Workgroup: Independent Submission
Internet-Draft:
draft-thornburgh-fwk-dc-token-iss-00
Published: 19 May 2020
Intended Status: Experimental
Expires: 20 November 2020
Authors: M. Thornburgh
Adobe

A Framework For Decentralized Bearer Token Issuance in HTTP

Abstract

This memo describes a protocol framework for HTTP clients to obtain bearer tokens for accessing restricted resources, where in some applications the client may not have prior knowledge of, or a direct relationship with, the resource server's authorization infrastructure (such as in decentralized identity systems). Semi-concrete applications of the framework using proof-of-possession and TLS client certificate mechanisms are also described.

Author's Note

This work is an independent contribution and is not associated with, or endorsed by, Adobe.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This memo describes a general protocol framework for HTTP clients to obtain bearer tokens (Section 1.2 of [RFC6750]) from a resource server's authorization service in order to access protected resources on the server. This framework is especially intended for systems (such as decentralized identity systems like [WebID], and decentralized social or mashup data systems like the Solid project) where the client might not have prior knowledge of, or a preexisting direct relationship with, the authorization service for the resource server; however, it can be applied in other use cases as well.

The protocol includes a method for the client to discover the nature(s) of principals (such as identities, capabilities, sender-constrained access tokens, or verifiable credentials) that the server expects to interact with, and methods for the client to discover the API endpoint URIs for multiple potential mechanisms for obtaining bearer tokens. The framework is constructed to mitigate man-in-the-middle token-stealing attacks.

This memo defines two mechanisms within the framework for a client to obtain a bearer token: one using a cryptographic proof-of-possession, and one using TLS [RFC8446] client certificates. These mechanisms retain generality, and must be further refined in other specifications according to the application and the nature of the principals expected by the servers. Other mechanisms within the framework are also possible.

1.1. Motivation

This work was originally motivated by a desire to address security, semantic, and operational shortcomings in an experimental, decentralized, application-layer authentication scheme for the Solid project that was based on [WebID], OpenID Connect [OpenID.Core], and proof-of-possession key semantics [RFC7800].

An explicit goal of the solution is to leverage the benefits of bearer tokens for accessing restricted resources:

* The token can encapsulate (by direct encoding or by reference) exactly and only the implementation-specific and deployment-specific properties needed to make access control decisions in the resource server;

* The effort (including computational, cryptographic, and network) required to establish a client's identity and authorizations can be done once by the client and the authorization service, compiled to a token, and this effort amortized over many requests to the same resource server, with simple revalidation and

lifetime semantics that can be influenced by both parties; specifically:

- The server's authorization system chooses an expiration period for the token, and can also revoke it at any time, to cause a reauthentication and revalidation;

- The client can forget the token at any time and acquire a new one to cause a reauthentication and revalidation; this can be particularly advantageous if the client acquires new privileges, authorizations, or endorsements that might otherwise be subject to unknown caching policies in an access controller;

* The representation of the token can be optimized for network transmission and for decoding, verification, and processing according to the server's implementation;

* HTTP header compression schemes such as HPACK [RFC7541] can reduce network resource consumption when a token is reused for multiple requests in the same origin.

As work progressed, a general form emerged that could address multiple use cases beyond the original motivator.

1.1.1. Use Cases

It is envisioned that the framework described in this memo can be used in at least the following cases, with appropriate further specification, to realize the benefits listed above:

* Decentralized identity systems such as WebID and Decentralized Identifiers [DID];

* Centralized or decentralized authorization systems based on Verifiable Credentials [VC];

* Authenticated access to a multitude of decentralized, uncoordinated resource servers, such as for social or mashup data applications;

* Identity systems based on aspects of a TLS client certificate, without requiring use of that certificate for all accesses to a resource server (particularly in browser-based applications, to allow selective unauthenticated access to non-protected resources within the limitations of negotiating client certificates in TLS);

* Obtaining an audience-constrained bearer token given a sender-constrained access credential or capability issued by a central authority;

* Obtaining an audience-constrained bearer token in a centralized, federated, or confederated identity system given an identity bound with a pre-shared public key.

This list of use cases should not be construed as exhaustive or limiting. Other effective applications of this framework are possible.

1.2. Terminology

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

The term "bearer token" in this document has the meaning described in [RFC6750].

The term "protection space" in this document has the meaning described in Section 2.2 of [RFC7235].

2. General Framework

The server challenges an unauthenticated client (Section 2.1 of [RFC7235]) with an HTTP 401 response, including a WWW-Authenticate response header with the Bearer *auth-scheme* (Section 3 of [RFC6750]), and comprising parameters including how to use one or more token acquisition mechanisms. The client examines the challenge and determines which mechanisms, if any, it is able to use to acquire a bearer token. If possible, the client uses a compatible mechanism, including attributes of the original request and the challenge, to request a bearer token. The token will have a stated lifetime and will be valid for accesses within the same protection space as the original request, until the token expires or is revoked.

A WWW-Authenticate challenge for any mechanism includes at least these *auth-params*:

scope **REQUIRED**: A space-delimited list of case-sensitive strings, each a well-known or server-defined value indicating the nature(s) of the principal expected to be used when requesting a bearer token. To avoid ambiguity, server-defined scopes **SHOULD** be URIs.

nonce
REQUIRED: An opaque (to the client) string to be included unmodified when requesting a bearer token. See Section 2.1 for considerations on constructing the challenge nonce.

error If present, a reason code indicating that the request had a problem other than not presenting an access token. The following reason codes are initially defined:

invalid_token A bearer token was presented, but it was expired, revoked, or otherwise not recognized as valid.

proof_required An access token requiring proof-of-possession of a key (but potentially otherwise valid) was presented.

Additionally, one or more mechanism-specific *auth-params* are included in the challenge to indicate the availability of that mechanism and its unique parameters (usually the URI at which to use the mechanism). This memo defines two mechanism-specific *auth-params*:

token_pop_endpoint If present, the Proof-of-Possession mechanism (Section 3) is available. The parameter value is the URI at which to exchange a proof-of-possession for a bearer token.

client_cert_endpoint If present, the TLS Client Certificate mechanism (Section 4) is available. The parameter value is the URI at which to request a bearer token.

The challenge can include other *auth-params* (such as realm), including ones for other mechanisms. Unrecognized *auth-params* **SHOULD** be ignored.

If a request is made for a resource within a protection space and that request includes an Authorization header with an invalid Bearer token, the resource server **SHOULD** reply with an HTTP 401 response and WWW-Authenticate header as above, even if processing the request doesn't otherwise require authorization. This is to allow a client to obtain a fresh bearer token proactively (for example, before the current token expires, to avoid delaying a real request by the user).

2.1. Nonce Considerations

The nonce in the WWW-Authenticate challenge **SHOULD** have the following properties:

* Be cryptographically strong and unguessable;

*Be recognizable when returned in a token request as having been issued for this protection space (for example, by recording the nonce in a database, or including a cryptographic signature);

*Be valid for a limited (short) time;

*Be redeemable at most once;

*Be coupled to the original request URI in a recognizable way.

2.2. Common Token Response

It is anticipated that most mechanisms (especially ones that use an HTTP API) will respond to a token request using a common response format. Both of the mechanisms described in this memo use the common format described in this section, which is substantially the same as the format described in Section 5 of [RFC6749].

A successful common response is an HTTP 200 response with Content-Type application/json, and having a response body in JSON [RFC8259] format encoding a JSON object with at least the following members:

access_token An opaque (to the client) string; a bearer access token (Section 1.1 of [RFC6750]) which can be used for requests in the same protection space as the original request;

expires_in The number of seconds from the Date of this response after which the access_token will no longer be valid;

token_type A case-insensitive string identifying the kind of token returned in this response. This value **MUST** be Bearer.

If there is a problem with the request, the response **SHALL** be an HTTP 400 response with Content-Type application/json, and having a response body in JSON format encoding a JSON object with at least an error member, and others as appropriate, whose keys and values are defined in Section 5.2 of [RFC6749].

Additional members **MAY** be included in a successful or unsuccessful response object depending on the scope(s) from the challenge, the mechanism used, and the implementation. Unrecognized response object members **SHOULD** be ignored.

2.3. Common Mechanism Flow

It is anticipated that most mechanisms will comprise a simple mechanism-specific API endpoint and respond with a Common Response (Section 2.2). The abstract flow for a client to acquire a bearer token in the common way is illustrated in Figure 1.

```
  Client                  Mechanism Endpoint      Resource Server
  |                            |                              |
  |-- request URI --------------------------------------------->|
  |<----------------------------- 401 Bearer nonce, scope, --|
  |                            |      endpoints               |
  |determine compatibility,    |                              |
  |prepare token request       |                              |
  |-- POST token request------->|                              |
  |                            |validate request,             |
  |                            |issue token                   |
  |<--------- Common Response --|                              |
  |                            |                              |
  |                                                           |
  |-- request URI with access_token ------------------------->|
  |                              validate & translate token,|
  |                                    apply access controls|
  |                                                           |
  |<-------------------------------------- answer resource --|
```

Figure 1: Common Protocol Flow Sequence Diagram

Note that the "validate request" step can involve complex operations and include fetching supplemental information from external sources, depending on the semantics of the mechanism, scopes, and principal.

3. Proof-of-Possession Mechanism

The client recognizes the availability of, and its compatibility with, this mechanism, by recognizing combinations of challenge scopes with which it is compatible, the presence of the token_pop_endpoint, and control of an appropriate principal having proof-of-possession semantics (for example, an access token bound to a proof-of-possession key, or a JSON Web Token (JWT) [RFC7519] with a cnf claim [RFC7800]) and compatibility with the same combination of challenge scopes.

The client constructs and signs a *proof-token* (Section 3.1).

The client sends the *proof-token* to the token_pop_endpoint API URI with HTTP POST (Section 3.2). The API endpoint validates the request including the *proof-token*, and if appropriate, it responds with a bearer token.

3.1. Proof Token

The *proof-token* is a JWT [RFC7519], with a signature proving possesion of the key bound to the client's principal, and having the following claims:

sub
REQUIRED: The client's principal (having proof-of-possession semantics and compatible with a combination of the challenge scopes);

aud **REQUIRED**: The absolute URI (Section 4.3 of [RFC3986]), including scheme, authority (host and optional port), path, and query, but not including fragment identifier, corresponding to the original request that resulted in the HTTP 401 challenge; if this claim is an array, it **MUST** have exactly one element;

nonce **REQUIRED**: The nonce from the WWW-Authenticate challenge;

jti **RECOMMENDED**: Use of this claim is recommended so that the client can salt the *proof-token*'s signature; the verifier can ignore this claim, if present;

exp **OPTIONAL**: If present, this claim **MUST NOT** be after the expiration time of the sub (if it has one), and **MUST NOT** be before the current time on the verifier; ordinarily the validity of the nonce is sufficient to establish not-before and not-after constraints on the proof, so this claim isn't usually necessary (and clocks on end-user devices, where *proof-tokens* are likely to be generated, are notoriously inaccurate). The issuer **MAY** take the expiration periods of the *proof-token* and the sub into account when determining the expiration period of the bearer token it issues, but it is not required to do so and is free to issue bearer tokens with any expiration period.

Additional claims can appear in the *proof-token* according to, and conditioned on, the semantics of the scope(s). Unrecognized or incompatible claims **SHOULD** be ignored.

3.2. Proof-of-Possession API

This API endpoint is implemented by the authorization server (Section 1.1 of [RFC6749]) for the protection space of the original request.

The client uses this API by making an HTTP POST request to the token_pop_endpoint URI. The request body has Content-Type application/x-www-form-urlencoded and includes at least the following parameter:

proof_token **REQUIRED**: A *proof-token* (Section 3.1) as described above.

Additional parameters can be sent according to, and conditioned on, the semantics of the scope(s). Unrecognized or incompatible parameters **SHOULD** be ignored.

The authorization server verifies the request:

1. Parse the proof_token parameter and find its claims;

2. Verify that the proof_token's signature matches the proof-of-possession key associated with the sub claim, and that it hasn't expired;

3. Verify that the aud claim is an absolute URI for a resource in a protection space for which this endpoint is responsible;

4. Verify the nonce claim (for example, by confirming that it was really issued by this system and not too far in the past, that it hasn't been redeemed yet, and that it was issued for a request for the aud claim);

5. Verify the validity and authenticity of the sub claim according to its kind and the semantics of the relevant scope(s);

6. Perform any other processing, verification, and validation appropriate to the relevant scope(s), additional claims, or additional parameters.

If the request is verified, the authorization server issues a bearer access_token valid for the protection space of the original request and for a limited time. The authorization server responds using the common response format (Section 2.2).

3.3. Proof-of-Possession Example

Note: This section is not normative.

A client (for example, an in-browser application working on behalf of a user) attempts an HTTP request to a resource server for an access-restricted URI initially without presenting any special credentials:

```
GET /some/restricted/resource HTTP/1.1
Host: www.example
Origin: https://app.example
```

The resource server does not allow this request without authorization. It generates an unguessable, opaque nonce that the server will be able to later recognize as having generated. The server responds with an HTTP 401 Unauthorized message, and includes the protection space identifier (realm), the nonce, the appropriate scopes, and at least the token_pop_endpoint in the WWW-Authenticate response header with the Bearer method. The server also includes an HTML response body to allow the user to perform a first-party login using another method, for cases where the resource was navigated to directly in the browser:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="/auth/",
  scope="webid openid",
  nonce="j16C4SOLQWFor3VYUtZWnrUr5AG5uwDF7q9RFsDk",
  token_pop_endpoint="/auth/webid-pop",
  client_cert_endpoint="https://webid-tls.example/auth/webid-tls"
Access-Control-Allow-Origin: https://app.example
Access-Control-Expose-Headers: WWW-Authenticate
Date: Mon,  6 May 2019 01:48:48 GMT
Content-type: text/html

<html>Human first-party login page...</html>
```

The client recognizes the response as compatible with this mechanism by recognizing the scheme as Bearer, compatible scopes (in this example, openid and webid), and the presence of the nonce and the token_pop_endpoint.

The client controls a principal appropriate to the scopes (in this example, a JWT substantially similar to an OpenID Connect ID Token [OpenID.Core] and containing a confirmation key [RFC7800]) and determines to use the proof-of-possession mechanism.

The client creates a new *proof-token* JWT as described above (Section 3.1), setting its aud claim to the absolute URI of the original request, the nonce claim to the nonce parameter from the WWW-Authenticate response header, the sub claim to its ID Token, includes other claims as appropriate to the scopes (iss in this example), and signs this *proof-token* with the proof-of-possession key bound to its principal and with a signing algorithm compatible with the signing key and the scopes:

```
{
  "typ": "JWT",
  "alg": "RS256"
}
.
{
  "sub": "eyJhbGciOiJ...",
  "aud": "https://www.example/some/restricted/resource",
  "nonce": "j16C4SOLQWFor3VYUtZWnrUr5AG5uwDF7q9RFsDk",
  "jti": "1C49A92C-C260-4F76-9D7B-E81AE13037B8",
  "iss": "https://app.example/oauth/code"
}
.
RS256-signature-here
```

The client sends a request to the token_pop_endpoint URI and includes the *proof-token*:

```
POST /auth/webid-pop
Host: www.example
Origin: https://app.example
Content-type: application/x-www-form-urlencoded

proof_token=eyJ0eXAiOiJKV1QiCg...
```

The token_pop_endpoint verifies the request as described in Section 3.2, determines that the request is good, and issues a bearer token:

```
HTTP/1.1 200
Content-type: application/json; charset=utf-8
Cache-control: no-cache, no-store
Pragma: no-cache
Access-Control-Allow-Origin: https://app.example
Date: Mon,  6 May 2019 01:48:50 GMT

{
  "access_token": "RPAOmgrWb5wD7DzloDjZ7Ain",
  "expires_in": 1800,
  "token_type": "Bearer"
}
```

The client can now use the access_token in an Authorization header for requests to resources in the same protection space as the original request until the access token expires or is revoked:

```
GET /some/restricted/resource HTTP/1.1
Host: www.example
Origin: https://app.example
Authorization: Bearer RPAOmgrWb5wD7DzloDjZ7Ain
```

The server validates and translates the bearer token in its implementation-specific way, and makes a determination whether to grant the requested access.

4. TLS Client Certificate Mechanism

The client recognizes the availability of, and its compatibility with, this mechanism, by recognizing combinations of challenge scopes with which it is compatible, the presence of the client_cert_endpoint, and either direct control of an appropriate TLS [RFC8446] client certificate and its signing key, or in the case of browser-based Javascript applications, an assumption that such a certificate is configured into the browser and that it will be selected by the user.

The client constructs and sends a token request to the client_cert_endpoint API URI with HTTP POST (Section 4.1), using its TLS client certificate.

The API endpoint validates the request, including aspects of the client certificate, and if appropriate, it responds with a bearer token.

4.1. Client Certificate API

This API endpoint is implemented by the authorization server for the protection space of the original request.

The client uses this API by making an HTTP POST request to the client_cert_endpoint URI. The request body has Content-Type application/x-www-form-urlencoded and includes at least the following parameters:

uri **REQUIRED**: The absolute URI, including scheme, authority (host and optional port), path, and query, but not including fragment identifier, corresponding to the original request that resulted in the HTTP 401 response;

nonce **REQUIRED**: The nonce from the WWW-Authenticate challenge.

Additional parameters can be sent according to, and conditioned on, the semantics of the scope(s). Unrecognized or incompatible parameters **SHOULD** be ignored.

A TLS client certificate is **REQUIRED** when communicating with this API endpoint. That means the origin of this API endpoint will probably be different from that of the original request URI so that the server can request a client certificate in a distinct TLS connection handshake (Section 4.3.2 of [RFC8446]).

The authorization server verifies the request:

1. Verify that uri is an absolute URI and is in a protection space for which this endpoint is responsible;

2. Verify the nonce (for example, confirming that it was really generated by this system, not too far in the past, that it hasn't been redeemed yet, and if possible that it corresponds to a request for uri);

3. Verify the validity and authenticity of the client certificate (beyond those validations required for the TLS connection) according to the semantics of the relevant scope(s);

4. Perform any other processing, verification, and validation appropriate to the relevant scope(s) or additional parameters.

If the request is acceptable, the authorization server issues a bearer access_token valid for the protection space of the original request and for a limited time. The authorization server responds using the common response format (Section 2.2).

4.2. Client Certificate Example

Note: This section is not normative.

A client (for example, an in-browser application working on behalf of a user) attempts an HTTP request to a resource server for an access-restricted URI initially without presenting any special credentials:

```
GET /some/restricted/resource HTTP/1.1
Host: www.example
Origin: https://app.example
```

The resource server does not allow this request without authorization. It generates an unguessable, opaque nonce that the authorization server will be able to later recognize as having generated. The server responds with an HTTP 401 Unauthorized message, and includes the protection space identifier (realm), the nonce, the appropriate scopes, and at least the client_cert_endpoint in the WWW-Authenticate response header with the Bearer method. The server also includes an HTML response body to allow the user to perform a first-party login using another method, for cases where the resource was navigated to directly in the browser:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="/auth/",
  scope="webid openid",
  nonce="j16C4SOLQWFor3VYUtZWnrUr5AG5uwDF7q9RFsDk",
  token_pop_endpoint="/auth/webid-pop",
  client_cert_endpoint="https://webid-tls.example/auth/webid-tls"
Access-Control-Allow-Origin: https://app.example
Access-Control-Expose-Headers: WWW-Authenticate
Date: Mon,  6 May 2019 01:48:48 GMT
Content-type: text/html

<html>Human first-party login page...</html>
```

The client recognizes the response as compatible with this mechanism by recognizing the scheme as Bearer, compatible scopes (in this example, webid), and the presence of the nonce and the client_cert_endpoint.

The client determines to use the client certificate mechanism (for example, by being configured by the user to do so when available, with the assumption the user will choose an appropriate certificate when prompted by the browser).

The client sends, using its TLS client certificate, a token request to the client_cert_endpoint URI and includes the required parameters:

```
POST /auth/webid-tls HTTP/1.1
Host: webid-tls.example
Origin: https://app.example
Content-type: application/x-www-form-urlencoded

uri=https://www.example/some/restricted/resource
&nonce=j16C4SOLQWFor3VYUtZWnrUr5AG5uwDF7q9RFsDk
```

The client_cert_endpoint verifies the request as described in Section 4.1 (in this example, with scope webid, the validation and processing steps further comprise establishing and validating the user's WebID according to [WebID-TLS]). The endpoint determines that the request is good, and issues a bearer token:

```
HTTP/1.1 200
Content-type: application/json; charset=utf-8
Cache-control: no-cache, no-store
Pragma: no-cache
Access-Control-Allow-Origin: https://app.example
Date: Mon,  6 May 2019 01:48:50 GMT

{
  "access_token": "RPAOmgrWb5wD7DzloDjZ7Ain",
  "expires_in": 1800,
  "token_type": "Bearer"
}
```

The client can now use the access_token in an Authorization header for requests to resources in the same protection space as the original request until the bearer token expires or is revoked:

```
GET /some/restricted/resource HTTP/1.1
Host: www.example
Origin: https://app.example
Authorization: Bearer RPAOmgrWb5wD7DzloDjZ7Ain
```

The server validates and translates the bearer token in its implementation-specific way, and makes a determination whether to grant the requested access.

5. IANA Considerations

TBD. Mechanism parameters "token_pop_endpoint" and "client_cert_endpoint" for auth-scheme "Bearer".

6. Security Considerations

When using the Proof-of-Possession mechanism (Section 3), the scope designer should carefully consider whether additional information should go in the *proof-token* (which would therefore be signed) or can be POST parameters (which would not be signed). The safe choice (which therefore **SHOULD** be the default) is to include any additional information in the *proof-token*.

Bearer tokens can be shared freely with other parties by an application. Therefore, a bearer token obtained with the TLS Client Certificate mechanism (Section 4) **MUST NOT** be construed to carry the same weight when authenticating an HTTP request as if the client used the corresponding client certificate for the request's connection. However, particularly for browser-based applications where the application and the resource server(s) are not associated with each other, the user typically doesn't audit the data being sent in HTTP requests (even when a client certificate is used), so the portion of the application running in the browser could be receiving data from anywhere else and sending it over HTTP using the user's client certificate anyway.

Security considerations specific to challenge scopes are beyond the purview of this memo.

7. 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>.

[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>.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <https://www.rfc-editor.org/info/rfc7235>.

[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>.

[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>.

[RFC8259]
Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

8. Informative References

[DID] Reed, D., Sporny, M., Longley, D., Allen, C., Grant, R., and M. Sabadello, "Decentralized Identifiers (DIDs) v1.0", April 2020, <https://www.w3.org/TR/did-core/>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

[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>.

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

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

[WebID] Sambra, A., Story, H., Berners-Lee, T., and S. Corlosquet, Ed., "WebID 1.0: Web Identity and Discovery", March 2014, <https://www.w3.org/2005/Incubator/webid/spec/identity/>.

[WebID-TLS] Inkster, T., Story, H., Harbulot, B., Corlosquet, S., Ed., and A. Sambra, Ed., "WebID Authentication over TLS", March 2014, <https://www.w3.org/2005/Incubator/webid/spec/tls/>.

[VC] Sporny, M., Longley, D., Chadwick, D., Noble, G., Ed., Burnett, D., Ed., and B. Zundel, Ed., "Verifiable Credentials Data Model 1.0", November 2019, <https://www.w3.org/TR/vc-data-model/>.

Author's Address

Michael C. Thornburgh
Adobe
345 Park Avenue

San Jose, CA 95110-2704
United States of America

Phone: +1 408 536 6000
Email: mthornbu@adobe.com
URI: https://zenomt.zenomt.com/card.ttl#me