RTCWEB
Internet-Draft
Intended status: Standards Track
Expires: April 25, 2013

E. Rescorla
RTFM, Inc.

October 22, 2012

RTCWEB Security Architecture
draft-ietf-rtcweb-security-arch-04

Abstract

The Real-Time Communications on the Web (RTCWEB) working group is tasked with standardizing protocols for enabling real-time communications within user-agents using web technologies (e.g JavaScript). The major use cases for RTCWEB technology are real-time audio and/or video calls, Web conferencing, and direct data transfer. Unlike most conventional real-time systems (e.g., SIP-based soft phones) RTCWEB communications are directly controlled by some Web server, which poses new security challenges. For instance, a Web browser might expose a JavaScript API which allows a server to place a video call. Unrestricted access to such an API would allow any site which a user visited to "bug" a user's computer, capturing any activity which passed in front of their camera. [I-D.ietf-rtcweb-security] defines the RTCWEB threat model. This document defines an architecture which provides security within that threat model.

and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 25, 2013.

Copyright Notice

Table of Contents

1. Introduction

The Real-Time Communications on the Web (RTCWEB) working group is tasked with standardizing protocols for real-time communications between Web browsers. The major use cases for RTCWEB technology are real-time audio and/or video calls, Web conferencing, and direct data transfer. Unlike most conventional real-time systems, (e.g., SIP-based[RFC3261] soft phones) RTCWEB communications are directly controlled by some Web server, as shown in Figure 1.

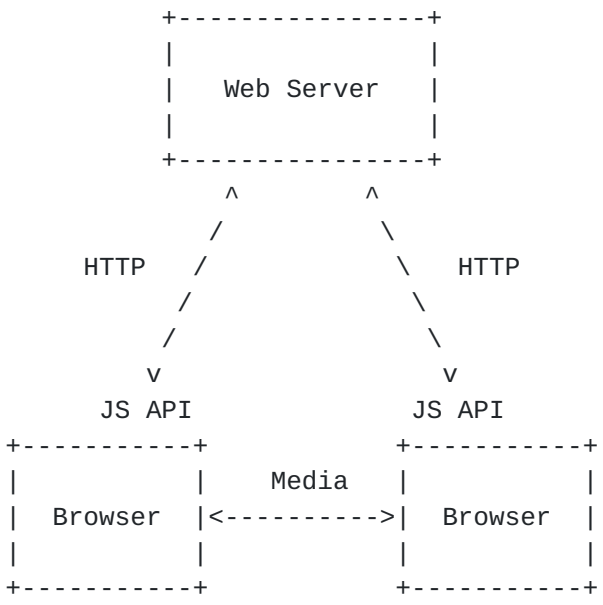


Figure 1: A simple RTCWEB system

This system presents a number of new security challenges, which are analyzed in [I-D.ietf-rtcweb-security]. This document describes a security architecture for RTCWEB which addresses the threats and requirements described in that document.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. Trust Model

The basic assumption of this architecture is that network resources exist in a hierarchy of trust, rooted in the browser, which serves as the user's TRUSTED COMPUTING BASE (TCB). Any security property which

the user wishes to have enforced must be ultimately guaranteed by the browser (or transitively by some property the browser verifies). Conversely, if the browser is compromised, then no security guarantees are possible. Note that there are cases (e.g., Internet kiosks) where the user can't really trust the browser that much. In these cases, the level of security provided is limited by how much they trust the browser.

Optimally, we would not rely on trust in any entities other than the browser. However, this is unfortunately not possible if we wish to have a functional system. Other network elements fall into two categories: those which can be authenticated by the browser and thus are partly trusted--though to the minimum extent necessary--and those which cannot be authenticated and thus are untrusted. This is a natural extension of the end-to-end principle.

3.1. Authenticated Entities

There are two major classes of authenticated entities in the system:

- Calling services: Web sites whose origin we can verify (in practice this means HTTPS).
- Other users: RTCWEB peers whose origin we can verify cryptographically (optimally via DTLS-SRTP).

Note that merely being authenticated does not make these entities trusted. For instance, just because we can verify that https://www.evil.org/ is owned by Dr. Evil does not mean that we can trust Dr. Evil to access our camera and microphone. However, it gives the user an opportunity to determine whether he wishes to trust Dr. Evil or not; after all, if he desires to contact Dr. Evil (perhaps to arrange for ransom payment), it's safe to temporarily give him access to the camera and microphone for the purpose of the call, but he doesn't want Dr. Evil to be able to access his camera and microphone other than during the call. The point here is that we must first identify other elements before we can determine whether and how much to trust them.

It's also worth noting that there are settings where authentication is non-cryptographic, such as other machines behind a firewall. Naturally, the level of trust one can have in identities verified in this way depends on how strong the topology enforcement is.

3.2. Unauthenticated Entities

Other than the above entities, we are not generally able to identify other network elements, thus we cannot trust them. This does not mean that it is not possible to have any interaction with them, but

it means that we must assume that they will behave maliciously and design a system which is secure even if they do so.

4. Overview

This section describes a typical RTCWeb session and shows how the various security elements interact and what guarantees are provided to the user. The example in this section is a "best case" scenario in which we provide the maximal amount of user authentication and media privacy with the minimal level of trust in the calling service. Simpler versions with lower levels of security are also possible and are noted in the text where applicable. It's also important to recognize the tension between security (or performance) and privacy. The example shown here is aimed towards settings where we are more concerned about secure calling than about privacy, but as we shall see, there are settings where one might wish to make different tradeoffs--this architecture is still compatible with those settings.

For the purposes of this example, we assume the topology shown in the figure below. This topology is derived from the topology shown in Figure 1, but separates Alice and Bob's identities from the process of signaling. Specifically, Alice and Bob have relationships with some Identity Provider (IdP) that supports a protocol such OpenID or BrowserID) that can be used to attest to their identity. This separation isn't particularly important in "closed world" cases where Alice and Bob are users on the same social network and have identities based on that network. However, there are important settings where that is not the case, such as federation (calls from one network to another) and calling on untrusted sites, such as where two users who have a relationship via a given social network want to call each other on another, untrusted, site, such as a poker site.

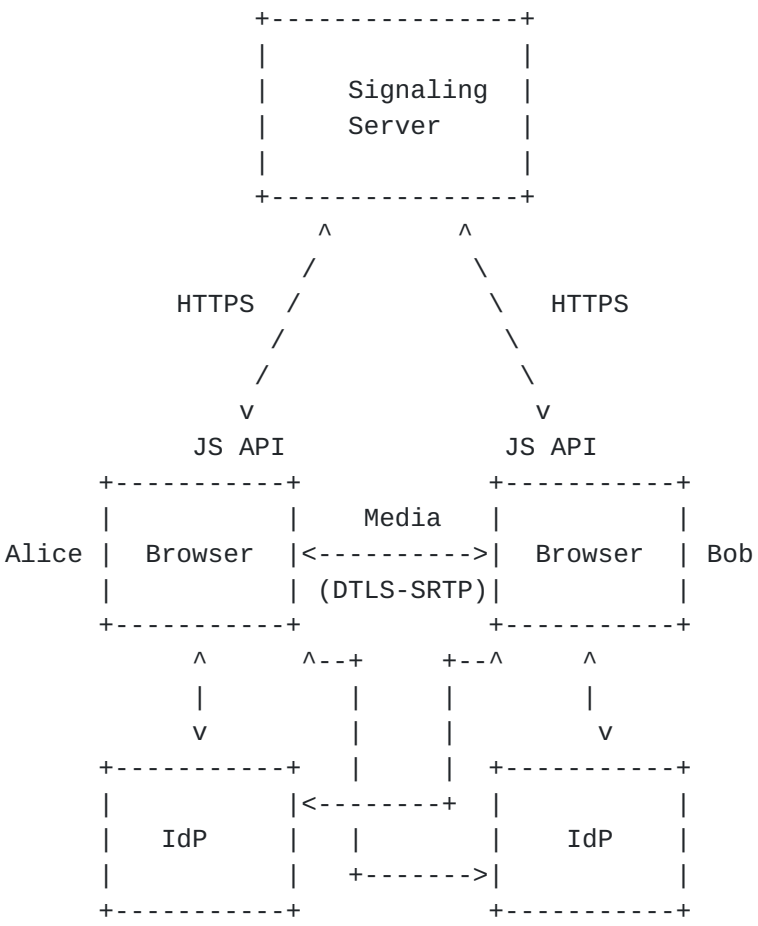


Figure 2: A call with IdP-based identity

4.1. Initial Signaling

Alice and Bob are both users of a common calling service; they both have approved the calling service to make calls (we defer the discussion of device access permissions till later). They are both connected to the calling service via HTTPS and so know the origin with some level of confidence. They also have accounts with some identity provider. This sort of identity service is becoming increasingly common in the Web environment in technologies such (BrowserID, Federated Google Login, Facebook Connect, OAuth, OpenID, WebFinger), and is often provided as a side effect service of your ordinary accounts with some service. In this example, we show Alice and Bob using a separate identity service, though they may actually be using the same identity service as calling service or have no identity service at all.

Alice is logged onto the calling service and decides to call Bob. She can see from the calling service that he is online and the calling service presents a JS UI in the form of a button next to Bob's name

which says "Call". Alice clicks the button, which initiates a JS callback that instantiates a PeerConnection object. This does not require a security check: JS from any origin is allowed to get this far.

Once the PeerConnection is created, the calling service JS needs to set up some media. Because this is an audio/video call, it creates two MediaStreams, one connected to an audio input and one connected to a video input. At this point the first security check is required: untrusted origins are not allowed to access the camera and microphone. In this case, because Alice is a long-term user of the calling service, she has made a permissions grant (i.e., a setting in the browser) to allow the calling service to access her camera and microphone any time it wants. The browser checks this setting when the camera and microphone requests are made and thus allows them.

In the current W3C API, once some streams have been added, Alice's browser + JS generates a signaling message The format of this data is currently undefined. It may be a complete message as defined by ROAP [I-D.jennings-rtcweb-signaling] or separate media description and transport messages as defined in [I-D.ietf-rtcweb-jsep] or may be assembled piecemeal by the JS. In either case, it will contain:

- Media channel information
- ICE candidates
- A fingerprint attribute binding the communication to Alice's public key [RFC5763]

[Note that it is currently unclear where JSEP will eventually put this information, in the SDP or in the transport info.] Prior to sending out the signaling message, the PeerConnection code contacts the identity service and obtains an assertion binding Alice's identity to her fingerprint. The exact details depend on the identity service (though as discussed in Section 5.6 PeerConnection can be agnostic to them), but for now it's easiest to think of as a BrowserID assertion. The assertion may bind other information to the identity besides the fingerprint, but at minimum it needs to bind the fingerprint.

This message is sent to the signaling server, e.g., by XMLHttpRequest [XmlHttpRequest] or by WebSockets [RFC6455] The signaling server processes the message from Alice's browser, determines that this is a call to Bob and sends a signaling message to Bob's browser (again, the format is currently undefined). The JS on Bob's browser processes it, and alerts Bob to the incoming call and to Alice's identity. In this case, Alice has provided an identity assertion and so Bob's browser contacts Alice's identity provider (again, this is done in a generic way so the browser has no specific knowledge of the

IdP) to verify the assertion. This allows the browser to display a trusted element indicating that a call is coming in from Alice. If Alice is in Bob's address book, then this interface might also include her real name, a picture, etc. The calling site will also provide some user interface element (e.g., a button) to allow Bob to answer the call, though this is most likely not part of the trusted UI.

If Bob agrees [I am ignoring early media for now], a PeerConnection is instantiated with the message from Alice's side. Then, a similar process occurs as on Alice's browser: Bob's browser verifies that the calling service is approved, the media streams are created, and a return signaling message containing media information, ICE candidates, and a fingerprint is sent back to Alice via the signaling service. If Bob has a relationship with an IdP, the message will also come with an identity assertion.

At this point, Alice and Bob each know that the other party wants to have a secure call with them. Based purely on the interface provided by the signaling server, they know that the signaling server claims that the call is from Alice to Bob. Because the far end sent an identity assertion along with their message, they know that this is verifiable from the IdP as well. Of course, the call works perfectly well if either Alice or Bob doesn't have a relationship with an IdP; they just get a lower level of assurance. Moreover, Alice might wish to make an anonymous call through an anonymous calling site, in which case she would of course just not provide any identity assertion and the calling site would mask her identity from Bob.

4.2. Media Consent Verification

As described in ([I-D.ietf-rtcweb-security]; Section 4.2) This proposal specifies that media consent verification be performed via ICE. Thus, Alice and Bob perform ICE checks with each other. At the completion of these checks, they are ready to send non-ICE data.

At this point, Alice knows that (a) Bob (assuming he is verified via his IdP) or someone else who the signaling service is claiming is Bob is willing to exchange traffic with her and (b) that either Bob is at the IP address which she has verified via ICE or there is an attacker who is on-path to that IP address detouring the traffic. Note that it is not possible for an attacker who is on-path but not attached to the signaling service to spoof these checks because they do not have the ICE credentials. Bob's security guarantees with respect to Alice are the converse of this.

4.3. DTLS Handshake

Once the ICE checks have completed [more specifically, once some ICE checks have completed], Alice and Bob can set up a secure channel. This is performed via DTLS [RFC4347] (for the data channel) and DTLS-SRTP [RFC5763] for the media channel. Specifically, Alice and Bob perform a DTLS handshake on every channel which has been established by ICE. The total number of channels depends on the amount of muxing; in the most likely case we are using both RTP/RTCP mux and muxing multiple media streams on the same channel, in which case there is only one DTLS handshake. Once the DTLS handshake has completed, the keys are exported [RFC5705] and used to key SRTP for the media channels.

At this point, Alice and Bob know that they share a set of secure data and/or media channels with keys which are not known to any third-party attacker. If Alice and Bob authenticated via their IdPs, then they also know that the signaling service is not attacking them. Even if they do not use an IdP, as long as they have minimal trust in the signaling service not to perform a man-in-the-middle attack, they know that their communications are secure against the signaling service as well.

4.4. Communications and Consent Freshness

From a security perspective, everything from here on in is a little anticlimactic: Alice and Bob exchange data protected by the keys negotiated by DTLS. Because of the security guarantees discussed in the previous sections, they know that the communications are encrypted and authenticated.

The one remaining security property we need to establish is "consent freshness", i.e., allowing Alice to verify that Bob is still prepared to receive her communications. ICE specifies periodic STUN keepalizes but only if media is not flowing. Because the consent issue is more difficult here, we require RTCWeb implementations to periodically send keepalives. As described in Section 5.3, these keepalives MUST be based on the consent freshness mechanism specified in [I-D.muthu-behave-consent-freshness]. If a keepalive fails and no new ICEchannels can be established, then the session is terminated.

5. Detailed Technical Description

5.1. Origin and Web Security Issues

The basic unit of permissions for RTCWEB is the origin [RFC6454]. Because the security of the origin depends on being able to

authenticate content from that origin, the origin can only be securely established if data is transferred over HTTPS [RFC2818]. Thus, clients MUST treat HTTP and HTTPS origins as different permissions domains. [Note: this follows directly from the origin security model and is stated here merely for clarity.]

Many web browsers currently forbid by default any active mixed content on HTTPS pages. I.e., when JS is loaded from an HTTP origin onto an HTTPS page, an error is displayed and the content is not executed unless the user overrides the error. Any browser which enforces such a policy will also not permit access to RTCWEB functionality from mixed content pages. It is RECOMMENDED that browsers which allow active mixed content nevertheless disable RTCWEB functionality in mixed content settings. [[OPEN ISSUE: Should this be a 2119 MUST? It's not clear what set of conditions would make this OK, other than that browser manufacturers have traditionally been permissive here here.]] Note that it is possible for a page which was not mixed content to become mixed content during the duration of the call. Implementations MAY choose to terminate the call or display a warning at that point, but it is also permissible to ignore this condition. This is a deliberate implementation complexity versus security tradeoff. [[OPEN ISSUE:: Should we be more aggressive about this?]]

5.2. Device Permissions Model

Implementations MUST obtain explicit user consent prior to providing access to the camera and/or microphone. Implementations MUST at minimum support the following two permissions models for HTTPS origins.

- Requests for one-time camera/microphone access.
- Requests for permanent access.

Because HTTP origins cannot be securely established against network attackers, implementations MUST NOT allow the setting of permanent access permissions for HTTP origins. Implementations MAY also opt to refuse all permissions grants for HTTP origins, but it is RECOMMENDED that currently they support one-time camera/microphone access.

In addition, they SHOULD support requests for access to a single communicating peer. E.g., "Call customerservice@ford.com". Browsers servicing such requests SHOULD clearly indicate that identity to the user when asking for permission.

API Requirement: The API MUST provide a mechanism for the requesting JS to indicate which of these forms of permissions it is requesting. This allows the client to know what sort of user interface experience to provide, i.e., to allow the client to clearly indicate to the user what he is agreeing to. In particular, browsers might display a non-invasive door hanger ("some features of this site may not work..." when asking for long-term permissions) but a more invasive UI ("here is your own video") for single-call permissions. The API MAY grant weaker permissions than the JS asked for if the user chooses to authorize only those permissions, but if it intends to grant stronger ones it SHOULD display the appropriate UI for those permissions and MUST clearly indicate what permissions are being requested.

API Requirement: The API MUST provide a mechanism for the requesting JS to relinquish the ability to see or modify the media (e.g., via MediaStream.record()). Combined with secure authentication of the communicating peer, this allows a user to be sure that the calling site is not accessing or modifying their conversion.

UI Requirement: The UI MUST clearly indicate when the user's camera and microphone are in use. This indication MUST NOT be suppressable by the JS and MUST clearly indicate how to terminate a call, and provide a UI means to immediately stop camera/ microphone input without the JS being able to prevent it.

UI Requirement: If the UI indication of camera/microphone use are displayed in the browser such that minimizing the browser window would hide the indication, or the JS creating an overlapping window would hide the indication, then the browser SHOULD stop camera and microphone input. [Note: this may not be necessary in systems that are non-windows-based but that have good notifications support, such as phones.]

Clients MAY permit the formation of data channels without any direct user approval. Because sites can always tunnel data through the server, further restrictions on the data channel do not provide any additional security. (though see Section 5.3 for a related issue).

Implementations which support some form of direct user authentication SHOULD also provide a policy by which a user can authorize calls only to specific counterparties. Specifically, the implementation SHOULD provide the following interfaces/controls:

- Allow future calls to this verified user.
- Allow future calls to any verified user who is in my system address book (this only works with address book integration, of course).

Implementations SHOULD also provide a different user interface indication when calls are in progress to users whose identities are directly verifiable. Section 5.5 provides more on this.

5.3. Communications Consent

Browser client implementations of RTCWEB MUST implement ICE. Server gateway implementations which operate only at public IP addresses may implement ICE-Lite instead of ICE but MUST implement one of the two.

Browser implementations MUST verify reachability via ICE prior to sending any non-ICE packets to a given destination. Implementations MUST NOT provide the ICE transaction ID to JavaScript during the lifetime of the transaction (i.e., during the period when the ICE stack would accept a new response for that transaction). [Note: this document takes no position on the split between ICE in JS and ICE in the browser. The above text is written the way it is for editorial convenience and will be modified appropriately if the WG decides on ICE in the JS.]

Implementations MUST send keepalives no less frequently than every 30 seconds regardless of whether traffic is flowing or not. If a keepalive fails then the implementation MUST either attempt to find a new valid path via ICE or terminate media for that ICE component. Note that ICE [RFC5245]; Section 10 keepalives use STUN Binding Indications which are one-way and therefore not sufficient. Instead, the consent freshness mechanism [I-D.muthu-behave-consent-freshness] MUST be used.

5.4. IP Location Privacy

A side effect of the default ICE behavior is that the peer learns one's IP address, which leaks large amounts of location information, especially for mobile devices. This has negative privacy consequences in some circumstances. The API requirements in this section are intended to mitigate this issue. Note that these requirements are NOT intended to protect the user's IP address from a malicious site. In general, the site will learn at least a user's server reflexive address from any HTTP transaction. Rather, these requirements are intended to allow a site to cooperate with the user to hide the user's IP address from the other side of the call. Hiding the user's IP address from the server requires some sort of explicit privacy preserving mechanism on the client (e.g., Torbutton [https://www.torproject.org/torbutton/]) and is out of scope for this specification.

API Requirement: The API MUST provide a mechanism to allow the JS to suppress ICE negotiation (though perhaps to allow candidate gathering) until the user has decided to answer the call [note: determining when the call has been answered is a question for the JS.] This enables a user to prevent a peer from learning their IP address if they elect not to answer a call and also from learning whether the user is online.

API Requirement: The API MUST provide a mechanism for the calling application JS to indicate that only TURN candidates are to be used. This prevents the peer from learning one's IP address at all. The API MUST provide a mechanism for the calling application to reconfigure an existing call to add non-TURN candidates. Taken together, these requirements allow ICE negotiation to start immediately on incoming call notification, thus reducing post-dial delay, but also to avoid disclosing the user's IP address until they have decided to answer. They also allow users to completely hide their IP address for the duration of the call. Finally, they allow a mechanism for the user to optimize performance by reconfiguring to allow non-TURN candidates during an active call if the user decides they no longer need to hide their IP address

5.5. Communications Security

Implementations MUST implement DTLS [RFC4347] and DTLS-SRTP [RFC5763][RFC5764]. All data channels MUST be secured via DTLS. DTLS-SRTP MUST be offered for every media channel and MUST be the default; i.e., if an implementation receives an offer for DTLS-SRTP and SDES, DTLS-SRTP MUST be selected. Media traffic MUST NOT be sent over plain (unencrypted) RTP.

[OPEN ISSUE: What should the settings be here? MUST?] Implementations MAY support SDES and RTP for media traffic for backward compatibility purposes.

API Requirement: The API MUST provide a mechanism to indicate that a fresh DTLS key pair is to be generated for a specific call. This is intended to allow for unlinkability. Note that there are also settings where it is attractive to use the same keying material repeatedly, especially those with key continuity-based authentication.

API Requirement: When DTLS-SRTP is used, the API MUST NOT permit the JS to obtain the negotiated keying material. This requirement preserves the end-to-end security of the media.

UI Requirements: A user-oriented client MUST provide an "inspector" interface which allows the user to determine the security characteristics of the media. [largely derived from [I-D.kaufman-rtcweb-security-ui]

The following properties SHOULD be displayed "up-front" in the browser chrome, i.e., without requiring the user to ask for them:

* A client MUST provide a user interface through which a user may determine the security characteristics for currently-displayed audio and video stream(s)
* A client MUST provide a user interface through which a user may determine the security characteristics for transmissions of their microphone audio and camera video.
* The "security characteristics" MUST include an indication as to whether the cryptographic keys were delivered out-of-band (from a server) or were generated as a result of a pairwise negotiation.
* If the far endpoint was directly verified, either via a third-party verifiable X.509 certificate or via a Web IdP mechanism (see Section 5.6) the "security characteristics" MUST include the verified information.

The following properties are more likely to require some "drill-down" from the user:

* The security characteristics MUST indicate the cryptographic algorithms in use (For example: "AES-CBC" or "Null Cipher".)
* The "security characteristics" MUST indicate whether PFS is provided.
* The "security characteristics" MUST include some mechanism to allow an out-of-band verification of the peer, such as a certificate fingerprint or an SAS.

5.6. Web-Based Peer Authentication

In a number of cases, it is desirable for the endpoint (i.e., the browser) to be able to directly identity the endpoint on the other side without trusting only the signaling service to which they are connected. For instance, users may be making a call via a federated system where they wish to get direct authentication of the other side. Alternately, they may be making a call on a site which they minimally trust (such as a poker site) but to someone who has an identity on a site they do trust (such as a social network.)

Recently, a number of Web-based identity technologies (OAuth, BrowserID, Facebook Connect), etc. have been developed. While the details vary, what these technologies share is that they have a Web-based (i.e., HTTP/HTTPS identity provider) which attests to your identity. For instance, if I have an account at example.org, I could

use the example.org identity provider to prove to others that I was alice@example.org. The development of these technologies allows us to separate calling from identity provision: I could call you on Poker Galaxy but identify myself as alice@example.org.

Whatever the underlying technology, the general principle is that the party which is being authenticated is NOT the signaling site but rather the user (and their browser). Similarly, the relying party is the browser and not the signaling site. Thus, the browser MUST securely generate the input to the IdP assertion process and MUST securely display the results of the verification process to the user in a way which cannot be imitated by the calling site.

In order to make this work, we must standardize the following items:

- The precise information from the signaling message that must be cryptographically bound to the user's identity and a mechanism for carrying assertions in JSEP messages. Section 5.6.3
- The interface to the IdP. Section 5.6.4 specifies a specific protocol mechanism which allows the use of any identity protocol without requiring specific further protocol support in the browser
- The JavaScript interfaces which the calling application can use to specify the IdP to use to generate assertions and to discover what assertions were received.

The first two items are defined in this document. The final one is defined in the companion W3C WebRTC API specification [TODO:REF]

The mechanisms in this document do not require the browser to implement any particular identity protocol or to support any particular IdP. Instead, this document provides a generic interface which any IdP can implement. Thus, new IdPs and protocols can be introduced without change to either the browser or the calling service. This avoids the need to make a commitment to any particular identity protocol, although browsers may opt to directly implement some identity protocols in order to provide superior performance or UI properties.

5.6.1. Trust Relationships: IdPs, APs, and RPs

Any federated identity protocol has three major participants:

Authenticating Party (AP): The entity which is trying to establish its identity.

Identity Provider (IdP): The entity which is vouching for the AP's identity.

Relying Party (RP): The entity which is trying to verify the AP's identity.

The AP and the IdP have an account relationship of some kind: the AP registers with the IdP and is able to subsequently authenticate directly to the IdP (e.g., with a password). This means that the browser must somehow know which IdP(s) the user has an account relationship with. This can either be something that the user configures into the browser or that is configured at the calling site and then provided to the PeerConnection by the calling site.

At a high level there are two kinds of IdPs:

Authoritative: IdPs which have verifiable control of some section of the identity space. For instance, in the realm of e-mail, the operator of "example.com" has complete control of the namespace ending in "@example.com". Thus, "alice@example.com" is whoever the operator says it is. Examples of systems with authoritative identity providers include DNSSEC, RFC 4474, and Facebook Connect (Facebook identities only make sense within the context of the Facebook system).

Third-Party: IdPs which don't have control of their section of the identity space but instead verify user's identities via some unspecified mechanism and then attest to it. Because the IdP doesn't actually control the namespace, RPs need to trust that the IdP is correctly verifying AP identities, and there can potentially be multiple IdPs attesting to the same section of the identity space. Probably the best-known example of a third-party identity provider is SSL certificates, where there are a large number of CAs all of whom can attest to any domain name.

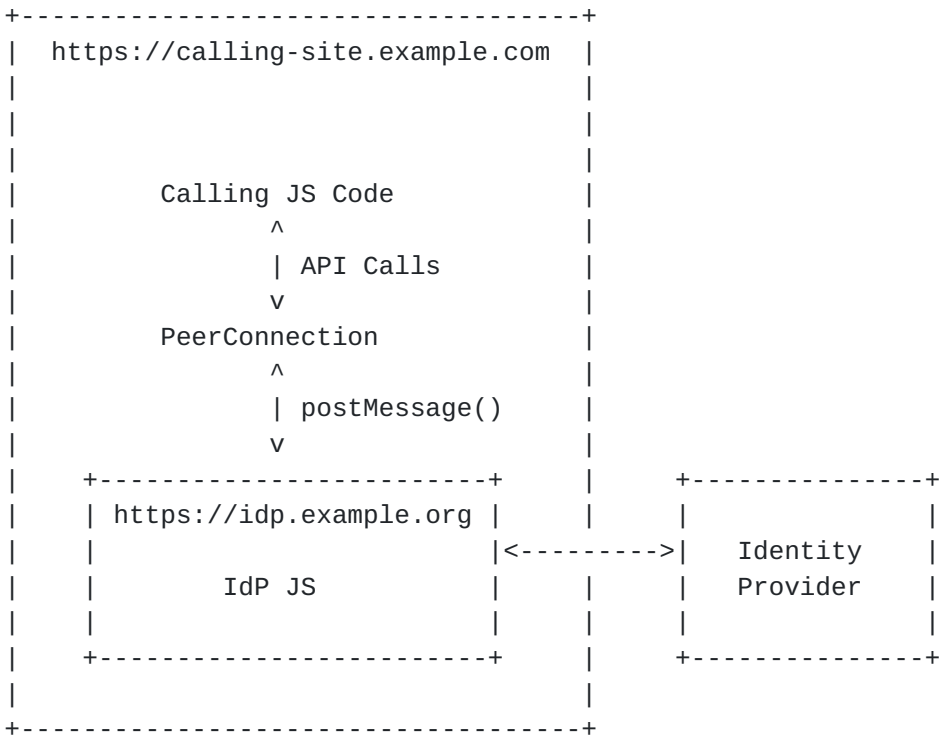
If an AP is authenticating via an authoritative IdP, then the RP does not need to explicitly trust the IdP at all: as long as the RP knows how to verify that the IdP indeed made the relevant identity assertion (a function provided by the mechanisms in this document), then any assertion it makes about an identity for which it is authoritative is directly verifiable.

By contrast, if an AP is authenticating via a third-party IdP, the RP needs to explicitly trust that IdP (hence the need for an explicit trust anchor list in PKI-based SSL/TLS clients). The list of trustable IdPs needs to be configured directly into the browser, either by the user or potentially by the browser manufacturer. This

is a significant advantage of authoritative IdPs and implies that if third-party IdPs are to be supported, the potential number needs to be fairly small.

5.6.2. Overview of Operation

In order to provide security without trusting the calling site, the PeerConnection component of the browser must interact directly with the IdP. The details of the mechanism are described in the W3C API specification, but the general idea is that the PeerConnection component downloads JS from a specific location on the IdP dictated by the IdP domain name. That JS (the "IdP proxy") runs in an isolated security context within the browser and the PeerConnection talks to it via a secure message passing channel.



When the PeerConnection object wants to interact with the IdP, the sequence of events is as follows:

1. The browser (the PeerConnection component) instantiates an IdP proxy with its source at the IdP. This allows the IdP to load whatever JS is necessary into the proxy, which runs in the IdP's security context.
2. If the user is not already logged in, the IdP does whatever is required to log them in, such as soliciting a username and password.

3. Once the user is logged in, the IdP proxy notifies the browser that it is ready.
4. The browser and the IdP proxy communicate via a standardized series of messages delivered via postMessage. For instance, the browser might request the IdP proxy to sign or verify a given identity assertion.

This approach allows us to decouple the browser from any particular identity provider; the browser need only know how to load the IdP's JavaScript--which is deterministic from the IdP's identity--and the generic protocol for requesting and verifying 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.

5.6.3. Binding Identity Assertions to JSEP Offer/Answer Transactions

5.6.3.1. Input to Assertion Generation Process

As discussed above, an identity assertion binds the user's identity (as asserted by the IdP) to the JSEP offer/exchange transaction and specifically to the media. In order to achieve this, the PeerConnection must provide the DTLS-SRTP fingerprint to be bound to the identity. This is provided in a JSON structure for extensibility, as shown below:

```
 {
   "fingerprint" : {
         {
            "algorithm":"SHA-1",
            "digest":"4A:AD:B9:B1:3F:...:E5:7C:AB"
          }
 }
```

The "algorithm" and digest values correspond directly to the algorithm and digest in the a=fingerprint line of the SDP.

Note: this structure does not need to be interpreted by the IdP or the IdP proxy. It is consumed solely by the RP's browser. The IdP merely treats it as an opaque value to be attested to. Thus, new parameters can be added to the assertion without modifying the IdP.

5.6.3.2. Carrying Identity Assertions

Once an IdP has generated an assertion, the JSEP message. This is done by adding a new a-line to the SDP, of the form a=identity. The sole contents of this value are a base-64-encoded version of the identity assertion. For example:

```
v=0
o=- 1181923068 1181923196 IN IP4 ua1.example.com
s=example1
c=IN IP4 ua1.example.com
a=setup:actpass
a=fingerprint: SHA-1 \
  4A:AD:B9:B1:3F:82:18:3B:54:02:12:DF:3E:5D:49:6B:19:E5:7C:AB
a=identity: \
  ImlkcCI6eyJkb21haW4iOiAiZXhhbXBsZS5vcmciLCAicHJvdG9jb2wiOiAiYm9n \
  dXMifSwiYXNzZXJ0aW9uIjpcIntcImlkZW50aXR5XCI6XCJib2JAZXhhbXBsZS5v \
  cmdcIixcImNvbnRlbnRzXCI6XCJhYmNkZWZnaGlqa2xtbm9wcXJzdHV2d3l6XCIs \
  XCJzaWduYXR1cmVcIjpcIjAxMDIwMzA0MDUwNlwifSJ9Cg==
t=0 0
m=audio 6056 RTP/AVP 0
a=sendrecv
a=tcap:1 UDP/TLS/RTP/SAVP RTP/AVP
a=pcfg:1 t=1
```

Each identity attribute should be paired (and attests to) with an a=fingerprint attribute and therefore can exist either at the session or media level. Multiple identity attributes may appear at either level, though implementations are discouraged from doing this unless they have a clear idea of what security claim they intend to be making.

5.6.4. IdP Interaction Details

5.6.4.1. General Message Structure

Messages between the PeerConnection object and the IdP proxy are formatted using JSON [RFC4627]. For instance, the PeerConnection would request a signature with the following "SIGN" message:

```
{
  "type":"SIGN",
  "id": "1",
  "origin":"https://calling-site.example.com",
  "message":"012345678abcdefghijkl"
}
```

All messages MUST contain a "type" field which indicates the general meaning of the message.

All requests from the PeerConnection object MUST contain an "id" field which MUST be unique for that PeerConnection object. Any responses from the IdP proxy MUST contain the same id in response, which allows the PeerConnection to correlate requests and responses.

All requests from the PeerConnection object MUST contain an "origin" field containing the origin of the JS which initiated the PC (i.e., the URL of the calling site). This origin value can be used by the IdP to make access control decisions. For instance, an IdP might only issue identity assertions for certain calling services in the same way that some IdPs require that relying Web sites have an API key before learning user identity.

Any message-specific data is carried in a "message" field. Depending on the message type, this may either be a string or a richer JSON object.

5.6.4.1.1. Errors

If an error occurs, the IdP sends a message of type "ERROR". The message MAY have an "error" field containing freeform text data which containing additional information about what happened. For instance:

```
{
  "type":"ERROR",
  "error":"Signature verification failed"
}
```

Figure 3: Example error

5.6.4.2. IdP Proxy Setup

In order to perform an identity transaction, the PeerConnection must first create an IdP proxy. As stated above, the details of this are specified in the W3C API document. From the perspective of this specification, however, the relevant facts are:

- The JS runs in the IdP's security context with the base page retrieved from the URL specified in Section 5.6.4.2.1
- The usual browser sandbox isolation mechanisms MUST be enforced with respect to the IdP proxy.
- JS running in the IdP proxy MUST be able to send and receive messages to the PeerConnection and the PC and IdP proxy are able to verify the source and destination of these messages.

Initially the IdP proxy is in an unready state; the IdP JS must be loaded and there may be several round trips to the IdP server, for instance to log the user in. When the IdP proxy is ready to receive commands, it delivers a "ready" message. As this message is unsolicited, it simply contains:

```
{ "type":"READY" }
```

Once the PeerConnection object receives the ready message, it can send commands to the IdP proxy.

5.6.4.2.1. Determining the IdP URI

Each IdP proxy instance is associated with two values:

domain name: The IdP's domain name
protocol: The specific IdP protocol which the IdP is using. This is a completely IdP-specific string, but allows an IdP to implement two protocols in parallel. This value may be the empty string.

Each IdP MUST serve its initial entry page (i.e., the one loaded by the IdP proxy) from the well-known URI specified in "/.well-known/idp-proxy/<protocol>" on the IdP's web site. This URI MUST be loaded via HTTPS [RFC2818]. For example, for the IdP "identity.example.com" and the protocol "example", the URL would be:

```
https://example.com/.well-known/idp-proxy/example
```

5.6.4.2.1.1. Authenticating Party

How an AP determines the appropriate IdP domain is out of scope of this specification. In general, however, the AP has some actual account relationship with the IdP, as this identity is what the IdP is attesting to. Thus, the AP somehow supplies the IdP information to the browser. Some potential mechanisms include:

- Provided by the user directly.
- Selected from some set of IdPs known to the calling site. E.g., a button that shows "Authenticate via Facebook Connect"

5.6.4.2.1.2. Relying Party

Unlike the AP, the RP need not have any particular relationship with the IdP. Rather, it needs to be able to process whatever assertion is provided by the AP. As the assertion contains the IdP's identity, the URI can be constructed directly from the assertion, and thus the RP can directly verify the technical validity of the assertion with no user interaction. Authoritative assertions need only be

verifiable. Third-party assertions also MUST be verified against local policy, as described in Section 5.6.4.2.3.1.

5.6.4.2.2. Requesting Assertions

In order to request an assertion, the PeerConnection sends a "SIGN" message. Aside from the mandatory fields, this message has a "message" field containing a string. The contents of this string are defined above, but are opaque from the perspective of the IdP.

A successful response to a "SIGN" message contains a message field which is a JS dictionary dictionary consisting of two fields:

idp: A dictionary containing the domain name of the provider and the protocol string

assertion: An opaque field containing the assertion itself. This is only interpretable by the idp or its proxy.

Figure 4 shows an example transaction, with the message "abcde..." being signed and bound to identity "ekr@example.org". In this case, the message has presumably been digitally signed/MACed in some way that the IdP can later verify it, but this is an implementation detail and out of scope of this document. Line breaks are inserted solely for readability.

```
    PeerConnection -> IdP proxy:
      {
        "type":"SIGN",
        "id":1,
        "message":"abcdefghijklmnopqrstuvwyz"
      }

    IdPProxy -> PeerConnection:
      {
        "type":"SUCCESS",
        "id":1,
        "message": {
          "idp":{
            "domain": "example.org"
            "protocol": "bogus"
          },
          "assertion":\"{\"identity\":\"bob@example.org\",
                        \"contents\":\"abcdefghijklmnopqrstuvwyz\",
                        \"signature\":\"010203040506\"}"
        }
      }
```

Figure 4: Example assertion request

5.6.4.2.3. Verifying Assertions

In order to verify an assertion, an RP sends a "VERIFY" message to the IdP proxy containing the assertion supplied by the AP in the "message" field.

The IdP proxy verifies the assertion. Depending on the identity protocol, this may require one or more round trips to the IdP. For instance, an OAuth-based protocol will likely require using the IdP as an oracle, whereas with BrowserID the IdP proxy can likely verify the signature on the assertion without contacting the IdP, provided that it has cached the IdP's public key.

Regardless of the mechanism, if verification succeeds, a successful response from the IdP proxy MUST contain a message field consisting of a dictionary/hash with the following fields:

identity The identity of the AP from the IdP's perspective. Details of this are provided in Section 5.6.4.2.3.1

contents The original unmodified string provided by the AP in the original SIGN request.

Figure 5 shows an example transaction. Line breaks are inserted solely for readability.

```
PeerConnection -> IdP Proxy:
  {
    "type":"VERIFY",
    "id":2,
    "message":\"{\"identity\":\"bob@example.org\",
                \"contents\":\"abcdefghijklmnopqrstuvwyz\",
                \"signature\":\"010203040506\"}"
  }

IdP Proxy -> PeerConnection:
  {
   "type":"SUCCESS",
   "id":2,
   "message": {
     "identity" : {
       "name" : "bob@example.org",
       "displayname" : "Bob"
     },
     "contents":"abcdefghijklmnopqrstuvwyz"
   }
  }
```

Figure 5: Example verification request

5.6.4.2.3.1. Identity Formats

Identities passed from the IdP proxy to the PeerConnection are structured as JSON dictionaries with one mandatory field: "name". This field MUST consist of an RFC822-formatted string representing the user's identity. [[OPEN ISSUE: Would it be better to have a typed field?]] The PeerConnection API MUST check this string as follows:

1. If the RHS of the string is equal to the domain name of the IdP proxy, then the assertion is valid, as the IdP is authoritative for this domain.
2. If the RHS of the string is not equal to the domain name of the IdP proxy, then the PeerConnection object MUST reject the assertion unless (a) the IdP domain is listed as an acceptable third-party IdP and (b) local policy is configured to trust this IdP domain for the RHS of the identity string.

Sites which have identities that do not fit into the RFC822 style (for instance, Facebook ids are simple numeric values) SHOULD convert them to this form by appending their IdP domain (e.g., 12345@identity.facebook.com), thus ensuring that they are authoritative for the identity.

The IdP proxy MAY also include a "displayname" field which contains a more user-friendly identity assertion. Browsers SHOULD take care in the UI to distinguish the "name" assertion which is verifiable directly from the "displayname" which cannot be verified and thus relies on trust in the IdP. In future, we may define other fields to allow the IdP to provide more information to the browser.

5.7. Security Considerations

Much of the security analysis of this problem is contained in [I-D.ietf-rtcweb-security] or in the discussion of the particular issues above. In order to avoid repetition, this section focuses on (a) residual threats that are not addressed by this document and (b) threats produced by failure/misbehavior of one of the components in the system.

5.7.1. Communications Security

While this document favors DTLS-SRTP, it permits a variety of communications security mechanisms and thus the level of communications security actually provided varies considerably. Any pair of implementations which have multiple security mechanisms in common are subject to being downgraded to the weakest of those common mechanisms by any attacker who can modify the signaling traffic. If communications are over HTTP, this means any on-path attacker. If communications are over HTTPS, this means the signaling server. Implementations which wish to avoid downgrade attack should only offer the strongest available mechanism, which is DTLS/DTLS-SRTP. Note that the implication of this choice will be that interop to non-DTLS-SRTP devices will need to happen through gateways.

Even if only DTLS/DTLS-SRTP are used, the signaling server can potentially mount a man-in-the-middle attack unless implementations have some mechanism for independently verifying keys. The UI requirements in Section 5.5 are designed to provide such a mechanism for motivated/security conscious users, but are not suitable for general use. The identity service mechanisms in Section 5.6 are more suitable for general use. Note, however, that a malicious signaling service can strip off any such identity assertions, though it cannot forge new ones. Note that all of the third-party security mechanisms available (whether X.509 certificates or a third-party IdP) rely on the security of the third party--this is of course also true of your connection to the Web site itself. Users who wish to assure themselves of security against a malicious identity provider can only do so by verifing peer credentials directly, e.g., by checking the peer's fingerprint against a value delivered out of band.

5.7.2. Privacy

The requirements in this document are intended to allow:

- Users to participate in calls without revealing their location.
- Potential callees to avoid revealing their location and even presence status prior to agreeing to answer a call.

However, these privacy protections come at a performance cost in terms of using TURN relays and, in the latter case, delaying ICE. Sites SHOULD make users aware of these tradeoffs.

Note that the protections provided here assume a non-malicious calling service. As the calling service always knows the users status and (absent the use of a technology like Tor) their IP address, they can violate the users privacy at will. Users who wish privacy against the calling sites they are using must use separate privacy enhancing technologies such as Tor. Combined RTCWEB/Tor implementations SHOULD arrange to route the media as well as the signaling through Tor. [Currently this will produce very suboptimal performance.]

5.7.3. Denial of Service

The consent mechanisms described in this document are intended to mitigate denial of service attacks in which an attacker uses clients to send large amounts of traffic to a victim without the consent of the victim. While these mechanisms are sufficient to protect victims who have not implemented RTCWEB at all, RTCWEB implementations need to be more careful.

Consider the case of a call center which accepts calls via RTCWeb. An attacker proxies the call center's front-end and arranges for multiple clients to initiate calls to the call center. Note that this requires user consent in many cases but because the data channel does not need consent, he can use that directly. Since ICE will complete, browsers can then be induced to send large amounts of data to the victim call center if it supports the data channel at all. Preventing this attack requires that automated RTCWEB implememententations implement sensible flow control and have the ability to triage out (i.e., stop responding to ICE probes on) calls which are behaving badly, and especially to be prepared to remotely throttle the data channel in the absence of plausible audio and video (which the attacker cannot control).

Another related attack is for the signaling service to swap the ICE candidates for the audio and video streams, thus forcing a browser to send video to the sink that the other victim expects will contain

audio (perhaps it is only expecting audio!) potentially causing overload. Muxing multiple media flows over a single transport makes it harder to individually suppress a single flow by denying ICE keepalives. Media-level (RTCP) mechanisms must be used in this case.

Yet another attack, suggested by Magnus Westerlund, is for the attacker to cross-connect offers and answers as follows. It induces the victim to make a call and then uses its control of other users browsers to get them to attempt a call to someone. It then translates their offers into apparent answers to the victim, which looks like large-scale parallel forking. The victim still responds to ICE responses and now the browsers all try to send media to the victim. [[OPEN ISSUE: How do we address this?]]

[TODO: Should we have a mechanism for verifying total expected bandwidth]

Note that attacks based on confusing one end or the other about consent are possible primarily even in the face of the third-party identity mechanism as long as major parts of the signaling messages are not signed. On the other hand, signing the entire message severely restricts the capabilities of the calling application, so there are difficult tradeoffs here.

5.7.4. IdP Authentication Mechanism

This mechanism relies for its security on the IdP and on the PeerConnection correctly enforcing the security invariants described above. At a high level, the IdP is attesting that the user identified in the assertion wishes to be associated with the assertion. Thus, it must not be possible for arbitrary third parties to get assertions tied to a user or to produce assertions that RPs will accept.

5.7.4.1. PeerConnection Origin Check

Fundamentally, the IdP proxy is just a piece of HTML and JS loaded by the browser, so nothing stops a Web attacker o from creating their own IFRAME, loading the IdP proxy HTML/JS, and requesting a signature. In order to prevent this attack, we require that all signatures be tied to a specific origin ("rtcweb://...") which cannot be produced by a page tied to a Web attacker. Thus, while an attacker can instantiate the IdP proxy, they cannot send messages from an appropriate origin and so cannot create acceptable assertions. [[OPEN ISSUE: Where is this enforced?]]

5.7.4.2. IdP Well-known URI

As described in Section 5.6.4.2.1 the IdP proxy HTML/JS landing page is located at a well-known URI based on the IdP's domain name. This requirement prevents an attacker who can write some resources at the IdP (e.g., on one's Facebook wall) from being able to impersonate the IdP.

5.7.4.3. Privacy of IdP-generated identities and the hosting site

Depending on the structure of the IdP's assertions, the calling site may learn the user's identity from the perspective of the IdP. In many cases this is not an issue because the user is authenticating to the site via the IdP in any case, for instance when the user has logged in with Facebook Connect and is then authenticating their call with a Facebook identity. However, in other case, the user may not have already revealed their identity to the site. In general, IdPs SHOULD either verify that the user is willing to have their identity revealed to the site (e.g., through the usual IdP permissions dialog) or arrange that the identity information is only available to known RPs (e.g., social graph adjacencies) but not to the calling site. The "origin" field of the signature request can be used to check that the user has agreed to disclose their identity to the calling site; because it is supplied by the PeerConnection it can be trusted to be correct.

5.7.4.4. Security of Third-Party IdPs

As discussed above, each third-party IdP represents a new universal trust point and therefore the number of these IdPs needs to be quite limited. Most IdPs, even those which issue unqualified identities such as Facebook, can be recast as authoritative IdPs (e.g., 123456@facebook.com). However, in such cases, the user interface implications are not entirely desirable. One intermediate approach is to have special (potentially user configurable) UI for large authoritative IdPs, thus allowing the user to instantly grasp that the call is being authenticated by Facebook, Google, etc.

5.7.4.5. Web Security Feature Interactions

A number of optional Web security features have the potential to cause issues for this mechanism, as discussed below.

5.7.4.5.1. Popup Blocking

If the user is not already logged into the IdP, the IdP proxy may need to pop up a top level window in order to prompt the user for their authentication information (it is bad practice to do this in an

IFRAME inside the window because then users have no way to determine the destination for their password). If the user's browser is configured to prevent popups, this may fail (depending on the exact algorithm that the popup blocker uses to suppress popups). It may be necessary to provide a standardized mechanism to allow the IdP proxy to request popping of a login window. Note that care must be taken here to avoid PeerConnection becoming a general escape hatch from popup blocking. One possibility would be to only allow popups when the user has explicitly registered a given IdP as one of theirs (this is only relevant at the AP side in any case). This is what WebIntents does, and the problem would go away if WebIntents is used.

5.7.4.5.2. Third Party Cookies

Some browsers allow users to block third party cookies (cookies associated with origins other than the top level page) for privacy reasons. Any IdP which uses cookies to persist logins will be broken by third-party cookie blocking. One option is to accept this as a limitation; another is to have the PeerConnection object disable third-party cookie blocking for the IdP proxy.

6. Acknowledgements

Bernard Aboba, Harald Alvestrand, Dan Druta, Cullen Jennings, Hadriel Kaplan, Matthew Kaufman, Jim McEachern, Martin Thomson, Magnus Westerland.

7. Changes since -03

The following changes have been made since the -02 draft.

- Editorial changes

8. Changes since -02

The following changes have been made since the -02 draft.

- Forbid persistent HTTP permissions.
- Clarified the text in S 5.4 to clearly refer to requirements on the API to provide functionality to the site.
- Fold in the IETF portion of draft-rescorla-rtcweb-generic-idp

9. References

9.1. Normative References

[I-D.ietf-rtcweb-security]
Rescorla, E., "Security Considerations for RTC-Web", draft-ietf-rtcweb-security-03 (work in progress), June 2012.

[I-D.muthu-behave-consent-freshness]
Perumal, M., Wing, D., and H. Kaplan, "STUN Usage for Consent Freshness and Session Liveness", draft-muthu-behave-consent-freshness-01 (work in progress), July 2012.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, April 2006.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, April 2010.

[RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, May 2010.

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, May 2010.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, December 2011.

9.2. Informative References

[I-D.ietf-rtcweb-jsep]
Uberti, J. and C. Jennings, "Javascript Session Establishment Protocol", draft-ietf-rtcweb-jsep-01 (work in progress), June 2012.

[I-D.jennings-rtcweb-signaling]
Jennings, C., Rosenberg, J., and R. Jesup, "RTCWeb Offer/Answer Protocol (ROAP)", draft-jennings-rtcweb-signaling-01 (work in progress), October 2011.

[I-D.kaufman-rtcweb-security-ui]
Kaufman, M., "Client Security User Interface Requirements for RTCWEB", draft-kaufman-rtcweb-security-ui-00 (work in progress), June 2011.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, March 2010.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, December 2011.

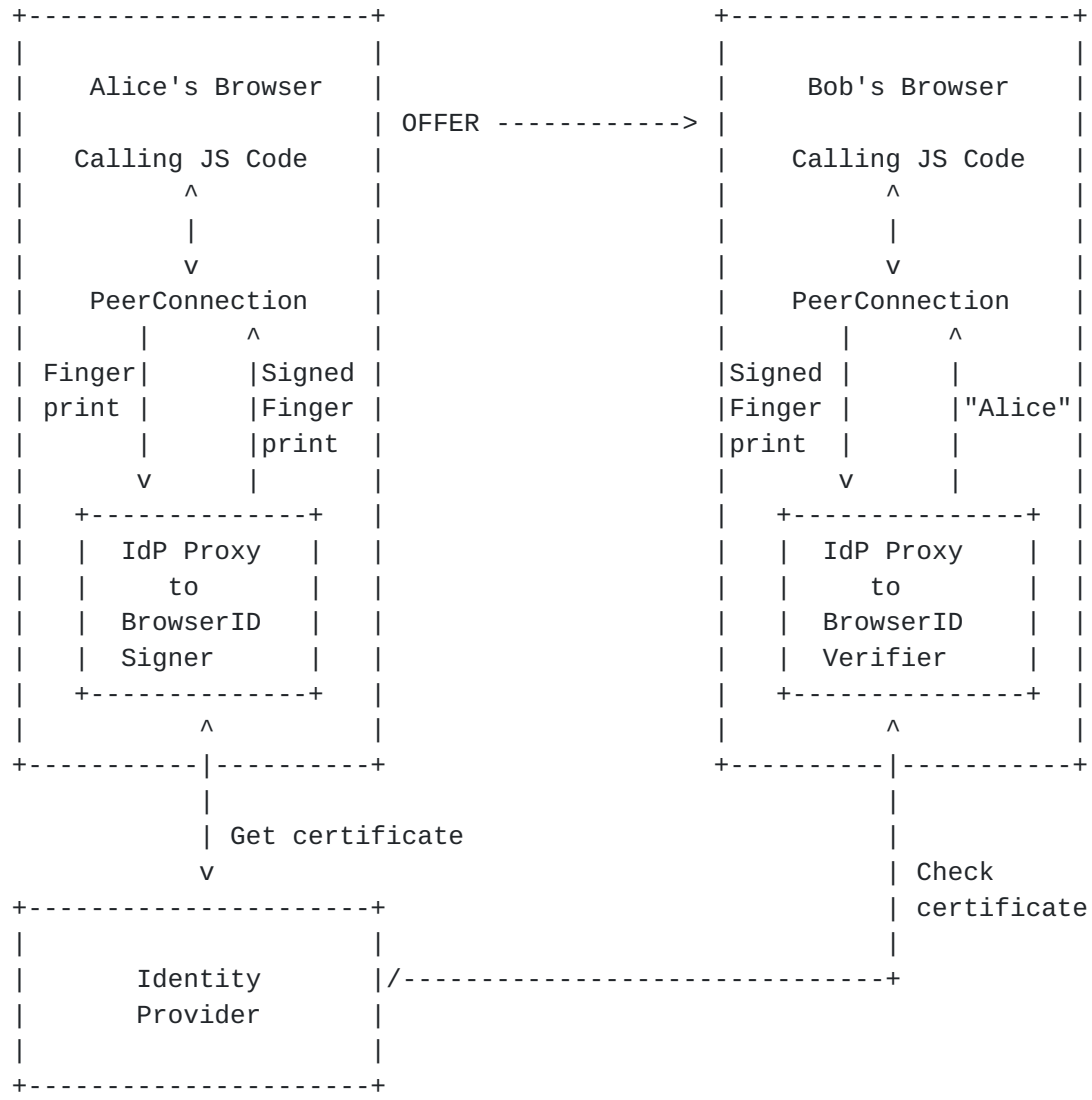
[XmlHttpRequest]
van Kesteren, A., "XMLHttpRequest Level 2".

Appendix A. Example IdP Bindings to Specific Protocols

This section provides some examples of how the mechanisms described in this document could be used with existing authentication protocols such as BrowserID or OAuth. Note that this does not require browser-level support for either protocol. Rather, the protocols can be fit into the generic framework. (Though BrowserID in particular works better with some client side support).

A.1. BrowserID

BrowserID [https://browserid.org/] is a technology which allows a user with a verified email address to generate an assertion (authenticated by their identity provider) attesting to their identity (phrased as an email address). The way that this is used in practice is that the relying party embeds JS in their site which talks to the BrowserID code (either hosted on a trusted intermediary or embedded in the browser). That code generates the assertion which is passed back to the relying party for verification. The assertion can be verified directly or with a Web service provided by the identity provider. It's relatively easy to extend this functionality to authenticate RTCWEB calls, as shown below.



The way this mechanism works is as follows. On Alice's side, Alice goes to initiate a call.

1. The calling JS instantiates a PeerConnection and tells it that it is interested in having it authenticated via BrowserID (i.e., it provides "browserid.org" as the IdP name.)
2. The PeerConnection instantiates the BrowserID signer in the IdP proxy
3. The BrowserID signer contacts Alice's identity provider, authenticating as Alice (likely via a cookie).
4. The identity provider returns a short-term certificate attesting to Alice's identity and her short-term public key.
5. The Browser-ID code signs the fingerprint and returns the signed assertion + certificate to the PeerConnection.

6. The PeerConnection returns the signed information to the calling JS code.
7. The signed assertion gets sent over the wire to Bob's browser (via the signaling service) as part of the call setup.

Obviously, the format of the signed assertion varies depending on what signaling style the WG ultimately adopts. However, for concreteness, if something like ROAP were adopted, then the entire message might look like:

```
{
  "messageType":"OFFER",
  "callerSessionId":"13456789ABCDEF",
  "seq": 1
  "sdp":"
v=0\n
o=- 2890844526 2890842807 IN IP4 192.0.2.1\n
s= \n
c=IN IP4 192.0.2.1\n
t=2873397496 2873404696\n
m=audio 49170 RTP/AVP 0\n
a=fingerprint: SHA-1 \
4A:AD:B9:B1:3F:82:18:3B:54:02:12:DF:3E:5D:49:6B:19:E5:7C:AB\n",
 "identity":{
      "idp":{     // Standardized
         "domain":"browserid.org",
         "method":"default"
      },
      "assertion":   // Contents are browserid-specific
        "\"assertion\": {
          \"digest\":\"<hash of the contents from the browser>\",
          \"audience\": \"[TBD]\"
          \"valid-until\": 1308859352261,
         },
         \"certificate\": {
           \"email\": \"rescorla@example.org\",
           \"public-key\": \"<ekrs-public-key>\",
           \"valid-until\": 1308860561861,
         }" // certificate is signed by example.org
      }
}
```

Note that while the IdP here is specified as "browserid.org", the actual certificate is signed by example.org. This is because BrowserID is a combined authoritative/third-party system in which browserid.org delegates the right to be authoritative (what BrowserID calls primary) to individual domains.

On Bob's side, he receives the signed assertion as part of the call setup message and a similar procedure happens to verify it.

1. The calling JS instantiates a PeerConnection and provides it the relevant signaling information, including the signed assertion.
2. The PeerConnection instantiates the IdP proxy which examines the IdP name and brings up the BrowserID verification code.
3. The BrowserID verifier contacts the identity provider to verify the certificate and then uses the key to verify the signed fingerprint.
4. Alice's verified identity is returned to the PeerConnection (it already has the fingerprint).
5. At this point, Bob's browser can display a trusted UI indication that Alice is on the other end of the call.

When Bob returns his answer, he follows the converse procedure, which provides Alice with a signed assertion of Bob's identity and keying material.

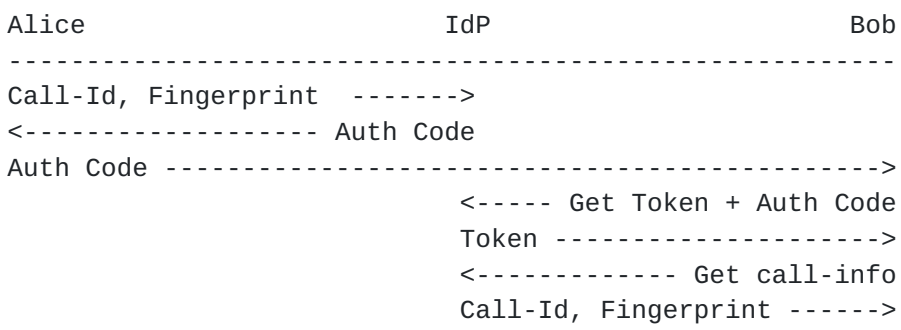
A.2. OAuth

While OAuth is not directly designed for user-to-user authentication, with a little lateral thinking it can be made to serve. We use the following mapping of OAuth concepts to RTCWEB concepts:

OAuth	RTCWEB
Client	Relying party
Resource owner	Authenticating party
Authorization server	Identity service
Resource server	Identity service

Table 1

The idea here is that when Alice wants to authenticate to Bob (i.e., for Bob to be aware that she is calling). In order to do this, she allows Bob to see a resource on the identity provider that is bound to the call, her identity, and her public key. Then Bob retrieves the resource from the identity provider, thus verifying the binding between Alice and the call.



This is a modified version of a common OAuth flow, but omits the redirects required to have the client point the resource owner to the IdP, which is acting as both the resource server and the authorization server, since Alice already has a handle to the IdP.

Above, we have referred to "Alice", but really what we mean is the PeerConnection. Specifically, the PeerConnection will instantiate an IFRAME with JS from the IdP and will use that IFRAME to communicate with the IdP, authenticating with Alice's identity (e.g., cookie). Similarly, Bob's PeerConnection instantiates an IFRAME to talk to the IdP.

Author's Address

Eric Rescorla
RTFM, Inc.
2064 Edgewood Drive
Palo Alto, CA 94303
USA

Phone: +1 650 678 2350
Email: ekr@rtfm.com