Identity for WebRTC 1.0

W3C Candidate Recommendation 27 September 2018

This version:
https://www.w3.org/TR/2018/CR-webrtc-identity-20180927/

Latest published version:
https://www.w3.org/TR/webrtc-identity/

Latest editor's draft:
https://w3c.github.io/webrtc-identity/identity.html

Test suite:
https://github.com/web-platform-tests/wpt/tree/master/webrtc

Implementation report:
https://wpt.fyi/webrtc

Previous version:
https://www.w3.org/TR/2018/CR-webrtc-20180621/

Editors:
Cullen Jennings, Cisco
Martin Thomson, Mozilla

Participate:
Mailing list
Browse open issues
IETF RTCWEB Working Group

Abstract

This document defines a set of ECMAScript APIs in WebIDL to allow and application using WebRTC to assert an identity, and to mark media streams as only viewable by

another identity. This specification is being developed in conjunction with a protocol specification developed by the IETF RTCWEB group.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This specification had previously been published as part of the 21 June 2018 Candidate Recommendation of WebRTC 1.0.

While the specification is feature complete and is expected to be stable, there are also a number of known substantive issues on the specification that will be addressed during the Candidate Recommendation period based on implementation experience feedback.

It might also evolve based on feedback gathered as its associated test suite evolves. This test suite will be used to build an implementation report of the API.

To go into Proposed Recommendation status, the group expects to demonstrate implementation of each feature in at least two deployed browsers, and at least one implementation of each optional feature. Mandatory feature with only one implementation may be marked as optional in a revised Candidate Recommendation where applicable.

This document was published by the Web Real-Time Communications Working Group as a Candidate Recommendation. This document is intended to become a W3C Recommendation. Comments regarding this document are welcome. Please send them to public-webrtc@w3.org (subscribe, archives). W3C publishes a Candidate Recommendation to indicate that the document is believed to be stable and to encourage implementation by the developer community. This Candidate Recommendation is expected to advance to Proposed Recommendation no earlier than 31 December 2018.

Please see the Working Group's implementation report.

Publication as a Candidate Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 February 2018 W3C Process Document.

Table of Contents

1. Introduction

This section is non-normative.

This document specifies APIs used for identity in WebRTC.

This specification is being developed in conjunction with a protocol specification developed by the IETF RTCWEB group and an API specification to get access to local media devices [GETUSERMEDIA] developed by the Media Capture Task Force. An overview of the system can be found in [RTCWEB-OVERVIEW] and [RTCWEB-SECURITY].

2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, and *SHOULD* are to be interpreted as described in [RFC2119].

This specification defines conformance criteria that apply to a single product: the **user agent** that implements the interfaces that it contains.

Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)

Implementations that use ECMAScript to implement the APIs defined in this specification *MUST* implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [WEBIDL-1], as this specification uses that specification and terminology.

3. Terminology

The `EventHandler` interface, representing a callback used for event handlers, and the **`ErrorEvent`** interface are defined in [HTML51].

The concepts **queue a task**, **fire a simple event** and **networking task source** are defined in [HTML51].

The terms **event**, **event handlers** and **event handler event types** are defined in [HTML51].

`performance.timeOrigin` and **`performance.now()`** are defined in [HIGHRES-TIME].

The terms **MediaStream**, **`MediaStreamTrack`**, and **`MediaStreamConstraints`** are defined in [GETUSERMEDIA]. Note that `MediaStreamTrack` is extended in `the MediaStreamTrack section` in this document.

The term **Blob** is defined in [FILEAPI].

The term **media description** is defined in [RFC4566].

The term **media transport** is defined in [RFC7656].

The term **generation** is defined in [TRICKLE-ICE] Section 2.

The terms **RTCStatsType**, **stats object** and **monitored object** are defined in [WEBRTC-STATS].

When referring to exceptions, the terms **throw** and **create** are defined in [WEBIDL-1].

The term "throw" is used as specified in [INFRA]: it terminates the current processing steps.

The terms **fulfilled**, **rejected**, **resolved**, **pending** and **settled** used in the context of Promises are defined in [ECMASCRIPT-6.0].

The terms **bundle**, **bundle-only** and **bundle-policy** are defined in [JSEP].

The **OAuth Client** and **Authorization Server** roles are defined in [RFC6749] Section 1.1.

The terms **`RTCPeerConnection`**, **target peer identity**, **setRemoteDescription**, **createOffer**, **createAnswer**, **addTrack** and **RTCError** are defined in [WEBRTC].

Identity

4. Identity Provider Interaction

WebRTC offers and answers (and hence the channels established by `RTCPeerConnection` objects) can be authenticated by using a web-based Identity Provider (IdP). The idea is that the entity sending an offer or answer acts as the Authenticating Party (AP) and obtains an identity assertion from the IdP which it attaches to the session description. The consumer of the session description (i.e., the `RTCPeerConnection` on which `setRemoteDescription` is called) acts as the Relying Party (RP) and verifies the assertion.

The interaction with the IdP is designed to decouple the browser from any particular identity provider; the browser need only know how to load the IdP's JavaScript, the location of which is determined by the IdP's identity, and the generic interface to generating and validating assertions. The IdP provides whatever logic is necessary to bridge the generic protocol to the IdP's specific requirements. Thus, a single browser can support any number of identity protocols, including being forward compatible with IdPs which did not exist at the time the browser was written.

4.1 Identity Provider Selection

An IdP is used to generate an identity assertion as follows:

1. If the `setIdentityProvider()` method has been called, the IdP provided shall be used.
2. If the `setIdentityProvider()` method has not been called, then the user agent *MAY* use an IdP configured into the browser.

In order to verify assertions, the IdP domain name and protocol are taken from the `domain` and `protocol` fields of the identity assertion.

4.2 Instantiating an IdP Proxy

In order to communicate with the IdP, the user agent loads the IdP JavaScript from the IdP. The URI for the IdP script is a well-known URI formed from the “domain” and “protocol” fields, as specified in [RTCWEB-SECURITY-ARCH].

The IdP *MAY* generate an HTTP redirect to another "https" origin, the browser *MUST* treat a redirect to any other scheme as a fatal error.

The user agent instantiates an isolated interpreted context, a JavaScript **realm** that

operates in the origin of the loaded JavaScript. Note that a redirect will change the origin of the loaded script.

The realm is populated with a global that implements both the **RTCIdentityProviderGlobalScope** and WorkerGlobalScope [WEBWORKERS] interfaces.

The user agent provides an instance of RTCIdentityProviderRegistrar named *rtcIdentityProvider* in the global scope of the realm. This object is used by the IdP to interact with the user agent.

WebIDL

```
[Global,
 Exposed=RTCIdentityProviderGlobalScope]
interface RTCIdentityProviderGlobalScope : WorkerGlobalScope {
    readonly attribute RTCIdentityProviderRegistrar
rtcIdentityProvider;
};
```

Attributes

rtcIdentityProvider of type *RTCIdentityProviderRegistrar*, readonly
: This object is used by the IdP to register an RTCIdentityProvider instance with the browser.

4.2.1 Implementing an IdP Securely

An environment that mimics the identity provider realm can be provided by any script. However, only scripts running in the origin of the IdP are able to generate an identical environment. Other origins can load and run the IdP proxy code, but they will be unable to replicate data that is unique to the origin of the IdP.

This means that it is critical that an IdP use data that is restricted to its own origin when generating identity assertions. Otherwise, another origin could load the IdP script and use it to impersonate users.

The data that the IdP script uses could be stored on the client (for example, in [INDEXEDDB]) or loaded from servers. Data that is acquired from a server *SHOULD* require credentials and be protected from cross-origin access.

There is no risk to the integrity of identity assertions if an IdP validates an identity assertion without using origin-private data.

5. Registering an IdP Proxy

An IdP proxy implements the `RTCIdentityProvider` methods, which are the means by which the user agent is able to request that an identity assertion be generated or validated.

Once instantiated, the IdP script is executed. The IdP *MUST* call the `register()` function on the **`RTCIdentityProviderRegistrar`** instance during script execution. If an IdP is not registered during this script execution, the user agent cannot use the IdP proxy and *MUST* fail any future attempt to interact with the IdP.

WebIDL

```
[Exposed=RTCIdentityProviderGlobalScope]
interface RTCIdentityProviderRegistrar {
    void register(RTCIdentityProvider idp);
};
```

Methods

register
: This method is invoked by the IdP when its script is first executed. This registers `RTCIdentityProvider` methods with the user agent.

5.1 Interface Exposed by Identity Providers

The callback functions in `RTCIdentityProvider` are exposed by identity providers and is called by `RTCPeerConnection` to acquire or validate identity assertions.

WebIDL

```
dictionary RTCIdentityProvider {
    required GenerateAssertionCallback generateAssertion;
    required ValidateAssertionCallback validateAssertion;
};
```

Dictionary `RTCIdentityProvider` Members

generateAssertion **of type** `GenerateAssertionCallback`**, required**
: A user agent invokes this method on the IdP to request the generation of an identity assertion.

The IdP provides a promise that resolves to an `RTCIdentityAssertionResult` to successfully generate an identity assertion. Any other value, or a rejected promise, is treated as an error.

***`validateAssertion`* of type `ValidateAssertionCallback`, required**

A user agent invokes this method on the IdP to request the validation of an identity assertion.

The IdP returns a Promise that resolves to an `RTCIdentityValidationResult` to successfully validate an identity assertion and to provide the actual identity. Any other value, or a rejected promise, is treated as an error.

WebIDL

```
callback GenerateAssertionCallback =
Promise<RTCIdentityAssertionResult> (DOMString contents,

DOMString origin,

RTCIdentityProviderOptions options);
```

Callback `GenerateAssertionCallback` Parameters

`contents` of type DOMString

The *contents* parameter includes the information that the user agent wants covered by the identity assertion. The IdP *MUST* treat `contents` as opaque string. A successful validation of the provided assertion *MUST* produce the same string.

`origin` of type DOMString

The *origin* parameter identifies the origin of the `RTCPeerConnection` that triggered this request. An IdP can use this information as input to policy decisions about use. This value is generated by the user agent based on the origin of the document that created the `RTCPeerConnection` and therefore can be trusted to be correct.

`options` of type `RTCIdentityProviderOptions`

This includes the options provided by the application when calling `setIdentityProvider`. Though the dictionary is an optional argument to `setIdentityProvider`, default values are used as necessary when passing the value to the identity provider; see the definition of `RTCIdentityProviderOptions` for details.

WebIDL

```
callback ValidateAssertionCallback =
Promise<RTCIdentityValidationResult> (DOMString assertion,

DOMString origin);
```

Callback `ValidateAssertionCallback` Parameters

`assertion` of type DOMString
: The *assertion* parameter includes the assertion that was recovered from an `a=identity` in the session description; that is, the value that was part of the `RTCIdentityAssertionResult` provided by the IdP that generated the assertion.

`origin` of type DOMString
: The *origin* parameter identifies the origin of the `RTCPeerConnection` that triggered this request. An IdP can use this information as input to policy decisions about use.

5.2 Identity Assertion and Validation Results

WebIDL

```
dictionary RTCIdentityAssertionResult {
    required RTCIdentityProviderDetails idp;
    required DOMString                  assertion;
};
```

Dictionary `RTCIdentityAssertionResult` Members

`idp` of type *RTCIdentityProviderDetails*, required
: An IdP provides these details to identify the IdP that validates the identity assertion. This struct contains the same information that is provided to `setIdentityProvider`.

`assertion` of type DOMString, required
: An identity assertion. This is an opaque string that *MUST* contain all information necessary to assert identity. This value is consumed by the validating IdP.

WebIDL

```
dictionary RTCIdentityProviderDetails {
    required DOMString domain;
```

```
                DOMString protocol = "default";
};
```

Dictionary `RTCIdentityProviderDetails` Members

`domain` of type DOMString, required
: The domain name of the IdP that validated the associated identity assertion.

`protocol` of type DOMString, defaulting to `"default"`
: The protocol parameter used for the IdP. The string *MUST NOT* include the character `'/'` or `'\'`.

WebIDL

```
dictionary RTCIdentityValidationResult {
    required DOMString identity;
    required DOMString contents;
};
```

Dictionary `RTCIdentityValidationResult` Members

`identity` of type DOMString, required
: The validated identity of the peer.

`contents` of type DOMString, required
: The payload of the identity assertion. An IdP that validates an identity assertion *MUST* return the same string that was provided to the original IdP that generated the assertion.

  The user agent uses the *contents* string to determine if the identity assertion matches the session description.

6. Requesting Identity Assertions

The identity assertion request process is triggered by a call to `createOffer`, `createAnswer`, or `getIdentityAssertion`. When these calls are invoked and an identity provider has been set, the following steps are executed:

1. The `RTCPeerConnection` instantiates an IdP as described in Identity Provider Selection and Registering an IdP Proxy. If the IdP cannot be loaded,

instantiated, or the IdP proxy is not registered, this process fails.

2. If the `RTCPeerConnection` was not constructed with a set of certificates, and one has not yet been generated, wait for it to be generated.

3. The `RTCPeerConnection` invokes the `generateAssertion` method on the `RTCIdentityProvider` methods registered by the IdP.

   The `RTCPeerConnection` generates the *contents* parameter to this method as described in [RTCWEB-SECURITY-ARCH]. The value of *contents* includes the fingerprint of the certificate that was selected or generated during the construction of the `RTCPeerConnection`. The *origin* parameter contains the origin of the script that calls the `RTCPeerConnection` method that triggers this behavior. The *usernameHint* value is the same value that is provided to `setIdentityProvider`, if any such value was provided.

4. The IdP proxy returns a Promise to the `RTCPeerConnection`. The IdP proxy is expected to generate the identity assertion asynchronously.

   If the user has been authenticated by the IdP, and the IdP is able to generate an identity assertion, the IdP resolves the promise with an identity assertion in the form of an `RTCIdentityAssertionResult`.

   This step depends entirely on the IdP. The methods by which an IdP authenticates users or generates assertions is not specified, though they could involve interacting with the IdP server or other servers.

5. If the IdP proxy produces an error or returns a promise that does not resolve to a valid `RTCIdentityAssertionResult` (see 8. IdP Error Handling), then assertion generation fails.

6. The `RTCPeerConnection` *MAY* store the identity assertion for use with future offers or answers. If a fresh identity assertion is needed for any reason, applications can create a new `RTCPeerConnection`.

7. If the identity request was triggered by a `createOffer()` or `createAnswer()`, then the assertion is converted to a JSON string, base64-encoded and inserted into an `a=identity` attribute in the session description.

If assertion generation fails, then the promise for the corresponding function call is rejected with a newly created `OperationError`.

6.1 User Login Procedure

An IdP *MAY* reject an attempt to generate an identity assertion if it is unable to verify that a user is authenticated. This might be due to the IdP not having the necessary authentication information available to it (such as cookies).

Rejecting the promise returned by `generateAssertion` will cause the error to propagate to the application. Login errors are indicated by rejecting the promise with an `RTCError` with `errorDetail` set to "idp-need-login".

The URL to login at will be passed to the application in the `idpLoginUrl` attribute of the `RTCPeerConnection`.

An application can load the login URL in an IFRAME or popup window; the resulting page then *SHOULD* provide the user with an opportunity to enter any information necessary to complete the authorization process.

Once the authorization process is complete, the page loaded in the IFRAME or popup sends a message using *postMessage* [webmessaging] to the page that loaded it (through the *window.opener* attribute for popups, or through *window.parent* for pages loaded in an IFRAME). The message *MUST* consist of the *DOMString* "WEBRTC-LOGINDONE". This message informs the application that another attempt at generating an identity assertion is likely to be successful.

7. Verifying Identity Assertions

Identity assertion validation happens when `setRemoteDescription` is invoked on `RTCPeerConnection`. The process runs asynchronously, meaning that validation of an identity assertion might not block the completion of `setRemoteDescription`.

The identity assertion request process involves the following asynchronous steps:

1. The `RTCPeerConnection` awaits any prior identity validation. Only one identity validation can run at a time for an `RTCPeerConnection`. This can happen because the resolution of `setRemoteDescription` is not blocked by identity validation unless there is a target peer identity.

2. The `RTCPeerConnection` loads the identity assertion from the session description and decodes the base64 value, then parses the resulting JSON. The *idp* parameter of the resulting dictionary contains a *domain* and an optional *protocol* value that identifies the IdP, as described in [RTCWEB-SECURITY-ARCH].

3. If the identity assertion is malformed, or if *protocol* includes the character `'/'` or `'\'`, this process fails.

4. The `RTCPeerConnection` instantiates the identified IdP as described in 4.1 Identity Provider Selection and 5. Registering an IdP Proxy. If the IdP cannot be loaded, instantiated or the IdP proxy is not registered, this process fails.

5. The `RTCPeerConnection` invokes the `validateAssertion` method registered by the IdP.

   The *assertion* parameter is taken from the decoded identity assertion. The *origin* parameter contains the origin of the script that calls the `RTCPeerConnection` method that triggers this behavior.

6. The IdP proxy returns a promise and performs the validation process asynchronously.

   The IdP proxy verifies the identity assertion using whatever means necessary. Depending on the authentication protocol this could involve interacting with the IdP server.

7. If the IdP proxy produces an error or returns a promise that does not resolve to a valid `RTCIdentityValidationResult` (see 8. IdP Error Handling), then identity validation fails.

8. Once the assertion is successfully verified, the IdP proxy resolves the promise with an `RTCIdentityValidationResult` containing the validated identity and the original contents that are the payload of the assertion.

9. The `RTCPeerConnection` decodes the `contents` and validates that it contains a fingerprint value for every `a=fingerprint` attribute in the session description. This ensures that the certificate used by the remote peer for communications is covered by the identity assertion.

   > ***NOTE***
   >
   > ***A user agent is required to fail to communicate with peers that offer a certificate that doesn't match an `a=fingerprint` line in the negotiated session description.***

   > ***NOTE***
   >
   > ***The user agent decodes `contents` using the format described in [RTCWEB-SECURITY-ARCH]. However the IdP* MUST *treat `contents` as opaque and return the same string to allow for future extensions.***

10. The `RTCPeerConnection` validates that the domain portion of the identity

matches the domain of the IdP as described in [RTCWEB-SECURITY-ARCH]. If this check fails then the identity validation fails.

11. The `RTCPeerConnection` resolves the `peerIdentity` attribute with a new instance of `RTCIdentityAssertion` that includes the IdP domain and peer identity.

12. The user agent *MAY* display identity information to a user in its UI. Any user identity information that is displayed in this fashion *MUST* use a mechanism that cannot be spoofed by content.

If identity validation fails, the `peerIdentity` promise is rejected with a newly created `OperationError` if it is not settled. Then, if there is no target peer identity, set `peerIdentity` to a new unresolved promise. This permits the use of renegotiation (or a subsequent answer, if the session description was a provisional answer) to resolve or reject the identity.

If identity validation fails and there is a target peer identity for the `RTCPeerConnection`, the promise returned by `setRemoteDescription` is rejected with the same `DOMException`.

8. IdP Error Handling

Errors in IdP processing will - in most cases - result in the failure of the procedure that invoked the IdP proxy. This will result in the rejection of the promise returned by `getIdentityAssertion`, `createOffer`, or `createAnswer`. An IdP proxy error causes a `setRemoteDescription` promise to be rejected if there is a target peer identity; IdP errors in calls to `setRemoteDescription` where there is no target peer identity cause the `peerIdentity` promise to be rejected instead.

If an error occurs these promises are rejected with an `RTCError` if an error occurs in interacting with the IdP proxy. The following scenarios result in errors:

- An `RTCPeerConnection` might be configured with an identity provider, but loading of the IdP URI fails. Any procedure that attempts to invoke such an identity provider and cannot load the URI fails with an `RTCError` with `errorDetail` set to "idp-load-failure" and the httpRequestStatusCode attribute of the error set to the HTTP status code of the response.

- If the IdP loads fails due to the TLS certificate used for the HTTPS connection not being trusted, it fails with an `RTCError` with `errorDetail` set to "idp-tls-failure". This typically happens when the IdP uses certificate pinning and an intermediary such as an enterprise firewall has intercepted the TLS

connection.

- If the script loaded from the identity provider is not valid JavaScript or does not implement the correct interfaces, it causes an IdP failure with an `RTCError` with `errorDetail` set to "idp-bad-script-failure".

- An apparently valid identity provider might fail in several ways. If the IdP token has expired, then the IdP *MUST* fail with an `RTCError` with `errorDetail` set to "idp-token-expired". If the IdP token is not valid, then the IdP *MUST* fail with an `RTCError` with `errorDetail` set to "idp-token-invalid". If an identity provider throws an exception or returns a promise that is ultimately rejected, then the procedure that depends on the IdP *MUST* also fail. These types of errors will cause an IdP failure with an `RTCError` with `errorDetail` set to "idp-execution-failure".

- The user agent *SHOULD* limit the time that it allows for an IdP to 15 seconds. This includes both the loading of the IdP proxy and the identity assertion generation or validation. Failure to do so potentially causes the corresponding operation to take an indefinite amount of time. This timer can be cancelled when the IdP proxy produces a response. Expiration of this timer cases an IdP failure with an `RTCError` with `errorDetail` set to "idp-timeout".

- If the identity provider requires the user to login, the operation will fail `RTCError` with `errorDetail` set to "idp-need-login" and the `idpLoginUrl` attribute of the error set to the URL that can be used to login.

- Even when the IdP proxy produces a positive result, the procedure that uses this information might still fail. Additional validation of an `RTCIdentityValidationResult` value is still necessary. The procedure for validation of identity assertions describes additional steps that are required to successfully validate the output of the IdP proxy.

Any error generated by the IdP *MAY* provide additional information in the `idpErrorInfo` attribute. The information in this string is defined by the IdP in use.

9. RTCPeerConnection Interface Extensions

The Identity API extends the `RTCPeerConnection` interface as described below.

WebIDL

```
partial interface RTCPeerConnection {
    void                setIdentityProvider(DOMString provider,
```

```
                                                optional
RTCIdentityProviderOptions options);
    Promise<DOMString> getIdentityAssertion();
    readonly attribute Promise<RTCIdentityAssertion> peerIdentity;
    readonly attribute DOMString?                    idpLoginUrl;
    readonly attribute DOMString?                    idpErrorInfo;
};
```

Attributes

peerIdentity of type Promise<*RTCIdentityAssertion*>, readonly
: A promise that resolves with the identity of the peer if the identity is successfully validated.

  This promise is rejected if an identity assertion is present in a remote session description and validation of that assertion fails for any reason. If the promise is rejected, a new unresolved value is created, unless a target peer identity has been established. If this promise successfully resolves, the value will not change.

idpLoginUrl of type DOMString, readonly, nullable
: The URL that an application can navigate to so that the user can login to the IdP, as described in 6.1 User Login Procedure.

idpErrorInfo of type DOMString, readonly, nullable
: An attribute that the IdP can use to pass additional information back to the applications about the error. The format of this string is defined by the IdP and may be JSON.

Methods

setIdentityProvider
: Sets the identity provider to be used for a given `RTCPeerConnection` object.

  When the **`setIdentityProvider`** method is invoked, the user agent *MUST* run the following steps:

  1. If the `RTCPeerConnection` object's [[IsClosed]] slot is `true`, throw an `InvalidStateError`.

  2. If *options.protocol* includes the the character `'/'` or `'\'`, throw a `SyntaxError`.

3. Set the current identity provider values to the tuple (`provider`, `options`).

4. If any identity provider value has changed, discard any stored identity assertion.

Identity provider information is not used until an identity assertion is required, either in response to a call to `getIdentityAssertion`, or a session description is requested with a call to either `createOffer` or `createAnswer`.

`getIdentityAssertion`

Initiates the process of obtaining an identity assertion. Applications need not make this call. It is merely intended to allow them to start the process of obtaining identity assertions before a call is initiated. If an identity is needed and an identity provider has been set using the `setIdentityProvider` method, then an identity will be automatically requested when an offer or answer is created.

When `getIdentityAssertion` is invoked, queue a task to run the following steps:

1. If the `RTCPeerConnection` object's [[IsClosed]] slot is `true`, throw an `InvalidStateError`.

2. Request an identity assertion from the IdP.

3. Resolve the promise with the base64 and JSON encoded assertion.

WebIDL

```
dictionary RTCIdentityProviderOptions {
    DOMString protocol = "default";
    DOMString usernameHint;
    DOMString peerIdentity;
};
```

`RTCIdentityProviderOptions` Members

`protocol` of type DOMString

The name of the protocol that is used by the identity provider. This *MUST NOT* include '/' (U+002F) or '\' (U+005C) characters. This value defaults to "default" if not provided.

`usernameHint` of type DOMString

A hint to the identity provider about the identity of the principal for which it should generate an identity assertion. If absent, the value `undefined` is used.

peerIdentity of type DOMString
: The identity of the peer. For identity providers that bind their assertions to a particular pair of communication peers, this allows them to generate an assertion that includes both local and remote identities. If this value is omitted, but a value is provided for the `peerIdentity` member of `RTCConfiguration`, the value from `RTCConfiguration` is used.

WebIDL

```
[Constructor(DOMString idp, DOMString name),
 Exposed=Window]
interface RTCIdentityAssertion {
    attribute DOMString idp;
    attribute DOMString name;
};
```

`RTCIdentityAssertion` Attributes

idp of type DOMString
: The domain name of the identity provider that validated this identity.

name of type DOMString
: An RFC5322-conformant [RFC5322] representation of the verified peer identity. This identity will have been verified via the procedures described in [RTCWEB-SECURITY-ARCH].

10. Media Stream API Extensions for Network Use

10.1 Isolated Media Streams

A MediaStream acquired using `getUserMedia()` is, by default, accessible to an application. This means that the application is able to access the contents of tracks, modify their content, and send that media to any peer it chooses.

WebRTC supports calling scenarios where media is sent to a specifically identified peer, without the contents of media streams being accessible to applications. This is enabled by use of the `peerIdentity` parameter to `getUserMedia()`.

An application willingly relinquishes access to media by including a `peerIdentity` parameter in the `MediaStreamConstraints`. This attribute is set to a `DOMString` containing the identity of a specific peer.

The `MediaStreamConstraints` dictionary is expanded to include the `peerIdentity` parameter.

WebIDL

```
partial dictionary MediaStreamConstraints {
    DOMString peerIdentity;
};
```

Dictionary MediaStreamConstraints Members

peerIdentity of type DOMString
: If set, `peerIdentity` isolates media from the application. Media can only be sent to the identified peer.

A user that is prompted to provide consent for access to a camera or microphone can be shown the value of the `peerIdentity` parameter, so that they can be informed that the consent is more narrowly restricted.

When the **`peerIdentity`** option is supplied to `getUserMedia()`, all of the `MediaStreamTrack`s in the resulting `MediaStream` are isolated so that content is not accessible to any application. Isolated `MediaStreamTrack`s can be used for two purposes:

- Displayed in an appropriate media tag (e.g., a video or audio element). The browser *MUST* ensure that content is inaccessible to the application by ensuring that the resulting content is given the same protections as content that is CORS cross-origin, as described in the relevant Security and privacy considerations section of [HTML51].
- Used as the argument to addTrack on an `RTCPeerConnection` instance, subject to the restrictions in isolated streams and RTCPeerConnection.

A `MediaStreamTrack` that is added to another `MediaStream` remains isolated. When an isolated `MediaStreamTrack` is added to a `MediaStream` with a different peerIdentity, the `MediaStream` gets a combination of isolation restrictions. A `MediaStream` containing `MediaStreamTrack` instances with mixed isolation properties can be displayed, but cannot be sent using `RTCPeerConnection`.

Any `peerIdentity` property *MUST* be retained on cloned copies of

MediaStreamTracks.

10.1.1 Extended MediaStreamTrack Properties

MediaStreamTrack is expanded to include an *isolated* attribute and a corresponding event. This allows an application to quickly and easily determine whether a track is accessible.

WebIDL

```
partial interface MediaStreamTrack {
    readonly attribute boolean      isolated;
             attribute EventHandler onisolationchange;
};
```

Attributes

isolated of type boolean, readonly
: A MediaStreamTrack is isolated (and the corresponding *isolated* attribute set to *true*) when content is inaccessible to the owning document. This occurs as a result of setting the *peerIdentity* option. A track is also isolated if it comes from a cross origin source.

onisolationchange of type EventHandler
: This event handler, of type isolationchange, is fired when the value of the *isolated* attribute changes.

10.1.2 Isolated Streams and RTCPeerConnection

A MediaStreamTrack with a *peerIdentity* option set can be added to any RTCPeerConnection. However, the content of an isolated track *MUST NOT* be transmitted unless all of the following constraints are met:

- A MediaStreamTrack from a stream acquired using the *peerIdentity* option can be transmitted if the RTCPeerConnection has successfully validated the identity of the peer AND that identity is the same identity that was used in the *peerIdentity* option associated with the track. That is, the name attribute of the peerIdentity attribute of the RTCPeerConnection instance *MUST* match the value of the peerIdentity option passed to getUserMedia().

  Rules for matching identity are described in [RTCWEB-SECURITY-ARCH].

- The peer has indicated that it will respect the isolation properties of streams. That is, a DTLS connection with a promise to respect stream confidentiality, as defined in [RTCWEB-ALPN] has been established.

Failing to meet these conditions means that no media can be sent for the affected `MediaStreamTrack`. Video *MUST* be replaced by black frames, audio *MUST* be replaced by silence, and equivalently information-free content *MUST* be provided for other media types.

Remotely sourced `MediaStreamTrack`s *MUST* be isolated if they are received over a DTLS connection that has been negotiated with track isolation. This protects isolated media from the application in the receiving browser. These tracks *MUST* only be displayed to a user using the appropriate media element (e.g., <video> or <audio>).

Any `MediaStreamTrack` that has the *peerIdentity* option set causes all tracks sent using the same `RTCPeerConnection` to be isolated at the receiving peer. All DTLS connections created for an `RTCPeerConnection` with isolated local streams *MUST* be negotiated so that media remains isolated at the remote peer. This causes non-isolated media to become isolated at the receiving peer if any isolated tracks are added to the same `RTCPeerConnection`.

> ***NOTE***
>
> ***Tracks that are not bound to a particular peerIdentity do not cause other streams to be isolated, these tracks simply do not have their content transmitted.***

If a stream becomes isolated after initially being accessible, or an isolated stream is added to an active session, then media for that stream is replaced by information-free content (e.g., black frames or silence).

10.1.3 Protection Afforded by Media Isolation

Media isolation ensures that the content of a `MediaStreamTrack` is not accessible to web applications. However, to ensure that media with a *peerIdentity* option set can be sent to peers, some meta-information about the media will be exposed to applications.

Applications will be able to observe the parameters of the media that affect session negotiation and conversion into RTP. This includes the codecs that might be supported by the track, the bitrate, the number of packets, and the current settings that are set on the `MediaStreamTrack`.

In particular, the statistics that `RTCPeerConnection` records are not reduced in capability. New statistics that might compromise isolation *MUST* be avoided, or explicitly suppressed for isolated streams.

Most of these data are exposed to the network when the media is transmitted. Only the settings for the `MediaStreamTrack` present a new source of information. This can includes the frame rate and resolution of video tracks, the bandwidth of audio tracks, and other information about the source, which would not otherwise be revealed to a network observer. Since settings don't change at a high frequency or in response to changes in media content, settings only reveal limited reveal information about the content of a track. However, any setting that might change dynamically in response to the content of an isolated `MediaStreamTrack` *MUST* have changes suppressed.

11. Identity Examples

The identity system is designed so that applications need not take any special action in order for users to generate and verify identity assertions; if a user has configured an IdP into their browser, then the browser will automatically request/generate assertions and the other side will automatically verify them and display the results. However, applications may wish to exercise tighter control over the identity system as shown by the following examples.

This example shows how to configure the identity provider.

EXAMPLE 1

```
pc.setIdentityProvider('example.com');
```

This example shows how to configure the identity provider with all the options.

EXAMPLE 2

```
pc.setIdentityProvider('example.com', {
  usernameHint: 'alice@example.com',
  peerIdentity: 'bob@example.net'
});
```

This example shows how to consume identity assertions inside a Web application.

EXAMPLE 3

```
async function consumeIdentityAssertion() {
  const identity = await pc.peerIdentity;
  console.log('IdP = ', identity.idp, 'identity =',
identity.name);
}
```

12. Change Log

This section will be removed before publication.

Changes since June 21, 2018

1. This document was split from the [WEBRTC] specification.
2. Editors were changed to Cullen Jennings and Martin Thomson.

A. Acknowledgements

The editors wish to thank the Working Group chairs and Team Contact, Harald Alvestrand, Stefan Håkansson, Erik Lagerway and Dominique Hazaël-Massieux, for their support. Substantial text in this specification was provided by many people including Martin Thomson, Harald Alvestrand, Justin Uberti, Eric Rescorla, Peter Thatcher, Jan-Ivar Bruaroey and Peter Saint-Andre. Dan Burnett would like to acknowledge the significant support received from Voxeo and Aspect during the development of this specification.

B. References

B.1 Normative references

[ECMASCRIPT-6.0]
ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. Allen Wirfs-Brock. Ecma International. June 2015. Standard. URL: http://www.ecma-international.org/ecma-262/6.0/index.html

[fetch]

Fetch Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://fetch.spec.whatwg.org/

[FILEAPI]
File API. Marijn Kruisselbrink. W3C. 26 October 2017. W3C Working Draft. URL: https://www.w3.org/TR/FileAPI/

[GETUSERMEDIA]
Media Capture and Streams. Daniel Burnett; Adam Bergkvist; Cullen Jennings; Anant Narayanan; Bernard Aboba. W3C. 3 October 2017. W3C Candidate Recommendation. URL: https://www.w3.org/TR/mediacapture-streams/

[HIGHRES-TIME]
High Resolution Time Level 2. Ilya Grigorik; James Simonsen; Jatinder Mann. W3C. 1 March 2018. W3C Candidate Recommendation. URL: https://www.w3.org/TR/hr-time-2/

[HTML51]
HTML 5.1 2nd Edition. Steve Faulkner; Arron Eicholz; Travis Leithead; Alex Danilo. W3C. 3 October 2017. W3C Recommendation. URL: https://www.w3.org/TR/html51/

[INFRA]
Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: https://infra.spec.whatwg.org/

[JSEP]
Javascript Session Establishment Protocol. Justin Uberti; Cullen Jennings; Eric Rescorla. IETF. 10 October 2017. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-jsep/

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC4566]
SDP: Session Description Protocol. M. Handley; V. Jacobson; C. Perkins. IETF. July 2006. Proposed Standard. URL: https://tools.ietf.org/html/rfc4566

[RFC6749]
The OAuth 2.0 Authorization Framework. D. Hardt, Ed.. IETF. October 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6749

[RFC7656]
A Taxonomy of Semantics and Mechanisms for Real-Time Transport Protocol (RTP) Sources. J. Lennox; K. Gross; S. Nandakumar; G. Salgueiro; B. Burman, Ed.. IETF. November 2015. Informational. URL: https://tools.ietf.org/html/rfc7656

[RTCWEB-ALPN]
Application Layer Protocol Negotiation for Web Real-Time Communications. M.

Thomson. IETF. 23 July 2014. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-alpn

[RTCWEB-SECURITY-ARCH]
WebRTC Security Architecture. Eric Rescorla. IETF. 10 December 2016. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch

[TRICKLE-ICE]
Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol. E. Ivov; E. Rescorla; J. Uberti. IETF. 20 July 2015. Internet Draft (work in progress). URL: http://datatracker.ietf.org/doc/draft-ietf-mmusic-trickle-ice

[WEBIDL-1]
WebIDL Level 1. Cameron McCormack. W3C. 15 December 2016. W3C Recommendation. URL: https://www.w3.org/TR/2016/REC-WebIDL-1-20161215/

[webmessaging]
HTML5 Web Messaging. Ian Hickson. W3C. 19 May 2015. W3C Recommendation. URL: https://www.w3.org/TR/webmessaging/

[WEBRTC]
WebRTC 1.0: Real-time Communication Between Browsers. Adam Bergkvist; Daniel Burnett; Cullen Jennings; Anant Narayanan; Bernard Aboba; Taylor Brandstetter; Jan-Ivar Bruaroey. W3C. 21 June 2018. W3C Candidate Recommendation. URL: https://www.w3.org/TR/webrtc/

[WEBRTC-STATS]
Identifiers for WebRTC's Statistics API. Harald Alvestrand; Varun Singh. W3C. 3 July 2018. W3C Candidate Recommendation. URL: https://www.w3.org/TR/webrtc-stats/

[WEBWORKERS]
Web Workers. Ian Hickson. W3C. 24 September 2015. W3C Working Draft. URL: https://www.w3.org/TR/workers/

B.2 Informative references

[HTML]
HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INDEXEDDB]
Indexed Database API. Nikunj Mehta; Jonas Sicking; Eliot Graff; Andrei Popescu; Jeremy Orlow; Joshua Bell. W3C. 8 January 2015. W3C Recommendation. URL: https://www.w3.org/TR/IndexedDB/

[RFC5322]

Internet Message Format. P. Resnick, Ed.. IETF. October 2008. Draft Standard. URL: https://tools.ietf.org/html/rfc5322

[RTCWEB-OVERVIEW]
Overview: Real Time Protocols for Brower-based Applications. H. Alvestrand. IETF. 14 February 2014. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-overview

[RTCWEB-SECURITY]
Security Considerations for WebRTC. Eric Rescorla. IETF. 22 January 2014. Active Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-security

[WEBIDL]
Web IDL. Cameron McCormack; Boris Zbarsky; Tobie Langel. W3C. 15 December 2016. W3C Editor's Draft. URL: https://heycam.github.io/webidl/

↑