.

In such challenges:

- The scheme name is `DPoP`.
- The authentication parameter `realm` **MAY** be included to indicate the scope of protection in the manner described in [RFC9110], Section 11.5.
- A `scope` authentication parameter **MAY** be included as defined in [RFC6750], Section 3.

- An `error` parameter ([RFC6750], Section 3) **SHOULD** be included to indicate the reason why the request was declined, if the request included an access token but failed authentication. The error parameter values described in [RFC6750], Section 3.1 are suitable, as are any appropriate values defined by extension. The value `use_dpop_nonce` can be used as described in Section 9 to signal that a nonce is needed in the DPoP proof of a subsequent request(s). Additionally, `invalid_dpop_proof` is used to indicate that the DPoP proof itself was deemed invalid based on the criteria of Section 4.3.
- An `error_description` parameter ([RFC6750], Section 3) **MAY** be included along with the `error` parameter to provide developers a human-readable explanation that is not meant to be displayed to end-users.
- An `algs` parameter **SHOULD** be included to signal to the client the JWS algorithms that are acceptable for the DPoP proof JWT. The value of the parameter is a space-delimited list of JWS `alg` (Algorithm) header values ([RFC7515], Section 4.1.1).
- Additional authentication parameters **MAY** be used, and unknown parameters **MUST** be ignored by recipients.

Figure 15 shows a response to a protected resource request without authentication.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP algs="ES256 PS256"
```

Figure 15: HTTP 401 Response to a Protected Resource Request without Authentication

Figure 16 shows a response to a protected resource request that was rejected due to the failed confirmation of the DPoP binding in the access token. Figure 16 uses "\" line wrapping per [RFC8792].

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP error="invalid_token", \
   error_description="Invalid DPoP key binding", algs="ES256"
```

Figure 16: HTTP 401 Response to a Protected Resource Request with an Invalid Token

Note that browser-based client applications using Cross-Origin Resource Sharing (CORS) [WHATWG.Fetch] only have access to CORS-safelisted response HTTP headers by default. In order for the application to obtain and use the `WWW-Authenticate` HTTP response header value, the server needs to make it available to the application by including `WWW-Authenticate` in the `Access-Control-Expose-Headers` response header list value.

This authentication scheme is for origin-server authentication only. Therefore, this authentication scheme **MUST NOT** be used with the `Proxy-Authenticate` or `Proxy-Authorization` header fields.

Note that the syntax of the `Authorization` header field for this authentication scheme follows the usage of the `Bearer` scheme defined in Section 2.1 of [RFC6750]. While it is not the preferred credential syntax of [RFC9110], it is compatible with the general authentication framework therein and is used for consistency and familiarity with the `Bearer` scheme.

7.2. Compatibility with the Bearer Authentication Scheme

Protected resources simultaneously supporting both the `DPoP` and `Bearer` schemes need to update how the evaluation process is performed for bearer tokens to prevent downgraded usage of a DPoP-bound access token. Specifically, such a protected resource **MUST** reject a DPoP-bound access token received as a bearer token per [RFC6750].

Section 11.6.1 of [RFC9110] allows a protected resource to indicate support for multiple authentication schemes (i.e., `Bearer` and `DPoP`) with the `WWW-Authenticate` header field of a 401 (Unauthorized) response.

A protected resource that supports only [RFC6750] and is unaware of DPoP would most presumably accept a DPoP-bound access token as a bearer token (JWT [RFC7519] says to ignore unrecognized claims, Introspection [RFC7662] says that other parameters might be present while placing no functional requirements on their presence, and [RFC6750] is effectively silent on the content of the access token since it relates to validity). As such, a client can send a DPoP-bound access token using the `Bearer` scheme upon receipt of a `WWW-Authenticate: Bearer` challenge from a protected resource (or it can send a DPoP-bound access token if it has prior knowledge of the capabilities of the protected resource). The effect of this likely simplifies the logistics of phased upgrades to protected resources in their support DPoP or prolonged deployments of protected resources with mixed token type support.

If a protected resource supporting both `Bearer` and `DPoP` schemes elects to respond with multiple `WWW-Authenticate` challenges, attention should be paid to which challenge(s) should deliver the actual error information. It is **RECOMMENDED** that the following rules be adhered to:

- If no authentication information has been included with the request, then the challenges **SHOULD NOT** include an error code or other error information, as per Section 3.1 of [RFC6750] (Figure 17).
- If the mechanism used to attempt authentication could be established unambiguously, then the corresponding challenge **SHOULD** be used to deliver error information (Figure 18).
- Otherwise, both `Bearer` and `DPoP` challenges **MAY** be used to deliver error information (Figure 19).

The following examples use "\" line wrapping per [RFC8792].

```
GET /protectedresource HTTP/1.1
Host: resource.example.org

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer, DPoP algs="ES256 PS256"
```

Figure 17: HTTP 401 Response to a Protected Resource Request without Authentication

```
GET /protectedresource HTTP/1.1
Host: resource.example.org
Authorization: Bearer INVALID_TOKEN

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer error="invalid_token", \
    error_description="Invalid token", DPoP algs="ES256 PS256"
```

Figure 18: HTTP 401 Response to a Protected Resource Request with Invalid Authentication

```
GET /protectedresource HTTP/1.1
Host: resource.example.org
Authorization: Bearer Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU
Authorization: DPoP Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU

HTTP/1.1 400 Bad Request
WWW-Authenticate: Bearer error="invalid_request", \
 error_description="Multiple methods used to include access token", \
 DPoP algs="ES256 PS256", error="invalid_request", \
 error_description="Multiple methods used to include access token"
```

Figure 19: HTTP 400 Response to a Protected Resource Request with Ambiguous Authentication

7.3. Client Considerations

Authorization including a DPoP proof may not be idempotent (depending on server enforcement of `jti`, `iat`, and `nonce` claims). Consequently, all previously idempotent requests for protected resources that were previously idempotent may no longer be idempotent. It is **RECOMMENDED** that clients generate a unique DPoP proof, even when retrying idempotent requests in response to HTTP errors generally understood as transient.

Clients that encounter frequent network errors may experience additional challenges when interacting with servers with stricter nonce validation implementations.

8. Authorization Server-Provided Nonce

This section specifies a mechanism using opaque nonces provided by the server that can be used to limit the lifetime of DPoP proofs. Without employing such a mechanism, a malicious party controlling the client (potentially including the end-user) can create DPoP proofs for use arbitrarily far in the future.

Including a nonce value contributed by the authorization server in the DPoP proof **MAY** be used by authorization servers to limit the lifetime of DPoP proofs. The server determines when to issue a new DPoP nonce challenge and if it is needed, thereby requiring the use of the nonce value in subsequent DPoP proofs. The logic through which the server makes that determination is out of scope of this document.

An authorization server **MAY** supply a nonce value to be included by the client in DPoP proofs sent. In this case, the authorization server responds to requests that do not include a nonce with an HTTP 400 (Bad Request) error response per Section 5.2 of [RFC6749] using `use_dpop_nonce` as the error code value. The authorization server includes a `DPoP-Nonce` HTTP header in the response supplying a nonce value to be used when sending the subsequent request. Nonce values **MUST** be unpredictable. This same error code is used when supplying a new nonce value when there was a nonce mismatch. The client will typically retry the request with the new nonce value supplied upon receiving a `use_dpop_nonce` error with an accompanying nonce value.

For example, in response to a token request without a nonce when the authorization server requires one, the authorization server can respond with a `DPoP-Nonce` value such as the following to provide a nonce value to include in the DPoP proof:

```
HTTP/1.1 400 Bad Request
DPoP-Nonce: eyJ7S_zG.eyJH0-Z.HX4w-7v

{
 "error": "use_dpop_nonce",
 "error_description":
   "Authorization server requires nonce in DPoP proof"
}
```

Figure 20: HTTP 400 Response to a Token Request without a Nonce

Other HTTP headers and JSON fields **MAY** also be included in the error response, but there **MUST NOT** be more than one `DPoP-Nonce` header.

Upon receiving the nonce, the client is expected to retry its token request using a DPoP proof including the supplied nonce value in the `nonce` claim of the DPoP proof. An example unencoded JWT payload of such a DPoP proof including a nonce is shown below.

```
{
 "jti": "-BwC3ESc6acc2lTc",
 "htm": "POST",
 "htu": "https://server.example.com/token",
 "iat": 1562262616,
 "nonce": "eyJ7S_zG.eyJH0-Z.HX4w-7v"
}
```

Figure 21: DPoP Proof Payload including a Nonce Value

The nonce is opaque to the client.

If the `nonce` claim in the DPoP proof does not exactly match a nonce recently supplied by the authorization server to the client, the authorization server **MUST** reject the request. The rejection response **MAY** include a `DPoP-Nonce` HTTP header providing a new nonce value to use for subsequent requests.

The intent is that clients need to keep only one nonce value and servers need to keep a window of recent nonces. That said, transient circumstances may arise in which the stored nonce values for the server and the client differ. However, this situation is self-correcting. With any rejection message, the server can send the client the nonce value it wants to use to the client, and the client can store that nonce value and retry the request with it. Even if the client and/or server discard their stored nonce values, that situation is also self-correcting because new nonce values can be communicated when responding to or retrying failed requests.

Note that browser-based client applications using CORS [WHATWG.Fetch] only have access to CORS-safelisted response HTTP headers by default. In order for the application to obtain and use the `DPoP-Nonce` HTTP response header value, the server needs to make it available to the application by including `DPoP-Nonce` in the `Access-Control-Expose-Headers` response header list value.

8.1. Nonce Syntax

The nonce syntax in ABNF as used by [RFC6749] (which is the same as the scope-token syntax) is shown below.

```
nonce = 1*NQCHAR
```

Figure 22: Nonce ABNF

8.2. Providing a New Nonce Value

It is up to the authorization server when to supply a new nonce value for the client to use. The client is expected to use the existing supplied nonce in DPoP proofs until the server supplies a new nonce value.

The authorization server **MAY** supply the new nonce in the same way that the initial one was supplied: by using a `DPoP-Nonce` HTTP header in the response. The `DPoP-Nonce` HTTP header field uses the nonce syntax defined in Section 8.1. Each time this happens, it requires an extra protocol round trip.

A more efficient manner of supplying a new nonce value is also defined by including a `DPoP-Nonce` HTTP header in the HTTP 200 (OK) response from the previous request. The client **MUST** use the new nonce value supplied for the next token request and for all subsequent token requests until the authorization server supplies a new nonce.

Responses that include the `DPoP-Nonce` HTTP header should be uncacheable (e.g., using `Cache-Control: no-store` in response to a `GET` request) to prevent the response from being used to serve a subsequent request and a stale nonce value from being used as a result.

An example 200 OK response providing a new nonce value is shown below.

```
HTTP/1.1 200 OK
Cache-Control: no-store
DPoP-Nonce: eyJ7S_zG.eyJbYu3.xQmBj-1
```

Figure 23: HTTP 200 Response Providing the Next Nonce Value

9. Resource Server-Provided Nonce

Resource servers can also choose to provide a nonce value to be included in DPoP proofs sent to them. They provide the nonce using the `DPoP-Nonce` header in the same way that authorization servers do as described in Sections 8 and 8.2. The error signaling is performed as described in Section 7.1. Resource servers use an HTTP 401 (Unauthorized) error code with an accompanying `WWW-Authenticate: DPoP` value and `DPoP-Nonce` value to accomplish this.

For example, in response to a resource request without a nonce when the resource server requires one, the resource server can respond with a `DPoP-Nonce` value such as the following to provide a nonce value to include in the DPoP proof. The example below uses "\" line wrapping per [RFC8792].

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP error="use_dpop_nonce", \
  error_description="Resource server requires nonce in DPoP proof"
DPoP-Nonce: eyJ7S_zG.eyJH0-Z.HX4w-7v
```

Figure 24: HTTP 401 Response to a Resource Request without a Nonce

Note that the nonces provided by an authorization server and a resource server are different and should not be confused with one another since nonces will be only accepted by the server that issued them. Likewise, should a client use multiple authorization servers and/or resource

servers, a nonce issued by any of them should be used only at the issuing server. Developers should also be careful to not confuse DPoP nonces with the OpenID Connect [OpenID.Core] ID Token nonce.

10. Authorization Code Binding to a DPoP Key

Binding the authorization code issued to the client's proof-of-possession key can enable end-to-end binding of the entire authorization flow. This specification defines the `dpop_jkt` authorization request parameter for this purpose. The value of the `dpop_jkt` authorization request parameter is the JWK Thumbprint [RFC7638] of the proof-of-possession public key using the SHA-256 hash function, which is the same value as used for the `jkt` confirmation method defined in Section 6.1.

When a token request is received, the authorization server computes the JWK Thumbprint of the proof-of-possession public key in the DPoP proof and verifies that it matches the `dpop_jkt` parameter value in the authorization request. If they do not match, it **MUST** reject the request.

An example authorization request using the `dpop_jkt` authorization request parameter is shown below and uses "\" line wrapping per [RFC8792].

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz\
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb\
    &code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM\
    &code_challenge_method=S256\
    &dpop_jkt=NzbLsXh8uDCcd-6MNwXF4W_7noWXFZAfHkxZsRGC9Xs HTTP/1.1
Host: server.example.com
```

Figure 25: Authorization Request Using the `dpop_jkt` Parameter

Use of the `dpop_jkt` authorization request parameter is **OPTIONAL**. Note that the `dpop_jkt` authorization request parameter **MAY** also be used in combination with Proof Key for Code Exchange (PKCE) [RFC7636], which is recommended by [SECURITY-TOPICS] as a countermeasure to authorization code injection. The `dpop_jkt` authorization request parameter only provides similar protections when a unique DPoP key is used for each authorization request.

10.1. DPoP with Pushed Authorization Requests

When Pushed Authorization Requests (PARs) [RFC9126] are used in conjunction with DPoP, there are two ways in which the DPoP key can be communicated in the PAR request:

- The `dpop_jkt` parameter can be used as described in Section 10 to bind the issued authorization code to a specific key. In this case, `dpop_jkt` **MUST** be included alongside other authorization request parameters in the POST body of the PAR request.
- Alternatively, the `DPoP` header can be added to the PAR request. In this case, the authorization server **MUST** check the provided DPoP proof JWT as defined in Section 4.3. It **MUST** further behave as if the contained public key's thumbprint was provided using `dpop_jkt`, i.e., reject the subsequent token request unless a DPoP proof for the same key is

provided. This can help to simplify the implementation of the client, as it can "blindly" attach the `DPoP` header to all requests to the authorization server regardless of the type of request. Additionally, it provides a stronger binding, as the `DPoP` header contains a proof of possession of the private key.

Both mechanisms **MUST** be supported by an authorization server that supports PAR and DPoP. If both mechanisms are used at the same time, the authorization server **MUST** reject the request if the JWK Thumbprint in `dpop_jkt` does not match the public key in the `DPoP` header.

Allowing both mechanisms ensures that clients using `dpop_jkt` do not need to distinguish between front-channel and pushed authorization requests, and at the same time, clients that only have one code path for protecting all calls to authorization server endpoints do not need to distinguish between requests to the PAR endpoint and the token endpoint.

11. Security Considerations

In DPoP, the prevention of token replay at a different endpoint (see Section 2) is achieved through authentication of the server per [RFC6125] and the binding of the DPoP proof to a certain URI and HTTP method. However, DPoP has a somewhat different nature of protection than TLS-based methods such as OAuth Mutual TLS [RFC8705] or OAuth Token Binding [TOKEN-BINDING] (see also Sections 11.1 and 11.7). TLS-based mechanisms can leverage a tight integration between the TLS layer and the application layer to achieve strong message integrity, authenticity, and replay protection.

11.1. DPoP Proof Replay

If an adversary is able to get hold of a DPoP proof JWT, the adversary could replay that token at the same endpoint (the HTTP endpoint and method are enforced via the respective claims in the JWTs). To limit this, servers **MUST** only accept DPoP proofs for a limited time after their creation (preferably only for a relatively brief period on the order of seconds or minutes).

In the context of the target URI, servers can store the `jti` value of each DPoP proof for the time window in which the respective DPoP proof JWT would be accepted to prevent multiple uses of the same DPoP proof. HTTP requests to the same URI for which the `jti` value has been seen before would be declined. When strictly enforced, such a single-use check provides a very strong protection against DPoP proof replay, but it may not always be feasible in practice, e.g., when multiple servers behind a single endpoint have no shared state.

In order to guard against memory exhaustion attacks, a server that is tracking `jti` values should reject DPoP proof JWTs with unnecessarily large `jti` values or store only a hash thereof.

Note: To accommodate for clock offsets, the server **MAY** accept DPoP proofs that carry an `iat` time in the reasonably near future (on the order of seconds or minutes). Because clock skews between servers and clients may be large, servers **MAY** limit DPoP proof lifetimes by using server-provided nonce values containing the time at the server rather than comparing the client-supplied `iat` time to the time at the server. Nonces created in this way yield the same result even in the face of arbitrarily large clock skews.

Server-provided nonces are an effective means for further reducing the chances for successful DPoP proof replay. Unlike cryptographic nonces, it is acceptable for clients to use the same `nonce` multiple times and for the server to accept the same nonce multiple times. As long as the `jti` value is tracked and duplicates are rejected for the lifetime of the `nonce`, there is no additional risk of token replay.

11.2. DPoP Proof Pre-generation

An attacker in control of the client can pre-generate DPoP proofs for specific endpoints arbitrarily far into the future by choosing the `iat` value in the DPoP proof to be signed by the proof-of-possession key. Note that one such attacker is the person who is the legitimate user of the client. The user may pre-generate DPoP proofs to exfiltrate from the machine possessing the proof-of-possession key upon which they were generated and copy them to another machine that does not possess the key. For instance, a bank employee might pre-generate DPoP proofs on a bank computer and then copy them to another machine for use in the future, thereby bypassing bank audit controls. When DPoP proofs can be pre-generated and exfiltrated, all that is actually being proved in DPoP protocol interactions is possession of a DPoP proof -- not of the proof-of-possession key.

Use of server-provided nonce values that are not predictable by attackers can prevent this attack. By providing new nonce values at times of its choosing, the server can limit the lifetime of DPoP proofs, preventing pre-generated DPoP proofs from being used. When server-provided nonces are used, possession of the proof-of-possession key is being demonstrated -- not just possession of a DPoP proof.

The `ath` claim limits the use of pre-generated DPoP proofs to the lifetime of the access token. Deployments that do not utilize the nonce mechanism **SHOULD NOT** issue long-lived DPoP constrained access tokens, preferring instead to use short-lived access tokens and refresh tokens. Whilst an attacker could pre-generate DPoP proofs to use the refresh token to obtain a new access token, they would be unable to realistically pre-generate DPoP proofs to use a newly issued access token.

11.3. DPoP Nonce Downgrade

A server **MUST NOT** accept any DPoP proofs without the `nonce` claim when a DPoP nonce has been provided to the client.

11.4. Untrusted Code in the Client Context

If an adversary is able to run code in the client's execution context, the security of DPoP is no longer guaranteed. Common issues in web applications leading to the execution of untrusted code are XSS and remote code inclusion attacks.

If the private key used for DPoP is stored in such a way that it cannot be exported, e.g., in a hardware or software security module, the adversary cannot exfiltrate the key and use it to create arbitrary DPoP proofs. The adversary can, however, create new DPoP proofs as long as the

client is online and uses these proofs (together with the respective tokens) either on the victim's device or on a device under the attacker's control to send arbitrary requests that will be accepted by servers.

To send requests even when the client is offline, an adversary can try to pre-compute DPoP proofs using timestamps in the future and exfiltrate these together with the access or refresh token.

An adversary might further try to associate tokens issued from the token endpoint with a key pair under the adversary's control. One way to achieve this is to modify existing code, e.g., by replacing cryptographic APIs. Another way is to launch a new authorization grant between the client and the authorization server in an iframe. This grant needs to be "silent", i.e., not require interaction with the user. With code running in the client's origin, the adversary has access to the resulting authorization code and can use it to associate their own DPoP keys with the tokens returned from the token endpoint. The adversary is then able to use the resulting tokens on their own device even if the client is offline.

Therefore, protecting clients against the execution of untrusted code is extremely important even if DPoP is used. Besides secure coding practices, Content Security Policy [W3C.CSP] can be used as a second layer of defense against XSS.

11.5. Signed JWT Swapping

Servers accepting signed DPoP proof JWTs **MUST** verify that the `typ` field is `dpop+jwt` in the headers of the JWTs to ensure that adversaries cannot use JWTs created for other purposes.

11.6. Signature Algorithms

Implementers **MUST** ensure that only asymmetric digital signature algorithms (such as `ES256`) that are deemed secure can be used for signing DPoP proofs. In particular, the algorithm `none` **MUST NOT** be allowed.

11.7. Request Integrity

DPoP does not ensure the integrity of the payload or headers of requests. The DPoP proof only contains claims for the HTTP URI and method, but not the message body or general request headers, for example.

This is an intentional design decision intended to keep DPoP simple to use, but as described, it makes DPoP potentially susceptible to replay attacks where an attacker is able to modify message contents and headers. In many setups, the message integrity and confidentiality provided by TLS is sufficient to provide a good level of protection.

Note: While signatures covering other parts of requests are out of the scope of this specification, additional information to be signed can be added into DPoP proofs.

11.8. Access Token and Public Key Binding

The binding of the access token to the DPoP public key, as specified in Section 6, uses a cryptographic hash of the JWK representation of the public key. It relies on the hash function having sufficient second-preimage resistance so as to make it computationally infeasible to find or create another key that produces to the same hash output value. The SHA-256 hash function was used because it meets the aforementioned requirement while being widely available.

Similarly, the binding of the DPoP proof to the access token uses a hash of that access token as the value of the `ath` claim in the DPoP proof (see Section 4.2). This relies on the value of the hash being sufficiently unique so as to reliably identify the access token. The collision resistance of SHA-256 meets that requirement.

11.9. Authorization Code and Public Key Binding

Cryptographic binding of the authorization code to the DPoP public key is specified in Section 10. This binding prevents attacks in which the attacker captures the authorization code and creates a DPoP proof using a proof-of-possession key other than the one held by the client and redeems the authorization code using that DPoP proof. By ensuring end to end that only the client's DPoP key can be used, this prevents captured authorization codes from being exfiltrated and used at locations other than the one to which the authorization code was issued.

Authorization codes can, for instance, be harvested by attackers from places where the HTTP messages containing them are logged. Even when efforts are made to make authorization codes one-time-use, in practice, there is often a time window during which attackers can replay them. For instance, when authorization servers are implemented as scalable replicated services, some replicas may temporarily not yet have the information needed to prevent replay. DPoP binding of the authorization code solves these problems.

If an authorization server does not (or cannot) strictly enforce the single-use limitation for authorization codes and an attacker can access the authorization code (and if PKCE is used, the `code_verifier`), the attacker can create a forged token request, binding the resulting token to an attacker-controlled key. For example, using XSS, attackers might obtain access to the authorization code and PKCE parameters. Use of the `dpop_jkt` parameter prevents this attack.

The binding of the authorization code to the DPoP public key uses a JWK Thumbprint of the public key, just as the access token binding does. The same JWK Thumbprint considerations apply.

11.10. Hash Algorithm Agility

The `jkt` confirmation method member, the `ath` JWT claim, and the `dpop_jkt` authorization request parameter defined herein all use the output of the SHA-256 hash function as their value. The use of a single hash function by this specification was intentional and aimed at simplicity and avoidance of potential security and interoperability issues arising from common mistakes implementing and deploying parameterized algorithm agility schemes. However, the use of a

different hash function is not precluded if future circumstances change and make SHA-256 insufficient for the requirements of this specification. Should that need arise, it is expected that a short specification will be produced that updates this one. Using the output of an appropriate hash function as the value, that specification will likely define a new confirmation method member, a new JWT claim, and a new authorization request parameter. These items will be used in place of, or alongside, their respective counterparts in the same message structures and flows of the larger protocol defined by this specification.

11.11. Binding to Client Identity

In cases where DPoP is used with client authentication, it is only bound to authentication by being coincident in the same TLS tunnel. Since the DPoP proof is not directly bound to the authentication cryptographically, it's possible that the authentication or the DPoP messages were copied into the tunnel. While including the URI in the DPoP can partially mitigate some of this risk, modifying the authentication mechanism to provide cryptographic binding between authentication and DPoP could provide better protection. However, providing additional binding with authentication through the modification of authentication mechanisms or other means is beyond the scope of this specification.

12. IANA Considerations

12.1. OAuth Access Token Types Registration

IANA has registered the following access token type in the "OAuth Access Token Types" registry [IANA.OAuth.Params] established by [RFC6749].

Name: `DPoP`

Additional Token Endpoint Response Parameters: (none)

HTTP Authentication Scheme(s): `DPoP`

Change Controller: IETF

Reference: RFC 9449

12.2. OAuth Extensions Error Registration

IANA has registered the following error values in the "OAuth Extensions Error" registry [IANA.OAuth.Params] established by [RFC6749].

Invalid DPoP proof:

Name: `invalid_dpop_proof`

Usage Location: token error response, resource access error response

Protocol Extension: Demonstrating Proof of Possession (DPoP)

Change Controller: IETF

Reference: RFC 9449

Use DPoP nonce:

Name: `use_dpop_nonce`

Usage Location: token error response, resource access error response

Protocol Extension: Demonstrating Proof of Possession (DPoP)

Change Controller: IETF

Reference: RFC 9449

12.3. OAuth Parameters Registration

IANA has registered the following authorization request parameter in the "OAuth Parameters" registry [IANA.OAuth.Params] established by [RFC6749].

Name: `dpop_jkt`

Parameter Usage Location: authorization request

Change Controller: IETF

Reference: Section 10 of RFC 9449

12.4. HTTP Authentication Schemes Registration

IANA has registered the following scheme in the "HTTP Authentication Schemes" registry [IANA.HTTP.AuthSchemes] established by [RFC9110], Section 16.4.1.

Authentication Scheme Name: `DPoP`

Reference: Section 7.1 of RFC 9449

12.5. Media Type Registration

IANA has registered the `application/dpop+jwt` media type [RFC2046] in the "Media Types" registry [IANA.MediaTypes] in the manner described in [RFC6838], which is used to indicate that the content is a DPoP JWT.

Type name: application

Subtype name: dpop+jwt

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: binary. A DPoP JWT is a JWT; JWT values are encoded as a series of base64url-encoded values (some of which may be the empty string) separated by period ('.') characters.

Security considerations: See Section 11 of RFC 9449

Interoperability considerations: n/a

Published specification: RFC 9449

Applications that use this media type: Applications using RFC 9449 for application-level proof of possession

Fragment identifier considerations: n/a

Additional information:

File extension(s): n/a
Macintosh file type code(s): n/a

Person & email address to contact for further information: Michael B. Jones, michael_b_jones@hotmail.com

Intended usage: COMMON

Restrictions on usage: none

Author: Michael B. Jones, michael_b_jones@hotmail.com

Change controller: IETF

12.6. JWT Confirmation Methods Registration

IANA has registered the following JWT `cnf` member value in the "JWT Confirmation Methods" registry [IANA.JWT] established by [RFC7800].

Confirmation Method Value: `jkt`

Confirmation Method Description: JWK SHA-256 Thumbprint

Change Controller: IETF

Reference: Section 6 of RFC 9449

12.7. JSON Web Token Claims Registration

IANA has registered the following Claims in the "JSON Web Token Claims" registry [IANA.JWT] established by [RFC7519].

HTTP method:

Claim Name: `htm`

Claim Description: The HTTP method of the request

Change Controller: IETF

Reference: Section 4.2 of RFC 9449

HTTP URI:

Claim Name: `htu`

Claim Description: The HTTP URI of the request (without query and fragment parts)

Change Controller: IETF

Reference: Section 4.2 of RFC 9449

Access token hash:

Claim Name: `ath`

Claim Description: The base64url-encoded SHA-256 hash of the ASCII encoding of the associated access token's value

Change Controller: IETF

Reference: Section 4.2 of RFC 9449

12.7.1. "nonce" Registration Update

The Internet Security Glossary [RFC4949] provides a useful definition of nonce as a random or non-repeating value that is included in data exchanged by a protocol, usually for the purpose of guaranteeing liveness and thus detecting and protecting against replay attacks.

However, the initial registration of the `nonce` claim by [OpenID.Core] used language that was contextually specific to that application, which was potentially limiting to its general applicability.

Therefore, IANA has updated the entry for `nonce` in the "JSON Web Token Claims" registry [IANA.JWT] with an expanded definition to reflect that the claim can be used appropriately in other contexts and with the addition of this document as a reference, as follows.

Claim Name: `nonce`

Claim Description: Value used to associate a Client session with an ID Token (**MAY** also be used for nonce values in other applications of JWTs)

Change Controller: OpenID Foundation Artifact Binding Working Group, openid-specs-ab@lists.openid.net

Specification Document(s): Section 2 of [OpenID.Core] and RFC 9449

12.8. Hypertext Transfer Protocol (HTTP) Field Name Registration

IANA has registered the following HTTP header fields, as specified by this document, in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" [IANA.HTTP.Fields] established by [RFC9110]:

DPoP:

Field Name: `DPoP`

Status: permanent

Reference: RFC 9449

DPoP-Nonce:

Field Name: `DPoP-Nonce`

Status: permanent

Reference: RFC 9449

12.9. OAuth Authorization Server Metadata Registration

IANA has registered the following value in the "OAuth Authorization Server Metadata" registry [IANA.OAuth.Params] established by [RFC8414].

Metadata Name: `dpop_signing_alg_values_supported`

Metadata Description: JSON array containing a list of the JWS algorithms supported for DPoP proof JWTs

Change Controller: IETF

Reference: Section 5.1 of RFC 9449

12.10. OAuth Dynamic Client Registration Metadata

IANA has registered the following value in the IANA "OAuth Dynamic Client Registration Metadata" registry [IANA.OAuth.Params] established by [RFC7591].

Client Metadata Name: `dpop_bound_access_tokens`

Client Metadata Description: Boolean value specifying whether the client always uses DPoP for token requests

Change Controller: IETF

Reference: Section 5.2 of RFC 9449

13. References

13.1. Normative References

[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/info/rfc2119>.

[RFC3986] 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/info/rfc3986>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <https://www.rfc-editor.org/info/rfc6125>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7638] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <https://www.rfc-editor.org/info/rfc7638>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[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/info/rfc8174>.

[SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS.180-4, August 2015, <http://dx.doi.org/10.6028/NIST.FIPS.180-4>.

13.2. Informative References

[BREACH] CVE, "CVE-2013-3587", <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3587>.

[Cloudbleed] Graham-Cumming, J., "Incident report on memory leak caused by Cloudflare parser bug", February 2017, <https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/>.

[CRIME] CVE, "CVE-2012-4929", <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2012-4929>.

[GitHub.Tokens] Hanley, M., "Security alert: Attack campaign involving stolen OAuth user tokens issued to two third-party integrators", April 2022, <https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/>.

[Heartbleed] "CVE-2014-0160", <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160>.

[IANA.HTTP.AuthSchemes] IANA, "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry", <https://www.iana.org/assignments/http-authschemes/>.

[IANA.HTTP.Fields] IANA, "Hypertext Transfer Protocol (HTTP) Field Name Registry", <https://www.iana.org/assignments/http-fields/>.

[IANA.JOSE.ALGS] IANA, "JSON Web Signature and Encryption Algorithms", <https://www.iana.org/assignments/jose/>.

[IANA.JWT] IANA, "JSON Web Token Claims", <https://www.iana.org/assignments/jwt/>.

[IANA.MediaTypes] IANA, "Media Types", <https://www.iana.org/assignments/media-types/>.

[IANA.OAuth.Params] IANA, "OAuth Parameters", <https://www.iana.org/assignments/oauth-parameters/>.

[OpenID.Core] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/info/rfc4949>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <https://www.rfc-editor.org/info/rfc7523>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <https://www.rfc-editor.org/info/rfc7662>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/info/rfc8705>.

[RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, <https://www.rfc-editor.org/info/rfc8707>.

[RFC8725] Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", BCP 225, RFC 8725, DOI 10.17487/RFC8725, February 2020, <https://www.rfc-editor.org/info/rfc8725>.

[RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/info/rfc8792>.

[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/info/rfc9110>.

[RFC9126] Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D., and F. Skokan, "OAuth 2.0 Pushed Authorization Requests", RFC 9126, DOI 10.17487/RFC9126, September 2021, <https://www.rfc-editor.org/info/rfc9126>.

[SECURITY-TOPICS] Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", Work in Progress, Internet-Draft, draft-ietf-oauth-security-topics-23, 5 June 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-23>.

[TOKEN-BINDING] Jones, M., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Token Binding", Work in Progress, Internet-Draft, draft-ietf-oauth-token-binding-08, 19 October 2018, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-token-binding-08>.

[W3C.CSP] West, M., "Content Security Policy Level 3", W3C Working Draft, July 2023, <https://www.w3.org/TR/CSP3/>.

[W3C.WebCryptoAPI] Watson, M., "Web Cryptography API", W3C Recommendation, January 2017, <https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126>.

[WHATWG.Fetch] WHATWG, "Fetch Living Standard", July 2023, <https://fetch.spec.whatwg.org/>.

Acknowledgements

We would like to thank Brock Allen, Annabelle Backman, Dominick Baier, Spencer Balogh, Vittorio Bertocci, Jeff Corrigan, Domingos Creado, Philippe De Ryck, Andrii Deinega, William Denniss, Vladimir Dzhuvinov, Mike Engan, Nikos Fotiou, Mark Haine, Dick Hardt, Joseph Heenan, Bjorn Hjelm, Jacob Ideskog, Jared Jennings, Benjamin Kaduk, Pieter Kasselman, Neil Madden, Rohan Mahy, Karsten Meyer zu Selhausen, Nicolas Mora, Steinar Noem, Mark Nottingham, Rob Otto, Aaron Parecki, Michael Peck, Roberto Polli, Paul Querna, Justin Richer, Joseph Salowey, Rifaat Shekh-Yusef, Filip Skokan, Dmitry Telegin, Dave Tonge, Jim Willeke, and others for their valuable input, feedback, and general support of this work.

This document originated from discussions at the 4th OAuth Security Workshop in Stuttgart, Germany. We thank the organizers of this workshop (Ralf Küsters and Guido Schmitz).

Authors' Addresses

Daniel Fett
Authlete
Email: mail@danielfett.de

Brian Campbell
Ping Identity
Email: bcampbell@pingidentity.com

John Bradley
Yubico
Email: ve7jtb@ve7jtb.com

Torsten Lodderstedt
Tuconic
Email: torsten@lodderstedt.net

Michael Jones
Self-Issued Consulting
Email: michael_b_jones@hotmail.com
URI: https://self-issued.info/

David Waite
Ping Identity
Email: david@alkaline-solutions.com