WebRTC Security Architecture
draft-ietf-rtcweb-security-arch-13

Abstract

This document defines the security architecture for WebRTC, a protocol suite intended for use with real-time applications that can be deployed in browsers - "real time communication on the Web".

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Real-Time Communications on the Web (WebRTC) working group is tasked with standardizing protocols for real-time communications between Web browsers. The major use cases for WebRTC 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) WebRTC communications are directly controlled by some Web server, via a JavaScript (JS) API as shown in Figure 1.

```
                         +----------------+
                         |                |
                         |   Web Server   |
                         |                |
                         +----------------+
                             ^        ^
                            /          \
                    HTTP   /            \   HTTP
                          /              \
                         /                \
                        v                  v
                     JS API              JS API
               +-----------+            +-----------+
               |           |    Media   |           |
               |  Browser  |<---------->|  Browser  |
               |           |            |           |
               +-----------+            +-----------+
```

Figure 1: A simple WebRTC system

A more complicated system might allow for interdomain calling, as shown in Figure 2. The protocol to be used between the domains is not standardized by WebRTC, but given the installed base and the form of the WebRTC API is likely to be something SDP-based like SIP.

```
              +--------------+             +--------------+
              |              | SIP,XMPP,...|              |
              |  Web Server  |<----------->|  Web Server  |
              |              |             |              |
              +--------------+             +--------------+
                     ^                            ^
                     |                            |
               HTTP  |                            |  HTTP
                     |                            |
                     v                            v
                  JS API                       JS API
               +-----------+                +-----------+
               |           |       Media    |           |
               |  Browser  |<-------------->|  Browser  |
               |           |                |           |
               +-----------+                +-----------+
```

Figure 2: A multidomain WebRTC 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 WebRTC 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 can be granted permissions to access sensitive resources, and those which cannot be authenticated and thus are untrusted.

3.1. Authenticated Entities

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

- Calling services: Web sites whose origin we can verify (optimally via HTTPS, but in some cases because we are on a topologically restricted network, such as behind a firewall, and can infer authentication from firewall behavior).

- Other users: WebRTC 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. Additionally, sometimes we need to identify the communicating peer before we know what policies to apply.

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 WebRTC 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 figures 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 as OpenID Connect) that can be used to demonstrate their identity to other parties. For instance, Alice might have an account with a social network which she can then use to authenticate to other web sites without explicitly having an account with those sites; this is a fairly conventional pattern on the Web. Section 5.6.1 provides an overview of Identity Providers and the relevant terminology. Alice and Bob might have relationships with different IdPs as well.

This separation of identity provision and signaling 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 domain (Figure 3) However, there are important settings where that is not the case, such as federation (calls from one domain to another; Figure 4) 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.

Note that the servers themselves are also authenticated by an external identity service, the SSL/TLS certificate infrastructure (not shown). As is conventional in the Web, all identities are ultimately rooted in that system. For instance, when an IdP makes an identity assertion, the Relying Party consuming that assertion is able to verify because it is able to connect to the IdP via HTTPS.

```
                               +----------------+
                               |                |
                               |     Signaling  |
                               |     Server     |
                               |                |
                               +----------------+
                                   ^        ^
                                  /          \
                          HTTPS  /            \   HTTPS
                                /              \
                               /                \
                              v                  v
                           JS API              JS API
                     +-----------+            +-----------+
                     |           |    Media   |           |
               Alice |  Browser  |<---------->|  Browser  | Bob
                     |           | (DTLS+SRTP)|           |
                     +-----------+            +-----------+
                           ^      ^--+     +--^     ^
                           |         |     |        |
                           v         |     |        v
                     +-----------+   |     |  +-----------+
                     |           |<--------+  |           |
                     |   IdP1    |   |        |    IdP2   |
                     |           |   +------->|           |
                     +-----------+            +-----------+
```

Figure 3: A call with IdP-based identity

Figure 4 shows essentially the same calling scenario but with a call between two separate domains (i.e., a federated case), as in Figure 2. As mentioned above, the domains communicate by some unspecified protocol and providing separate signaling and identity allows for calls to be authenticated regardless of the details of the inter-domain protocol.

```
           +----------------+    Unspecified    +----------------+
           |                |      protocol     |                |
           |    Signaling   |<----------------->|    Signaling   |
           |    Server      |  (SIP, XMPP, ...) |    Server      |
           |                |                   |                |
           +----------------+                   +----------------+
                    ^                                   ^
                    |                                   |
              HTTPS |                                   | HTTPS
                    |                                   |
                    |                                   |
                    v                                   v
                 JS API                               JS API
           +-----------+                             +-----------+
           |           |             Media           |           |
     Alice |  Browser  |<--------------------------->|  Browser  | Bob
           |           |           DTLS+SRTP         |           |
           +-----------+                             +-----------+
                 ^      ^--+                      +--^     ^
                 |         |                      |        |
                 v         |                      |        v
           +-----------+   |                      |  +-----------+
           |           |<-------------------------+  |           |
           |   IdP1    |   |                         |    IdP2   |
           |           |   +------------------------>|           |
           +-----------+                             +-----------+
```

Figure 4: A federated call with IdP-based identity

4.1. Initial Signaling

For simplicity, assume the topology in Figure 3. 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 (with technologies such as Federated Google Login, Facebook Connect, OAuth, OpenID, WebFinger), and is often provided as a side effect service of a user's ordinary accounts with some service. In this example, we show Alice and Bob using a separate identity service, though the identity service may be the same entity as the calling service or there may be 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 a MediaStream with two MediaStreamTracks, 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, so the browser prompts Alice for permission.

In the current W3C API, once some streams have been added, Alice's browser + JS generates a signaling message [I-D.ietf-rtcweb-jsep] containing:

- o Media channel information

- o Interactive Connectivity Establishment (ICE) [RFC5245] candidates

- o A fingerprint attribute binding the communication to a key pair [RFC5763]. Note that this key may simply be ephemerally generated for this call or specific to this domain, and Alice may have a large number of such keys.

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 an OAuth token. 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]. preferably over TLS [RFC5246]. 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 in the browser chrome 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 a PeerConnection is instantiated with the message from Alice's side. Then, a similar process occurs as on Alice's browser: Bob's browser prompts him for device permission, 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. This level of security is provided merely by having the fingerprint in the message and having that message received securely from the signaling server. Because the far end sent an identity assertion along with their message, they know that this is verifiable from the IdP as well. Note that if the call is federated, as shown in Figure 4 then Alice is able to verify Bob's identity in a way that is not mediated by either her signaling server or Bob's. Rather, she verifies it directly with Bob's IdP.

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. I.e., they simply have whatever information their calling site claims about the caller/calllee's identity. 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) media consent verification is provided 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 between Alice and Bob but not attached to the signaling service to spoof these checks

because they do not have the ICE credentials. Bob has the same security guarantees with respect to Alice.

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 or channels. This is performed via DTLS [RFC4347] and DTLS-SRTP [RFC5763] keying for SRTP [RFC3711] for the media channel and SCTP over DTLS [I-D.ietf-tsvwg-sctp-dtls-encaps] for data channels. 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 mounting a man-in-the-middle attack on their traffic. 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 (i.e., that the signaling service cannot mount a passive attack on the communications).

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 so that Alice does not continue to send large traffic volumes to entities which went abruptly offline. ICE specifies periodic STUN keepalives but only if media is not flowing. Because the consent issue is more difficult here, we require WebRTC 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 ICE channels 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 WebRTC 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. That is, when JavaScript is loaded from an HTTP origin onto an HTTPS page, an error is displayed and the HTTP content is not executed unless the user overrides the error. Any browser which enforces such a policy will also not permit access to WebRTC functionality from mixed content pages (because they never display mixed content). Browsers which allow active mixed content MUST nevertheless disable WebRTC functionality in mixed content settings.

Note that it is possible for a page which was not mixed content to become mixed content during the duration of the call. The major risk here is that the newly arrived insecure JS might redirect media to a location controlled by the attacker. Implementations MUST either choose to terminate the call or display a warning at that point.

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.

o Requests for one-time camera/microphone access.

o 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 MUST refuse all permissions grants for HTTP origins.

In addition, they SHOULD support requests for access that promise that media from this grant will be sent to a single communicating peer (obviously there could be other requests for other peers). E.g., "Call customerservice@ford.com". The semantics of this request

are that the media stream from the camera and microphone will only be routed through a connection which has been cryptographically verified (through the IdP mechanism or an X.509 certificate in the DTLS-SRTP handshake) as being associated with the stated identity. Note that it is unlikely that browsers would have an X.509 certificate, but servers might. Browsers servicing such requests SHOULD clearly indicate that identity to the user when asking for permission. The idea behind this type of permissions is that a user might have a fairly narrow list of peers he is willing to communicate with, e.g., "my mother" rather than "anyone on Facebook". Narrow permissions grants allow the browser to do that enforcement.

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 device access, 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 when the indication is hidden. [Note: this may not be necessary in systems that are non-windows-based but that have good notifications support, such as phones.]

o Browsers MUST not permit permanent screen or application sharing permissions to be installed as a response to a JS request for permissions. Instead, they must require some other user action such as a permissions setting or an application install experience to grant permission to a site.

o Browsers MUST provide a separate dialog request for screen/ application sharing permissions even if the media request is made at the same time as camera and microphone.

o The browser MUST indicate any windows which are currently being shared in some unambiguous way. Windows which are not visible MUST not be shared even if the application is being shared. If the screen is being shared, then that MUST be indicated.

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 communicating peers. Specifically, the implementation SHOULD provide the following interfaces/controls:

- o Allow future calls to this verified user.

- o 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 WebRTC MUST implement ICE. Server gateway implementations which operate only at public IP addresses MUST implement either full ICE or ICE-Lite [RFC5245].

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). The JS MUST NOT be permitted to control the local ufrag and password, though it of course knows it.

While continuing consent is required, the ICE [RFC5245]; Section 10 keepalives use STUN Binding Indications which are one-way and therefore not sufficient. The current WG consensus is to use ICE Binding Requests for continuing consent freshness. ICE already requires that implementations respond to such requests, so this approach is maximally compatible. A separate document will profile the ICE timers to be used; see [I-D.muthu-behave-consent-freshness].

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. 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., Tor Browser [https://www.torproject.org/projects/torbrowser.html.en]) 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. This mechanism MUST also permit suppression of the related address field, since that leaks local addresses.

API Requirement: The API MUST provide a mechanism for the calling application to reconfigure an existing call to add non-TURN candidates. Taken together, this and the previous requirement 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

Note that some enterprises may operate proxies and/or NATs designed to hide internal IP addresses from the outside world. WebRTC provides no explicit mechanism to allow this function. Either such enterprises need to proxy the HTTP/HTTPS and modify the SDP and/or the JS, or there needs to be browser support to set the "TURN-only" policy regardless of the site's preferences.

5.5. Communications Security

Implementations MUST implement SRTP [RFC3711]. Implementations MUST implement DTLS [RFC4347] and DTLS-SRTP [RFC5763][RFC5764] for SRTP

keying. Implementations MUST implement [I-D.ietf-tsvwg-sctp-dtls-encaps].

All media channels MUST be secured via SRTP and SRTCP. Media traffic MUST NOT be sent over plain (unencrypted) RTP or RTCP; that is, implementations MUST NOT negotiate cipher suites with NULL encryption modes. DTLS-SRTP MUST be offered for every media channel. WebRTC implementations MUST NOT offer SDP Security Descriptions [RFC4568] or select it if offered. A SRTP MKI MUST NOT be used.

All data channels MUST be secured via DTLS.

All implementations MUST implement DTLS 1.0, with the cipher suite TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA with the the P-256 curve [FIPS186]. The DTLS-SRTP protection profile SRTP_AES128_CM_HMAC_SHA1_80 MUST be supported for SRTP. Implementations SHOULD implement DTLS 1.2 with the TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 cipher suite. Implementations MUST favor cipher suites which support PFS over non-PFS cipher suites and SHOULD favor AEAD over non-AEAD cipher suites.

Implementations MUST NOT implement DTLS renegotiation and MUST reject it with an appropriate alert ("no_renegotiation" for TLS 1.2) if offered.

API Requirement: The API MUST generate a new authentication key pair for every new call by default. This is intended to allow for unlinkability.

API Requirement: The API MUST provide a means to reuse a key pair for calls. This can be used to enable key continuity-based authentication, and could be used to amortize key generation costs.

API Requirement: Unless the user specifically configures an external key pair, different key pairs MUST be used for each origin. (This avoids creating a super-cookie.)

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.

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.

* 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. X.509 identities and Web IdP identities have similar semantics and should be displayed in a similar way.

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".) However, if Null ciphers are used, that MUST be presented to the user at the top-level UI.

* 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 identify the endpoint on the other side without trusting 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, 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 generate the input to the IdP assertion process and display the results of the verification process to the user in a way which cannot be imitated by the calling site.

The mechanisms defined 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 Web application at the calling site. The use case for having this information configured into the browser is that the user may "log into" the browser to bind it to some identity. This is becoming common in new browsers. However, it should also be possible for the IdP information to simply be provided by the calling application.

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 configure trust in the IdP at all. The identity mechanism can directly verify that the IdP indeed made the relevant identity assertion (a function provided by the mechanisms in this document), and any assertion it makes about an identity for which it is authoritative is directly verifiable. Note that this does not mean that the IdP might not lie, but that is a trustworthiness judgement that the user can make at the time he looks at the identity.

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.

Note that there are two logically separate functions here:

- Identity assertion generation.

- Identity assertion verification.

The same IdP JS "endpoint" is used for both functions but of course a given IdP might behave differently and load new JS to perform one function or the other.

```
+------------------------------------+
| Browser                            |
|                                    |
| +--------------------------------+ |
| | https://calling-site.example.com | |
| |                                | |
| |        Calling JS Code         | |
| |               ^                | |
| +---------------|----------------+ |
|                 | API Calls        |
|                 v                  |
|        PeerConnection              |
|                 ^                  |
|                 | API Calls        |
|     +-----------|-------------+    |   +---------------+
|     |           v             |    |   |               |
|     |       IdP Proxy         |<--------->|   Identity    |
|     |                         |    |   |   Provider    |
|     | https://idp.example.org |    |   |               |
|     +-------------------------+    |   +---------------+
|                                    |
+------------------------------------+
```

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. This allows the IdP to load whatever JS is necessary into the proxy. The resulting code runs in the IdP's security context.

2. The IdP registers an object with the browser that conforms to the API defined in [webrtc-api].

3. The browser invokes methods on the object registered by the IdP proxy to create or verify identity assertions.

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--the location of which is determined based on the IdP's identity--and to call the generic API for requesting and verifying identity 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. Items for Standardization

There are two parts to this work:

- 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. This is specified in Section 5.6.4.

- The interface to the IdP, which is defined in the companion W3C WebRTC API specification [webrtc-api].

The WebRTC API specification also defines JavaScript interfaces that the calling application can use to specify which IdP to use. That API also provides access to the assertion-generation capability and the status of the validation process.

5.6.4. Binding Identity Assertions to JSEP Offer/Answer Transactions

An identity assertion binds the user's identity (as asserted by the IdP) to the SDP 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 as a JavaScript object (also known as a dictionary or hash) with a single "fingerprint" key, as shown below:

```
{
  "fingerprint": [ {
    "algorithm": "sha-256",
    "digest": "4A:AD:B9:B1:3F:...:E5:7C:AB"
  }, {
    "algorithm": "sha-1",
    "digest": "74:E9:76:C8:19:...:F4:45:6B"
  } ]
}
```


The "fingerprint" value is an array of objects. Each object in the array contains "algorithm" and "digest" values, which correspond directly to the algorithm and digest values in the "a=fingerprint" line of the SDP [RFC8122].

This object is encoded in a JSON [RFC4627] string for passing to the IdP.

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.

5.6.4.1. Carrying Identity Assertions

Once an IdP has generated an assertion, it is attached to the SDP message. This is done by adding a new identity attribute to the SDP. The sole contents of this value are a base-64 encoded [RFC4648] identity assertion. For example:


```
v=0
o=- 1181923068 1181923196 IN IP4 ua1.example.com
s=example1
c=IN IP4 ua1.example.com
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:\
  eyJpZHAiOnsiZG9tYWluIjoiZXhhbXBsZS5vcmciLCJwcm90b2NvbCI6ImJvZ3Vz\
  In0sImFzc2VydGlvbiI6IntcImlkZW50aXR5XCI6XCJib2JAZXhhbXBsZS5vcmdc\
  IixcImNvbnRlbnRzXCI6XCJhYmNkZWZnaGlqa2xtbm9wcXJzdHV2d3l6XCIsXCJz\
  aWduYXR1cmVcIjpcIjAxMDIwMzA0MDUwNlwifSJ9
a=...
t=0 0
m=audio 6056 RTP/SAVP 0
a=sendrecv
...
```

The identity attribute attests to all "a=fingerprint" attributes in the session description. It is therefore a session-level attribute.

Multiple "a=fingerprint" values can be used to offer alternative certificates for a peer. The "a=identity" attribute MUST include all fingerprint values that are included in "a=fingerprint" lines.

The RP browser MUST verify that the in-use certificate for a DTLS connection is in the set of fingerprints returned from the IdP when verifying an assertion.

5.6.4.2. a=identity Attribute

The identity attribute is session level only. It contains an identity assertion, encoded as a base-64 string [RFC4648].

The syntax of this SDP attribute is defined using Augmented BNF [RFC5234]:

```
identity-attribute  = "identity:" identity-assertion
                      [ SP identity-extension
                        *(";" [ SP ] identity-extension) ]
identity-assertion  = base64
base64              = 1*(ALPHA / DIGIT / "+" / "/" / "=" )
identity-extension  = extension-att-name [ "=" extension-att-value ]
extension-att-name  = token
extension-att-value = 1*(%x01-09 / %x0b-0c / %x0e-3a / %x3c-ff)
                      ; byte-string from [RFC4566] omitting ";"
```

No extensions are defined for this attribute.

The identity assertion is a JSON [RFC4627] encoded dictionary that contains two values. The "assertion" attribute contains an opaque string that is consumed by the IdP. The "idp" attribute is a dictionary with one or two further values that identify the IdP, as described in Section 5.6.5.

5.6.5. Determining the IdP URI

In order to ensure that the IdP is under control of the domain owner rather than someone who merely has an account on the domain owner's server (e.g., in shared hosting scenarios), the IdP JavaScript is hosted at a deterministic location based on the IdP's domain name. 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 opaque IdP-specific string, but allows an IdP to implement two protocols in parallel. This value may be the empty string. If no value for protocol is provided, a value of "default" is used.

Each IdP MUST serve its initial entry page (i.e., the one loaded by the IdP proxy) from a well-known URI [RFC5785]. The well-known URI for an IdP proxy is formed from the following URI components:

1. The scheme, "https:". An IdP MUST be loaded using HTTPS [RFC2818].

2. The authority, which is the IdP domain name. The authority MAY contain a non-default port number. Any port number is removed when determining if an asserted identity matches the name of the IdP. The authority MUST NOT include a userinfo sub-component.

3. The path, starting with "/.well-known/idp-proxy/" and appended with the IdP protocol. Note that the separator characters '/' (%2F) and '\' (%5C) MUST NOT be permitted in the protocol field, lest an attacker be able to direct requests outside of the controlled "/.well-known/" prefix. Query and fragment values MAY be used by including '?' or '#' characters.

For example, for the IdP "identity.example.com" and the protocol "example", the URL would be:

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

The IdP MAY redirect requests to this URL, but they MUST retain the "https" scheme. This changes the effective origin of the IdP, but not the domain of the identities that the IdP is permitted to assert and validate. I.e., the IdP is still regarded as authoritative for the original domain.

5.6.5.1. 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:

- o Provided by the user directly.

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

5.6.5.2. 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 5.7.1.

5.6.6. Requesting Assertions

The input to identity assertion is the JSON-encoded object described in Section 5.6.4 that contains the set of certificate fingerprints the browser intends to use. This string is treated as opaque from the perspective of the IdP.

The browser also identifies the origin that the PeerConnection is run in, which allows the IdP to make decisions based on who is requesting the assertion.

An application can optionally provide a user identifier hint when specifying an IdP. This value is a hint that the IdP can use to select amongst multiple identities, or to avoid providing assertions for unwanted identities. The "username" is a string that has no meaning to any entity other than the IdP, it can contain any data the IdP needs in order to correctly generate an assertion.

An identity assertion that is successfully provided by the IdP consists of the following information:

idp: The domain name of an IdP and the protocol string. This MAY identify a different IdP or protocol from the one that generated the assertion.

assertion: An opaque value containing the assertion itself. This is only interpretable by the identified IdP or the IdP code running in the client.

Figure 5 shows an example assertion formatted as JSON. 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.

```
{
  "idp":{
    "domain": "example.org",
    "protocol": "bogus"
  },
  "assertion": "{\"identity\":\"bob@example.org\",
                 \"contents\":\"abcdefghijklmnopqrstuvwyz\",
                 \"signature\":\"010203040506\"}"
}
```


Figure 5: Example assertion

For use in signaling, the assertion is serialized into JSON, base64-encoded [RFC4648], and used as the value of the "a=identity" attribute.

5.6.7. Managing User Login

In order to generate an identity assertion, the IdP needs proof of the user's identity. It is common practice to authenticate users (using passwords or multi-factor authentication), then use Cookies [RFC6265] or HTTP authentication [RFC2617] for subsequent exchanges.

The IdP proxy is able to access cookies, HTTP authentication or other persistent session data because it operates in the security context of the IdP origin. Therefore, if a user is logged in, the IdP could have all the information needed to generate an assertion.

An IdP proxy is unable to generate an assertion if the user is not logged in, or the IdP wants to interact with the user to acquire more information before generating the assertion. If the IdP wants to interact with the user before generating an assertion, the IdP proxy can fail to generate an assertion and instead indicate a URL where login should proceed.

The application can then load the provided URL to enable the user to enter credentials. The communication between the application and the IdP is described in [webrtc-api].

5.7. Verifying Assertions

The input to identity validation is the assertion string taken from a decoded a=identity attribute.

The IdP proxy verifies the assertion. Depending on the identity protocol, the proxy might contact the IdP server or other servers. For instance, an OAuth-based protocol will likely require using the IdP as an oracle, whereas with a signature-based scheme might be able

to verify the assertion without contacting the IdP, provided that it has cached the relevant public key.

Regardless of the mechanism, if verification succeeds, a successful response from the IdP proxy consists of the following information:

identity: The identity of the AP from the IdP's perspective. Details of this are provided in Section 5.7.1.

contents: The original unmodified string provided by the AP as input to the assertion generation process.

Figure 6 shows an example response formatted as JSON for illustrative purposes.

```
{
  "identity": "bob@example.org",
  "contents": "{\"fingerprint\":[ ... ]}"
}
```

Figure 6: Example verification result

5.7.1. Identity Formats

The identity provided from the IdP to the RP browser MUST consist of a string representing the user's identity. This string is in the form "<user>@<domain>", where "user" consists of any character except '@', and domain is an internationalized domain name [RFC5890].

The PeerConnection API MUST check this string as follows:

1. If the domain portion 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. Comparison of domain names is done using the label equivalence rule defined in Section 2.3.2.4 of [RFC5890].

2. If the domain portion of the string is not equal to the domain name of the IdP proxy, then the PeerConnection object MUST reject the assertion unless:

    1. the IdP domain is trusted as an acceptable third-party IdP; and

    2. local policy is configured to trust this IdP domain for the domain portion of the identity string.

Sites that have identities that do not fit into the RFC822 style (for instance, identifiers that are simple numeric values, or values that contain '@' characters) SHOULD convert them to this form by escaping illegal characters and appending their IdP domain (e.g., user%40133@identity.example.com), thus ensuring that they are authoritative for the identity.

6. 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.

6.1. Communications Security

IF HTTPS is not used to secure communications to the signaling server, and the identity mechanism used in Section 5.6 is not used, then any on-path attacker can replace the DTLS-SRTP fingerprints in the handshake and thus substitute its own identity for that of either endpoint.

Even if HTTPS is 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 verifying peer credentials directly, e.g., by checking the peer's fingerprint against a value delivered out of band.

In order to protect against malicious content JavaScript, that JavaScript MUST NOT be allowed to have direct access to---or perform computations with---DTLS keys. For instance, if content JS were able to compute digital signatures, then it would be possible for content JS to get an identity assertion for a browser's generated key and then use that assertion plus a signature by the key to authenticate a call protected under an ephemeral DH key controlled by the content JS, thus violating the security guarantees otherwise provided by the

IdP mechanism. Note that it is not sufficient merely to deny the content JS direct access to the keys, as some have suggested doing with the WebCrypto API. [webcrypto]. The JS must also not be allowed to perform operations that would be valid for a DTLS endpoint. By far the safest approach is simply to deny the ability to perform any operations that depend on secret information associated with the key. Operations that depend on public information, such as exporting the public key are of course safe.

6.2. Privacy

The requirements in this document are intended to allow:

- o Users to participate in calls without revealing their location.

- o 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 WebRTC/Tor implementations SHOULD arrange to route the media as well as the signaling through Tor. Currently this will produce very suboptimal performance.

Additionally, any identifier which persists across multiple calls is potentially a problem for privacy, especially for anonymous calling services. Such services SHOULD instruct the browser to use separate DTLS keys for each call and also to use TURN throughout the call. Otherwise, the other side will learn linkable information. Additionally, browsers SHOULD implement the privacy-preserving CNAME generation mode of [I-D.ietf-avtcore-6222bis].

6.3. 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 WebRTC at all, WebRTC implementations need to be more careful.

Consider the case of a call center which accepts calls via WebRTC. 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 WebRTC implementations 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. Either media-level (RTCP) mechanisms must be used or the implementation must deny responses entirely, thus terminating the call.

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. Implementations can defend themselves from this attack by only responding to ICE Binding Requests for a limited number of remote ufrags (this is the reason for the requirement that the JS not be able to control the ufrag and password).

[I-D.ietf-rtcweb-rtp-usage] Section 13 documents a number of potential RTCP-based DoS attacks and countermeasures.

Note that attacks based on confusing one end or the other about consent are possible 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.

6.4. 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.

6.4.1. 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 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 content JavaScript. Thus, while an attacker can instantiate the IdP proxy, they cannot send messages from an appropriate origin and so cannot create acceptable assertions. I.e., the assertion request must have come from the browser. This origin check is enforced on the relying party side, not on the authenticating party side. The reason for this is to take the burden of knowing which origins are valid off of the IdP, thus making this mechanism extensible to other applications besides WebRTC. The IdP simply needs to gather the origin information (from the posted message) and attach it to the assertion.

Note that although this origin check is enforced on the RP side and not at the IdP, it is absolutely imperative that it be done. The mechanisms in this document rely on the browser enforcing access restrictions on the DTLS keys and assertion requests which do not come with the right origin may be from content JS rather than from browsers, and therefore those access restrictions cannot be assumed.

Note that this check only asserts that the browser (or some other entity with access to the user's authentication data) attests to the request and hence to the fingerprint. It does not demonstrate that the browser has access to the associated private key. However, attaching one's identity to a key that the user does not control does not appear to provide substantial leverage to an attacker, so a proof of possession is omitted for simplicity.

6.4.2. IdP Well-known URI

As described in Section 5.6.5 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.

6.4.3. 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.

6.4.4. 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.

6.4.5. Web Security Feature Interactions

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

6.4.5.1. Popup Blocking

The IdP proxy is unable to generate popup windows, dialogs or any other form of user interactions. This prevents the IdP proxy from being used to circumvent user interaction. The "LOGINNEEDED" message allows the IdP proxy to inform the calling site of a need for user login, providing the information necessary to satisfy this requirement without resorting to direct user interaction from the IdP proxy itself.

6.4.5.2. 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.

7. IANA Considerations

This specification defines the "identity" SDP attribute per the procedures of Section 8.2.4 of [RFC4566]. The required information for the registration is included here:

Contact Name: Eric Rescorla (ekr@rftm.com)

Attribute Name: identity

Long Form: identity

Type of Attribute: session-level

Charset Considerations: This attribute is not subject to the charset attribute.

Purpose: This attribute carries an identity assertion, binding an identity to the transport-level security session.

Appropriate Values: See Section 5.6.4.2 of RFCXXXX [[Editor Note: This document.

8. Acknowledgements

Bernard Aboba, Harald Alvestrand, Richard Barnes, Dan Druta, Cullen Jennings, Hadriel Kaplan, Matthew Kaufman, Jim McEachern, Martin Thomson, Magnus Westerland. Matthew Kaufman provided the UI material in Section 5.5.

9. Changes

9.1. Changes since -10

Update cipher suite profiles.

Rework IdP interaction based on implementation experience in Firefox.

9.2. Changes since -06

Replaced RTCWEB and RTC-Web with WebRTC, except when referring to the IETF WG

Forbade use in mixed content as discussed in Orlando.

Added a requirement to surface NULL ciphers to the top-level.

Tried to clarify SRTP versus DTLS-SRTP.

Added a section on screen sharing permissions.

Assorted editorial work.

9.3. Changes since -05

The following changes have been made since the -05 draft.

- o Response to comments from Richard Barnes
- o More explanation of the IdP security properties and the federation use case.
- o Editorial cleanup.

9.4. Changes since -03

Version -04 was a version control mistake. Please ignore.

The following changes have been made since the -04 draft.

- o Move origin check from IdP to RP per discussion in YVR.
- o Clarified treatment of X.509-level identities.
- o Editorial cleanup.

9.5. Changes since -03

9.6. Changes since -02

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

- o Forbid persistent HTTP permissions.
- o Clarified the text in S 5.4 to clearly refer to requirements on the API to provide functionality to the site.

- o Fold in the IETF portion of draft-rescorla-rtcweb-generic-idp
- o Retarget the continuing consent section to assume Binding Requests
- o Added some more privacy and linkage text in various places.
- o Editorial improvements

10. References

10.1. Normative References

[FIPS186] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", NIST PUB 186-4 , July 2013.

[I-D.ietf-avtcore-6222bis] Begen, A., Perkins, C., Wing, D., and E. Rescorla, "Guidelines for Choosing RTP Control Protocol (RTCP) Canonical Names (CNAMEs)", draft-ietf-avtcore-6222bis-06 (work in progress), July 2013.

[I-D.ietf-rtcweb-rtp-usage] Perkins, C., Westerlund, M., and J. Ott, "Web Real-Time Communication (WebRTC): Media Transport and Use of RTP", draft-ietf-rtcweb-rtp-usage-26 (work in progress), March 2016.

[I-D.ietf-rtcweb-security] Rescorla, E., "Security Considerations for WebRTC", draft-ietf-rtcweb-security-09 (work in progress), October 2017.

[I-D.ietf-tsvwg-sctp-dtls-encaps] Tuexen, M., Stewart, R., Jesup, R., and S. Loreto, "DTLS Encapsulation of SCTP Packets", draft-ietf-tsvwg-sctp-dtls-encaps-09 (work in progress), January 2015.

[I-D.muthu-behave-consent-freshness] Perumal, M., Wing, D., R, R., and T. Reddy, "STUN Usage for Consent Freshness", draft-muthu-behave-consent-freshness-04 (work in progress), July 2013.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <https://www.rfc-editor.org/info/rfc3711>.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <https://www.rfc-editor.org/info/rfc4347>.

[RFC4566] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, DOI 10.17487/RFC4566, July 2006, <https://www.rfc-editor.org/info/rfc4566>.

[RFC4568] Andreasen, F., Baugher, M., and D. Wing, "Session Description Protocol (SDP) Security Descriptions for Media Streams", RFC 4568, DOI 10.17487/RFC4568, July 2006, <https://www.rfc-editor.org/info/rfc4568>.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, DOI 10.17487/RFC4627, July 2006, <https://www.rfc-editor.org/info/rfc4627>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

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

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <https://www.rfc-editor.org/info/rfc5245>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[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, DOI 10.17487/RFC5763, May 2010, <https://www.rfc-editor.org/info/rfc5763>.

[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, DOI 10.17487/RFC5764, May 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <https://www.rfc-editor.org/info/rfc5785>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/info/rfc6454>.

[RFC8122] Lennox, J. and C. Holmberg, "Connection-Oriented Media Transport over the Transport Layer Security (TLS) Protocol in the Session Description Protocol (SDP)", RFC 8122, DOI 10.17487/RFC8122, March 2017, <https://www.rfc-editor.org/info/rfc8122>.

[webcrypto]
Dahl, Sleevi, "Web Cryptography API", June 2013.

Available at http://www.w3.org/TR/WebCryptoAPI/

[webrtc-api]
Bergkvist, Burnett, Jennings, Narayanan, "WebRTC 1.0: Real-time Communication Between Browsers", October 2011.

Available at http://dev.w3.org/2011/webrtc/editor/webrtc.html

10.2. Informative References

[I-D.ietf-rtcweb-jsep]
Uberti, J., Jennings, C., and E. Rescorla, "JavaScript Session Establishment Protocol", draft-ietf-rtcweb-jsep-24 (work in progress), October 2017.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, DOI 10.17487/RFC2617, June 1999, <https://www.rfc-editor.org/info/rfc2617>.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <https://www.rfc-editor.org/info/rfc3261>.

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <https://www.rfc-editor.org/info/rfc5705>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <https://www.rfc-editor.org/info/rfc6455>.

[XmlHttpRequest]
van Kesteren, A., "XMLHttpRequest Level 2", January 2012.

Appendix A. Example IdP Bindings to Specific Protocols

[[TODO: These still need some cleanup.]]

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

A.1. 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 WebRTC concepts:

OAuth	WebRTC
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.

```
Alice                        IdP                          Bob
---------------------------------------------------------
Call-Id, Fingerprint  ------->
<-------------------- Auth Code
Auth Code ---------------------------------------------->
                             <----- Get Token + Auth Code
                             Token --------------------->
                             <------------- Get call-info
                             Call-Id, Fingerprint ------>
```

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